



PCAN-UDS 2.x API

API Implementation of the UDS Standard
(ISO 14229-1:2013)

User Manual

PCAN® is a registered trademark of PEAK-System Technik GmbH. Other product names in this document may be the trademarks or registered trademarks of their respective companies. They are not explicitly marked by ™ or ®.

© 2024 PEAK-System Technik GmbH

Duplication (copying, printing, or other forms) and the electronic distribution of this document is only allowed with explicit permission of PEAK-System Technik GmbH. PEAK-System Technik GmbH reserves the right to change technical data without prior announcement. The general business conditions and the regulations of the license agreement apply. All rights are reserved.

PEAK-System Technik GmbH
Leydheckerstraße 10
64293 Darmstadt
Germany

Phone: +49 6151 8173-20
Fax: +49 6151 8173-29

www.peak-system.com
info@peak-system.com

Technical support:
E-mail: support@peak-system.com
Forum: forum.peak-system.com

Document version 2.3.3 (2024-05-17)

Contents

| | | |
|----------|--|-----------|
| 1 | PCAN-UDS 2.x API Documentation | 10 |
| 2 | Introduction | 11 |
| 2.1 | Understanding PCAN-UDS 2.x | 11 |
| 2.2 | Using PCAN-UDS 2.x | 12 |
| 2.3 | Features | 12 |
| 2.4 | System Requirements | 12 |
| 2.5 | Scope of Supply | 12 |
| 2.6 | Backward Compatibility Notes | 13 |
| 2.6.1 | PCAN-ISO-TP Compatibility | 13 |
| 2.6.2 | Binary Compatibility | 13 |
| 2.6.3 | Code Compatibility | 13 |
| 2.6.4 | Remark | 14 |
| 3 | DLL API Reference | 15 |
| 3.1 | Namespaces | 15 |
| 3.1.1 | Peak.Can.Uds | 15 |
| 3.2 | Units | 17 |
| 3.2.1 | PUDS Unit | 17 |
| 3.3 | Classes | 19 |
| 3.3.1 | UDSApi | 19 |
| 3.3.2 | TUDSApi | 19 |
| 3.4 | Structures | 21 |
| 3.4.1 | uds_msg | 21 |
| 3.4.2 | uds_sessioninfo | 22 |
| 3.4.3 | uds_netaddrinfo | 24 |
| 3.4.4 | uds_mapping | 25 |
| 3.4.5 | uds_msgconfig | 27 |
| 3.4.6 | uds_msgaccess | 28 |
| 3.5 | Types | 30 |
| 3.5.1 | uds_errstatus | 31 |
| 3.5.2 | uds_status | 32 |
| 3.5.3 | uds_parameter | 41 |
| 3.5.4 | uds_service | 53 |
| 3.5.5 | uds_address | 56 |
| 3.5.6 | uds_can_id | 58 |
| 3.5.7 | uds_status_offset | 60 |
| 3.5.8 | uds_msgprotocol | 61 |
| 3.5.9 | uds_msgtype | 63 |
| 3.5.10 | uds_nrc | 64 |
| 3.5.11 | uds_svc_param_dsc | 71 |
| 3.5.12 | uds_svc_param_er | 72 |
| 3.5.13 | uds_svc_param_cc | 74 |
| 3.5.14 | uds_svc_param_tp | 75 |
| 3.5.15 | uds_svc_param_cdtcs | 76 |
| 3.5.16 | uds_svc_param_roe | 77 |
| 3.5.17 | uds_svc_param_roe_recommended_service_id | 78 |
| 3.5.18 | uds_svc_param_lc | 79 |
| 3.5.19 | uds_svc_param_lc_baudrate_identifier | 80 |
| 3.5.20 | uds_svc_param_di | 82 |
| 3.5.21 | uds_svc_param_rdbpi | 86 |
| 3.5.22 | uds_svc_param_ddd | 87 |

| | | |
|--------|---|-----|
| 3.5.23 | uds_svc_param_rdtci | 88 |
| 3.5.24 | uds_svc_param_rdtci_dtcsvm | 91 |
| 3.5.25 | uds_svc_param_iocbi | 92 |
| 3.5.26 | uds_svc_param_rc | 93 |
| 3.5.27 | uds_svc_param_rc_rid | 94 |
| 3.5.28 | uds_svc_param_atp | 95 |
| 3.5.29 | uds_svc_param_rft_moop | 96 |
| 3.5.30 | uds_svc_authentication_subfunction | 98 |
| 3.5.31 | uds_svc_authentication_return_parameter | 99 |
| 3.6 | PCAN-ISO-TP 3.x Dependencies | 101 |
| 3.6.1 | cantp_bitrate | 101 |
| 3.6.2 | cantp_timestamp | 104 |
| 3.6.3 | cantp_handle | 105 |
| 3.6.4 | cantp_hwtype | 113 |
| 3.6.5 | cantp_isotp_addressing | 115 |
| 3.6.6 | cantp_baudrate | 116 |
| 3.6.7 | cantp_msg | 118 |
| 3.6.8 | cantp_can_msgtype | 121 |
| 3.6.9 | cantp_msgtype | 123 |
| 3.6.10 | cantp_can_info | 124 |
| 3.6.11 | cantp_msgdata | 125 |
| 3.6.12 | cantp_msgdata_can | 126 |
| 3.6.13 | cantp_msgdata_canfd | 128 |
| 3.6.14 | cantp_msgdata_isotp | 129 |
| 3.6.15 | cantp_msgflag | 131 |
| 3.6.16 | cantp_netstatus | 132 |
| 3.6.17 | cantp_netaddrinfo | 134 |
| 3.6.18 | cantp_isotp_msgtype | 136 |
| 3.6.19 | cantp_isotp_format | 137 |
| 3.7 | Methods | 139 |
| 3.7.1 | Initialize_2013 | 142 |
| 3.7.2 | Initialize_2013(cantp_handle, cantp_baudrate) | 142 |
| 3.7.3 | Initialize_2013(cantp_handle, cantp_baudrate, cantp_hwtype, UInt32, UInt16) | 145 |
| 3.7.4 | InitializeFD_2013 | 148 |
| 3.7.5 | Uninitialize_2013 | 151 |
| 3.7.6 | SetValue_2013 | 153 |
| 3.7.7 | SetValue_2013(cantp_handle, uds_parameter, UInt32, UInt32) | 154 |
| 3.7.8 | SetValue_2013(cantp_handle, uds_parameter, String, UInt32) | 156 |
| 3.7.9 | SetValue_2013(cantp_handle, uds_parameter, Byte[], UInt32) | 158 |
| 3.7.10 | SetValue_2013(cantp_handle, uds_parameter, IntPtr, UInt32) | 160 |
| 3.7.11 | AddMapping_2013 | 164 |
| 3.7.12 | RemoveMapping_2013 | 168 |
| 3.7.13 | RemoveMappingByCanId_2013 | 171 |
| 3.7.14 | AddCanIdFilter_2013 | 175 |
| 3.7.15 | RemoveCanIdFilter_2013 | 176 |
| 3.7.16 | GetValue_2013 | 178 |
| 3.7.17 | GetValue_2013(cantp_handle, uds_parameter, String, UInt32) | 179 |
| 3.7.18 | GetValue_2013(cantp_handle, uds_parameter, UInt32, UInt32) | 181 |
| 3.7.19 | GetValue_2013(cantp_handle, uds_parameter, Byte[], UInt32) | 184 |

| | | |
|--------|--|-----|
| 3.7.20 | GetValue_2013(cantp_handle, uds_parameter, IntPtr, UInt32) | 186 |
| 3.7.21 | GetCanBusStatus_2013 | 190 |
| 3.7.22 | GetMapping_2013 | 193 |
| 3.7.23 | GetMappings_2013 | 196 |
| 3.7.24 | GetSessionInformation_2013 | 200 |
| 3.7.25 | StatusIsOk_2013 | 203 |
| 3.7.26 | StatusIsOk_2013(uds_status) | 204 |
| 3.7.27 | StatusIsOk_2013(uds_status, uds_status) | 205 |
| 3.7.28 | StatusIsOk_2013(uds_status, uds_status, bool) | 207 |
| 3.7.29 | GetErrorText_2013 | 209 |
| 3.7.30 | MsgAlloc_2013 | 211 |
| 3.7.31 | MsgFree_2013 | 214 |
| 3.7.32 | MsgCopy_2013 | 216 |
| 3.7.33 | MsgMove_2013 | 218 |
| 3.7.34 | Read_2013 | 219 |
| 3.7.35 | Read_2013(cantp_handle, uds_msg) | 220 |
| 3.7.36 | Read_2013(cantp_handle, uds_msg, uds_msg) | 223 |
| 3.7.37 | Read_2013(cantp_handle, uds_msg, uds_msg, cantp_timestamp) | 226 |
| 3.7.38 | Write_2013 | 229 |
| 3.7.39 | Reset_2013 | 234 |
| 3.7.40 | WaitForSingleMessage_2013 | 235 |
| 3.7.41 | WaitForFunctionalResponses_2013 | 241 |
| 3.7.42 | WaitForService_2013 | 248 |
| 3.7.43 | WaitForServiceFunctional_2013 | 252 |
| 3.7.44 | SvcDiagnosticSessionControl_2013 | 258 |
| 3.7.45 | SvcECUReset_2013 | 262 |
| 3.7.46 | SvcSecurityAccess_2013 | 267 |
| 3.7.47 | SvcSecurityAccess_2013(cantp_handle, uds_msgconfig, uds_msg, Byte) | 267 |
| 3.7.48 | SvcSecurityAccess_2013(cantp_handle, uds_msgconfig, uds_msg, Byte, Byte[], UInt32) | 271 |
| 3.7.49 | SvcCommunicationControl_2013 | 276 |
| 3.7.50 | SvcCommunicationControl_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_cc, Byte) | 276 |
| 3.7.51 | SvcCommunicationControl_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_cc, Byte, UInt16) | 280 |
| 3.7.52 | SvcTesterPresent_2013 | 285 |
| 3.7.53 | SvcTesterPresent_2013(cantp_handle, uds_msgconfig, uds_msg) | 285 |
| 3.7.54 | SvcTesterPresent_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_tp) | 289 |
| 3.7.55 | SvcSecuredDataTransmission_2013 | 293 |
| 3.7.56 | SvcSecuredDataTransmission_2020 | 298 |
| 3.7.57 | SvcControlDTCSetting_2013 | 305 |
| 3.7.58 | SvcControlDTCSetting_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_cdtcs) | 305 |
| 3.7.59 | SvcControlDTCSetting_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_cdtcs, Byte[], UInt32) | 309 |
| 3.7.60 | SvcResponseOnEvent_2013 | 314 |
| 3.7.61 | SvcResponseOnEvent_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_roe, bool, byte) | 314 |
| 3.7.62 | SvcResponseOnEvent_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_roe, bool, byte, byte[], UInt32) | 318 |
| 3.7.63 | SvcResponseOnEvent_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_roe, bool, byte, byte[], UInt32, byte[], UInt32) | 323 |
| 3.7.64 | SvcLinkControl_2013 | 328 |

| | | |
|---------|--|-----|
| 3.7.65 | SvcLinkControl_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_lc, uds_svc_param_lc_baudrate_identifier) | 329 |
| 3.7.66 | SvcLinkControl_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_lc, uds_svc_param_lc_baudrate_identifier, UInt32) | 333 |
| 3.7.67 | SvcReadDataByIdentifier_2013 | 337 |
| 3.7.68 | SvcReadMemoryByAddress_2013 | 342 |
| 3.7.69 | SvcReadScalingDataByIdentifier_2013 | 347 |
| 3.7.70 | SvcReadDataByPeriodicIdentifier_2013 | 351 |
| 3.7.71 | SvcDynamicallyDefinedDataIdentifierDBID_2013 | 356 |
| 3.7.72 | SvcDynamicallyDefinedDataIdentifierDBMA_2013 | 362 |
| 3.7.73 | SvcDynamicallyDefinedDataIdentifierCDDDI_2013 | 368 |
| 3.7.74 | SvcDynamicallyDefinedDataIdentifierClearAllDDDI_2013 | 372 |
| 3.7.75 | SvcWriteDataByIdentifier_2013 | 376 |
| 3.7.76 | SvcWriteMemoryByAddress_2013 | 381 |
| 3.7.77 | SvcClearDiagnosticInformation_2013 | 387 |
| 3.7.78 | SvcClearDiagnosticInformation_2020 | 391 |
| 3.7.79 | SvcReadDTCInformation_2013 | 396 |
| 3.7.80 | SvcReadDTCInformationRDTCSBDTC_2013 | 400 |
| 3.7.81 | SvcReadDTCInformationRDTCSBRN_2013 | 404 |
| 3.7.82 | SvcReadDTCInformationReportExtended_2013 | 408 |
| 3.7.83 | SvcReadDTCInformationReportSeverity_2013 | 413 |
| 3.7.84 | SvcReadDTCInformationRSIODTC_2013 | 418 |
| 3.7.85 | SvcReadDTCInformationNoParam_2013 | 422 |
| 3.7.86 | SvcReadDTCInformationRDTCEDBR_2013 | 426 |
| 3.7.87 | SvcReadDTCInformationRUDMDTCBSM_2013 | 431 |
| 3.7.88 | SvcReadDTCInformationRUDMDTCSSBDTC_2013 | 435 |
| 3.7.89 | SvcReadDTCInformationRUDMDTCEDRBDN_2013 | 440 |
| 3.7.90 | SvcReadDTCInformationRDTCEDI_2020 | 444 |
| 3.7.91 | SvcReadDTCInformationRWWHOBDDTCBMR_2013 | 449 |
| 3.7.92 | SvcReadDTCInformationRWWHOBDDTCWPS_2013 | 453 |
| 3.7.93 | SvcReadDTCInformationRDTCBRGI_2020 | 458 |
| 3.7.94 | SvcInputOutputControlByIdentifier_2013 | 462 |
| 3.7.95 | SvcInputOutputControlByIdentifier_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_di, byte[], UInt32) | 462 |
| 3.7.96 | SvcInputOutputControlByIdentifier_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_di, byte[], UInt32, byte[], UInt32) | 468 |
| 3.7.97 | SvcRoutineControl_2013 | 473 |
| 3.7.98 | SvcRoutineControl_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_rc, uds_svc_param_rc_rid) | 474 |
| 3.7.99 | SvcRoutineControl_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_rc, uds_svc_param_rc_rid, byte[], UInt32) | 478 |
| 3.7.100 | SvcRequestDownload_2013 | 483 |
| 3.7.101 | SvcRequestUpload_2013 | 489 |
| 3.7.102 | SvcTransferData_2013 | 494 |
| 3.7.103 | SvcTransferData_2013(cantp_handle, uds_msgconfig, uds_msg, byte) | 494 |
| 3.7.104 | SvcTransferData_2013(cantp_handle, uds_msgconfig, uds_msg, byte, byte[], UInt32) | 498 |
| 3.7.105 | SvcRequestTransferExit_2013 | 503 |
| 3.7.106 | SvcRequestTransferExit_2013(cantp_handle, uds_msgconfig, uds_msg) | 503 |
| 3.7.107 | SvcRequestTransferExit_2013(cantp_handle, uds_msgconfig, uds_msg, byte[], UInt32) | 507 |
| 3.7.108 | SvcAccessTimingParameter_2013 | 512 |

| | | |
|---------|---|-----|
| 3.7.109 | SvcRequestFileTransfer_2013 | 517 |
| 3.7.110 | SvcRequestFileTransfer_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_rft_moop, UInt16, string) | 517 |
| 3.7.111 | SvcRequestFileTransfer_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_rft_moop, UInt16, string, byte, byte) | 521 |
| 3.7.112 | SvcRequestFileTransfer_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_rft_moop, UInt16, string, byte, byte, byte, byte[], byte[]) | 526 |
| 3.7.113 | SvcRequestFileTransfer_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_rft_moop, UInt16, byte[]) | 531 |
| 3.7.114 | SvcRequestFileTransfer_2013(cantp_handle, uds_msgconfig, out uds_msg, uds_svc_param_rft_moop, UInt16, byte[], byte, byte) | 536 |
| 3.7.115 | SvcRequestFileTransfer_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_rft_moop, UInt16, byte[], byte, byte, byte, byte[], byte[]) | 540 |
| 3.7.116 | SvcAuthenticationDA_2020 | 545 |
| 3.7.117 | SvcAuthenticationVCU_2020 | 549 |
| 3.7.118 | SvcAuthenticationVCU_2020(cantp_handle, uds_msgconfig, uds_msg, Byte, byte[], UInt16, byte[], UInt16) | 550 |
| 3.7.119 | SvcAuthenticationVCU_2020(cantp_handle, uds_msgconfig, uds_msg, Byte, byte[], UInt16) | 555 |
| 3.7.120 | SvcAuthenticationVCB_2020 | 559 |
| 3.7.121 | SvcAuthenticationPOWN_2020 | 565 |
| 3.7.122 | SvcAuthenticationPOWN_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], UInt16, byte[], UInt16) | 565 |
| 3.7.123 | SvcAuthenticationPOWN_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], UInt16) | 570 |
| 3.7.124 | SvcAuthenticationRCFA_2020 | 574 |
| 3.7.125 | SvcAuthenticationVPOWNU_2020 | 579 |
| 3.7.126 | SvcAuthenticationVPOWNU_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], byte[], UInt16, byte[], UInt16, byte[], UInt16) | 579 |
| 3.7.127 | SvcAuthenticationVPOWNU_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], byte[], UInt16, byte[], UInt16) | 585 |
| 3.7.128 | SvcAuthenticationVPOWNU_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], byte[], UInt16) | 591 |
| 3.7.129 | SvcAuthenticationVPOWNB_2020 | 596 |
| 3.7.130 | SvcAuthenticationVPOWNB_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], byte[], UInt16, byte[], UInt16, byte[], UInt16) | 596 |
| 3.7.131 | SvcAuthenticationVPOWNB_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], byte[], UInt16, byte[], UInt16) | 602 |
| 3.7.132 | SvcAuthenticationAC_2020 | 607 |
| 3.7.133 | GetDataServiceId_2013 | 611 |
| 3.7.134 | SetDataServiceId_2013 | 613 |
| 3.7.135 | GetDataNrc_2013 | 614 |
| 3.7.136 | SetDataNrc_2013 | 616 |
| 3.7.137 | GetDataParameter_2013 | 618 |
| 3.7.138 | SetDataParameter_2013 | 619 |
| 3.8 | Functions | 621 |
| 3.8.1 | UDS_Initialize_2013 | 624 |
| 3.8.2 | UDS_InitializeFD_2013 | 626 |
| 3.8.3 | UDS_Uninitialize_2013 | 628 |
| 3.8.4 | UDS_SetValue_2013 | 629 |
| 3.8.5 | UDS_GetValue_2013 | 630 |

| | | |
|--------|--|-----|
| 3.8.6 | UDS_AddMapping_2013 | 631 |
| 3.8.7 | UDS_RemoveMappingByCanId_2013 | 633 |
| 3.8.8 | UDS_RemoveMapping_2013 | 634 |
| 3.8.9 | UDS_AddCanIdFilter_2013 | 635 |
| 3.8.10 | UDS_RemoveCanIdFilter_2013 | 636 |
| 3.8.11 | UDS_GetMapping_2013 | 637 |
| 3.8.12 | UDS_GetMappings_2013 | 638 |
| 3.8.13 | UDS_GetSessionInformation_2013 | 639 |
| 3.8.14 | UDS_StatusIsOk_2013 | 640 |
| 3.8.15 | UDS_GetErrorText_2013 | 641 |
| 3.8.16 | UDS_GetCanBusStatus_2013 | 642 |
| 3.8.17 | UDS_MsgAlloc_2013 | 643 |
| 3.8.18 | UDS_MsgFree_2013 | 645 |
| 3.8.19 | UDS_MsgCopy_2013 | 645 |
| 3.8.20 | UDS_MsgMove_2013 | 646 |
| 3.8.21 | UDS_Read_2013 | 648 |
| 3.8.22 | UDS_Write_2013 | 649 |
| 3.8.23 | UDS_Reset_2013 | 651 |
| 3.8.24 | UDS_WaitForSingleMessage_2013 | 652 |
| 3.8.25 | UDS_WaitForFunctionalResponses_2013 | 654 |
| 3.8.26 | UDS_WaitForService_2013 | 656 |
| 3.8.27 | UDS_WaitForServiceFunctional_2013 | 659 |
| 3.8.28 | UDS_SvcDiagnosticSessionControl_2013 | 661 |
| 3.8.29 | UDS_SvcECUReset_2013 | 663 |
| 3.8.30 | UDS_SvcSecurityAccess_2013 | 664 |
| 3.8.31 | UDS_SvcCommunicationControl_2013 | 666 |
| 3.8.32 | UDS_SvcTesterPresent_2013 | 668 |
| 3.8.33 | UDS_SvcSecuredDataTransmission_2013 | 670 |
| 3.8.34 | UDS_SvcSecuredDataTransmission_2020 | 672 |
| 3.8.35 | UDS_SvcControlDTCSetting_2013 | 675 |
| 3.8.36 | UDS_SvcResponseOnEvent_2013 | 677 |
| 3.8.37 | UDS_SvcLinkControl_2013 | 679 |
| 3.8.38 | UDS_SvcReadDataByIdentifier_2013 | 681 |
| 3.8.39 | UDS_SvcReadMemoryByAddress_2013 | 683 |
| 3.8.40 | UDS_SvcReadScalingDataByIdentifier_2013 | 685 |
| 3.8.41 | UDS_SvcReadDataByPeriodicIdentifier_2013 | 686 |
| 3.8.42 | UDS_SvcDynamicallyDefinedDataIdentifierDBID_2013 | 688 |
| 3.8.43 | UDS_SvcDynamicallyDefinedDataIdentifierDBMA_2013 | 690 |
| 3.8.44 | UDS_SvcDynamicallyDefinedDataIdentifierCDDDI_2013 | 693 |
| 3.8.45 | UDS_SvcDynamicallyDefinedDataIdentifierClearAllDDDI_2013 | 694 |
| 3.8.46 | UDS_SvcWriteDataByIdentifier_2013 | 696 |
| 3.8.47 | UDS_SvcWriteMemoryByAddress_2013 | 698 |
| 3.8.48 | UDS_SvcClearDiagnosticInformation_2013 | 700 |
| 3.8.49 | UDS_SvcClearDiagnosticInformation_2020 | 702 |
| 3.8.50 | UDS_SvcReadDTCInformation_2013 | 704 |
| 3.8.51 | UDS_SvcReadDTCInformationRDTCSBDTC_2013 | 706 |
| 3.8.52 | UDS_SvcReadDTCInformationRDTCSBRN_2013 | 707 |
| 3.8.53 | UDS_SvcReadDTCInformationReportExtended_2013 | 709 |
| 3.8.54 | UDS_SvcReadDTCInformationReportSeverity_2013 | 711 |
| 3.8.55 | UDS_SvcReadDTCInformationRSIODTC_2013 | 713 |
| 3.8.56 | UDS_SvcReadDTCInformationNoParam_2013 | 714 |
| 3.8.57 | UDS_SvcReadDTCInformationRDTCEDBR_2013 | 716 |
| 3.8.58 | UDS_SvcReadDTCInformationRUDMDTCBSM_2013 | 718 |
| 3.8.59 | UDS_SvcReadDTCInformationRUDMDTCSSBDTC_2013 | 720 |
| 3.8.60 | UDS_SvcReadDTCInformationRUDMDTCEDRBDN_2013 | 722 |
| 3.8.61 | UDS_SvcReadDTCInformationRDTCEDI_2020 | 724 |
| 3.8.62 | UDS_SvcReadDTCInformationRWWHOBDDTCBMR_2013 | 725 |

| | | |
|----------|---|------------|
| 3.8.63 | UDS_SvcReadDTCInformationRWWHOBDDTCWPS_2013 | 727 |
| 3.8.64 | UDS_SvcReadDTCInformationRDTCBIRGI_2020 | 729 |
| 3.8.65 | UDS_SvcInputOutputControlByIdentifier_2013 | 731 |
| 3.8.66 | UDS_SvcRoutineControl_2013 | 733 |
| 3.8.67 | UDS_SvcRequestDownload_2013 | 735 |
| 3.8.68 | UDS_SvcRequestUpload_2013 | 737 |
| 3.8.69 | UDS_SvcTransferData_2013 | 739 |
| 3.8.70 | UDS_SvcRequestTransferExit_2013 | 742 |
| 3.8.71 | UDS_SvcAccessTimingParameter_2013 | 744 |
| 3.8.72 | UDS_SvcRequestFileTransfer_2013 | 745 |
| 3.8.73 | UDS_SvcAuthenticationDA_2020 | 748 |
| 3.8.74 | UDS_SvcAuthenticationVCU_2020 | 749 |
| 3.8.75 | UDS_SvcAuthenticationVCB_2020 | 752 |
| 3.8.76 | UDS_SvcAuthenticationPOWN_2020 | 754 |
| 3.8.77 | UDS_SvcAuthenticationRCFA_2020 | 756 |
| 3.8.78 | UDS_SvcAuthenticationVPOWNU_2020 | 758 |
| 3.8.79 | UDS_SvcAuthenticationVPOWNB_2020 | 760 |
| 3.8.80 | UDS_SvcAuthenticationAC_2020 | 763 |
| 3.9 | Definitions | 765 |
| 3.9.1 | Messages Related Definitions | 765 |
| 3.9.2 | PCAN-UDS 2.x Service Parameter Definitions | 765 |
| 4 | Additional Information | 768 |
| 4.1 | PCAN Fundamentals | 768 |
| 4.2 | PCAN-Basic | 769 |
| 4.3 | UDS and ISO-TP Network Addressing Information | 770 |
| 4.3.1 | Usage in a Non-Standardized Context | 772 |
| 4.3.2 | ISO-TP Network Addressing Format | 773 |
| 4.3.3 | PCAN-UDS 2.x Example | 774 |
| 4.4 | Using Events | 777 |
| 5 | License Information | 778 |

1 PCAN-UDS 2.x API Documentation

Welcome to the documentation of PCAN-UDS 2.x API, a PEAK CAN API that implements ISO 14229-1:2013, UDS in CAN, an international standard that allows a diagnostic tester (client) to control diagnostic functions in an on-vehicle Electronic Control Unit (ECU or server).

In the following chapters, you will find all the information needed to take advantage of this API.

- └ Introduction on page 11
- └ DLL API Reference on page 15
- └ Additional Information on page 768

2 Introduction

PCAN-UDS 2.x is a simple programming interface intended to support Windows automotive applications that use PEAK-Hardware to communicate with Electronic Control Units (ECU) connected to the bus systems of a car, for maintenance purpose.

2.1 Understanding PCAN-UDS 2.x

UDS stands for Unified Diagnostic Services and is a communication protocol of the automotive industry. This protocol is described in the norm ISO 14229-1:2013.

The UDS protocol is the result of 3 other standardized diagnostic communication protocols:

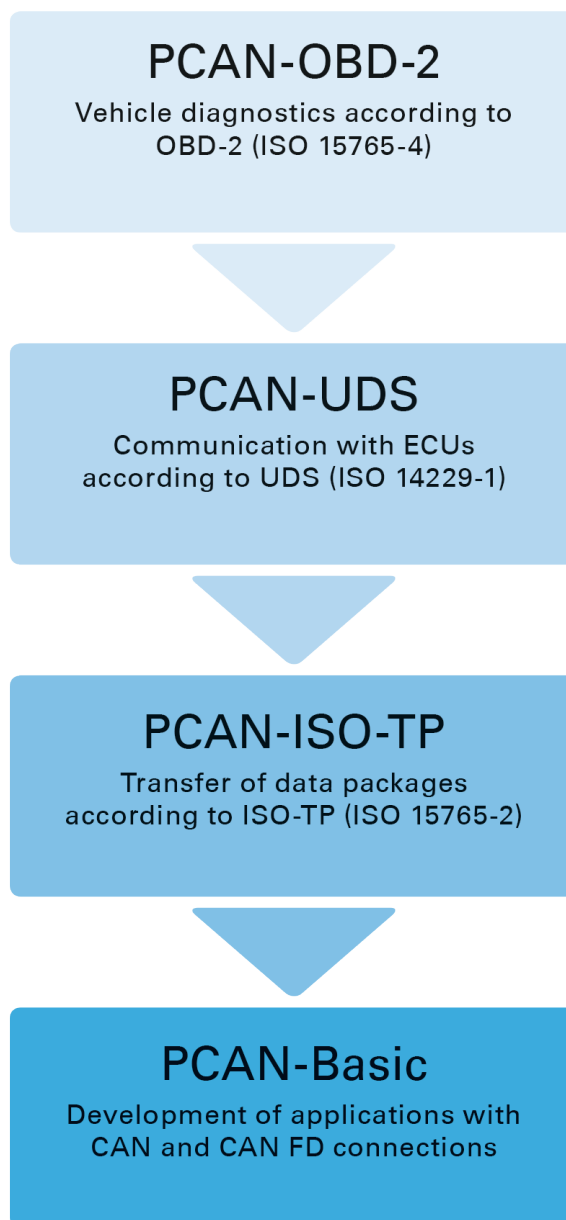
- ISO 14230-3, as known as Keyword 2000 Protocol (KWP2000)
- ISO 14229-1:2013, as known as Diagnostic on CAN
- ISO 15765-2, as known as ISO-TP

The idea of this protocol is to contact all electronic data units installed and interconnected in a car, to provide maintenance, as checking for errors, actualizing of firmware, etc.

UDS is a Client/Server oriented protocol. In a UDS session (diagnostic session), a program application on a computer constitutes the client (within UDS, it is called Tester), the server is the ECU being *tested*, and the diagnostic requests from client to server are called services. The client always starts with a request and this ends with a positive or negative response from the server (ECU).

Since the transport protocol of UDS is done using ISO-TP, an international standard for sending data packets over a CAN bus, the maximum data length that can be transmitted in a single data block is 4 Gigabytes.

PCAN-UDS 2.x API is an implementation of the UDS on CAN standard. The physical communication is carried out by PCAN-Hardware (PCAN-USB, PCAN-PCI, etc.) through the PCAN-ISO-TP 3.x and PCAN-Basic API (free CAN APIs from PEAK-System). Because of this, it is necessary to have also the PCAN-ISO-TP 3.x and PCAN-Basic APIs (PCAN-ISO-TP.dll and PCANBasic.dll) present on the working computer where UDS is intended to be used. PCAN-UDS 2.x, PCAN-ISO-TP 3.x, and PCAN-Basic APIs are free and available for all people that acquire a PCAN-Hardware.



2.2 Using PCAN-UDS 2.x

Since PCAN-UDS 2.x API is built on top of the PCAN-ISO-TP 3.x API and PCAN-Basic APIs, it shares similar functions. It offers the possibility to use several PUDS channels within the same application in an easy way. The communication process is divided in 3 phases: initialization, interaction, and finalization of a PUDS-Channel.

Initialization: To perform UDS on CAN communication using a channel, it is necessary to initialize it first. This is done by making a call to the function `UDS_Initialize_2013` (**class-method:** `Initialize_2013`).

Interaction: After successful initialization, a channel is ready to communicate with the connected CAN bus. Further configuration is not needed. The functions starting with `UDS_Svc` (**class-methods:** starting with `Svc`) can be used to transmit UDS requests and the utility functions starting with `UDS_WaitFor` (**class-methods:** starting with `WaitFor`) are used to retrieve the results of a previous request. The `UDS_Read_2013` and `UDS_Write_2013` (**class-methods:** `Read_2013` and `Write_2013`) are lower-level functions to read and write UDS messages from scratch. If desired, an extra configuration can be made to improve a communication session, like service request timeouts or PCAN-ISO-TP 3.x parameters.

Finalization: When the communication is finished, the function `UDS_Uninitialize_2013` (**class-method:** `Uninitialize_2013`) should be called in order to release the PUDS-Channel and the resources allocated for it. In this way the channel is marked as "Free" and can be used from other applications.

2.3 Features

- Implementation of the UDS protocol (14229-1:2013) for the communication with control units
- Windows DLLs for the development of applications for the platforms Windows® 11 (x64/ARM64), 10 (x86/x64)
- Physical communication via CAN using a CAN interface of the PCAN series
- Uses the PCAN-Basic programming interface to access the CAN hardware in the computer
- Uses the PCAN-ISO-TP 3.x programming interface (ISO 15765-2) for the transfer of data packages up to 4 Gigabytes via the CAN bus

2.4 System Requirements

- Windows 11 (x64/ARM64), Windows 10 (x64)
- PC CAN interface from PEAK-System
- PCAN-Basic API
- PCAN-ISO-TP 3.x API

2.5 Scope of Supply

- Interface DLLs for Windows (x86/x64/ARM64)
- Examples and header files for all common programming languages
- Documentation in PDF format

2.6 Backward Compatibility Notes

Until version 1.x, the PCAN-UDS API implemented the protocol UDS as described in the ISO norm 14229-1, revision 2006. Support for the ISO norm 14229-1, revision 2013, was introduced with version 2.0 of the API PCAN-UDS. New features of the norm 14229-1 caused changes in the API, that are to be considered when porting projects written with older versions of the PCAN-UDS API.

2.6.1 PCAN-ISO-TP Compatibility

The PCAN-UDS 2.x API is now based on PCAN-ISO-TP 3.x API. The legacy PCAN-UDS 1.0 API was based on PCAN-ISO-TP 2.0 API. Porting PCAN-UDS project requires to upgrade PCAN-ISO-TP. Backward compatibility notes for PCAN-ISO-TP 3.x are given in PCAN-ISO-TP user manual.

2.6.2 Binary Compatibility

The new `PCAN-UDS.dll`, version 2.x, is full compatible with applications written using earlier versions of the API. Existing applications don't need to be rebuilt when updating the `PCAN-UDS.dll`.

However, in some cases, a known issue can occur: writing UDS message or calling service function returns an unexpected `PUDS_ERROR_ALREADY_INITIALIZED` error. This new error occurred when PCAN-UDS 2.x detects that a message with the same service identifier is already pending in the reception queue (see `PUDS_STATUS_SERVICE_ALREADY_PENDING` in `uds_status` on page 32): this could lead to a possible shift between similar UDS requests and their responses. To correct this issue, the user must read a response for his previous request before writing the new message or clear the reception queue by calling `UDS_Reset`.

2.6.3 Code Compatibility

Both revisions of the ISO norm 14229-1 mentioned before are visually split into different header files for the supported programming languages, `PCAN-UDS_2006.*` and `PCAN-UDS_2013.*` (for Delphi `PUDS_2006.pas`, and `PUDS_2013.pas`). Additionally, for the C/C++ languages, a header file called the same as in the previous API version, `PCAN-UDS.h`, is now used for backward compatibility, making easier the update of existing C/C++ projects.

All these files are included in the PCAN-UDS package under the folder "Include" (header files with suffix `_2013`) and its subfolder "Backward Compatibility" (other header files).


Depending on the needs of a developer and his project, he can:

a. Start a new UDS:2013 project (recommended)

The header file called `PCAN-UDS_2013.*` (for Delphi, `PUDS_2013.pas`) is to be used. No other UDS header file is needed.

b. Start a new UDS:2006 project


The header file called `PCAN-UDS_2006.*` (for Delphi `PUDS_2006.pas`) is to be used. No other UDS header file is needed. The content of this file represents the content of the file `PCAN-UDS.*` (for Delphi, `PUDS.pas`) from the version 1.x of the PCAN-UDS API.

 **Note:** In C/C++: This header doesn't include the required dependency "`Window.h`". It may be needed to include this reference manually, depending on the configuration of the project.

c. Updating an existing project based on version 1.x of the PCAN-UDS API

▶ C/C++ projects:

1. Copy all 3 header files, `PCAN-UDS.h`, `PCAN-UDS_2006.h`, and `PCAN-UDS_2013.h` into the project folder.

 **Note:** The header file called `PCAN-UDS.h` will be replaced by the new backward-compatible header file of the same name. This header ensures that the old interface, i.e. all function prototypes from the PCAN-UDS 1.x API, and also the new interface introduced with version 2.0, are available for the project.

2. Open and re-compile the project / solution.

▶ .NET and Delphi projects (Support for the revision 2006 only):

1. Copy the header file `PCAN-UDS_2006.*` (for Delphi, `PUDS_2006.pas`) into the project folder.
2. Load the project / solution.
3. Exclude the file `PCAN-UDS.*` (for Delphi, `PUDS.pas`) from the project.
4. Include the file `PCAN-UDS_2006.*` (for Delphi, `PUDS_2006.pas`) to the project.
5. Save and compile the project / solution.

▶ .NET and Delphi projects (Support for both revisions, 2006 and 2013):

1. Copy the header files `PCAN-UDS_2006.*`, and `PCAN-UDS_2013.*` (for Delphi, `PUDS_2006.pas` and `PUDS_2013.pas`) into the project folder.
2. Exclude the file `PCAN-UDS.*` (for Delphi, `PUDS.pas`) from the project.
3. Include the files `PCAN-UDS_2006.*`, `PCAN-UDS_2013.*` (for Delphi, `PUDS_2006.pas` and `PUDS_2013.pas`).
4. Edit the file `PCAN-UDS_2013.*` (for Delphi, `PUDS_2013.pas`), and enable the commented define directive for `PUDS_API_COMPATIBILITY_ISO_2006`, within the description of the header file.
5. Save and compile the project / solution.

2.6.4 Remark

Beginning with version 2.1, the parameters of `UDS_WaitFor*_2013` functions have a new order. They do not keep the order of the previous version.

3 DLL API Reference

This section contains information about the data types (classes, structures, types, defines, enumerations) and API functions which are contained in the PCAN-UDS 2.x API.

3.1 Namespaces

PEAK offers the implementation of some specific programming interfaces as namespaces for the .NET Framework programming environment. The following namespaces are available:

Namespaces

| | Name | Description |
|---|----------------|--|
| ⌋ | Peak | Contains all namespaces that are part of the managed programming environment from PEAK-System |
| ⌋ | Peak.Can | Contains types and classes for using the PCAN API from PEAK-System |
| ⌋ | Peak.Can.Light | Contains types and classes for using the PCAN-Light API from PEAK-System |
| ⌋ | Peak.Can.Basic | Contains types and classes for using the PCAN-Basic API from PEAK-System |
| ⌋ | Peak.Can.Ccp | Contains types and classes for using the CCP API implementation from PEAK-System |
| ⌋ | Peak.Can.Xcp | Contains types and classes for using the XCP API implementation from PEAK-System |
| ⌋ | Peak.Can.IsoTp | Contains types and classes for using the PCAN-ISO-TP 3.x API implementation from PEAK-System |
| ⌋ | Peak.Can.Uds | Contains types and classes for using the PCAN-UDS 2.x API implementation from PEAK-System |
| ⌋ | Peak.Can.ObdII | Contains types and classes for using the PCAN-OBDII API implementation from PEAK-System |
| ⌋ | Peak.Lin | Contains types and classes used to handle with LIN devices from PEAK-System |
| ⌋ | Peak.RP1210A | Contains types and classes used to handle with CAN devices from PEAK-System through the TMC Recommended Practices 1210, version A, as known as RP1210(A) |


3.1.1 Peak.Can.Uds

The `Peak.Can.Uds` namespace contains types and classes to use the PCAN-UDS 2.x API within the .NET Framework programming environment and handle PCAN devices from PEAK-System.






Remarks

Under the Delphi environment, these elements are enclosed in the PUDS-Unit. The functionality of all elements included here is just the same. The difference between this namespace and the Delphi unit consists in the fact that Delphi accesses the Windows API directly (it is not Managed Code).

Classes

| | Class | Description |
|---|--------|--|
|  | UDSApi | Defines a class which represents the PCAN-UDS 2.x API. |

Structures
































| | Structure | Description |
|---|-----------------|--|
|  | uds_msg | Represents the content of a UDS message. |
|  | uds_sessioninfo | Represents the diagnostic session information of a server (ECU). |
|  | uds_netaddrinfo | Represents the network addressing information of a UDS message. |
|  | uds_msgconfig | Represents a PUDS message (uds_msg) configuration. |
|  | uds_mapping | Represents a mapping between a network address information and a CAN identifier. |



uds_msgaccess

This structure provides accessors to the corresponding data in the uds_msg structure.


Enumerations

| Name | Description |
|---|---|
|  uds_errstatus | Represents PUDS error codes (used in uds_status) |
|  uds_status | Represents a PUDS status or error code. |
|  uds_parameter | Represents a PUDS parameter to be read or set. |
|  uds_service | Represents a service identifier defined in ISO 14229-1. |
|  uds_address | Represents a standardized ISO-15765-4 address. |
|  uds_can_id | Represents a standardized ISO-15765-4 CAN identifier. |
|  uds_msgprotocol | Represents a standardized and supported network communication protocol. |
|  uds_status_offset | Defines constants used by the uds_status enumeration. |
|  uds_msgtype | Represents types and flags for a uds_msg. |
|  uds_nrc | Represents a UDS negative response code (NRC). |
|  uds_svc_param_dsc | Represents the subfunction parameter for the UDS service DiagnosticSessionControl. |
|  uds_svc_param_er | Represents the subfunction parameter for the UDS service ECUReset. |
|  uds_svc_param_cc | Represents the subfunction parameter for the UDS service CommunicationControl. |
|  uds_svc_param_tp | Represents the subfunction parameter for the UDS service TesterPresent. |
|  uds_svc_param_cdtcs | Represents the subfunction parameter for the UDS service ControlDTCSetting. |
|  uds_svc_param_roe | Represents the subfunction parameter for the UDS service ResponseOnEvent. |
|  uds_svc_param_roe_recommended_service_id | Represents the recommended service to respond to for the UDS service ResponseOnEvent. |
|  uds_svc_param_lc | Represents the subfunction parameter for the UDS service LinkControl. |
|  uds_svc_param_lc_baudrate_identifier | Represents the standard baud rate identifier for the UDS service LinkControl. |
|  uds_svc_param_di | Represents the data identifier parameter for the UDS services like ReadDataByIdentifier. |
|  uds_svc_param_rdbpi | Represents the subfunction parameter for the UDS service ReadDataByPeriodicIdentifier. |
|  uds_svc_param_ddd | Represents the subfunction parameter for the UDS service DynamicallyDefineDataIdentifier. |
|  uds_svc_param_rdtci | Represents the subfunction parameter for the UDS service ReadDTCInformation. |
|  uds_svc_param_rdtci_dtcsvm | Represents the DTC severity mask for the UDS service ReadDTCInformation. |
|  uds_svc_param_iocbi | Represents the subfunction parameter for the UDS service InputOutputControlByIdentifier. |
|  uds_svc_param_rc | Represents the subfunction parameter for the UDS service RoutineControl. |
|  uds_svc_param_rc_rid | Represents the routine identifier for the UDS service RoutineControl. |
|  uds_svc_param_atp | Represents the subfunction parameter for the UDS service AccessTimingParameter. |
|  uds_svc_param_rft_moop | Represents the mode of operation parameter for the UDS service RequestFileTransfer. |
|  uds_svc_authentication_subfunction | Represents the subfunction parameter for UDS service Authentication. |
|  uds_svc_authentication_return_parameter | Represents the return parameter for UDS service Authentication. |

3.2 Units

PEAK offers the implementation of some specific programming interfaces as Units for the Delphi's programming environment. The following units are available to be used:

Namespaces

| | Alias | Description |
|---|------------|--|
|  | PUDS Unit. | Delphi unit for using the PCAN-UDS 2.x API from PEAK-System. |


3.2.1 PUDS Unit

The PUDS-Unit contains types and classes to use the PCAN-UDS 2.x API within Delphi's programming environment and handle PCAN devices from PEAK-System.







Remarks

For the .NET Framework, these elements are enclosed in the `Peak.Can.Uds` namespace. The functionality of all elements included here is just the same. The difference between this Unit and the .NET namespace consists in the fact that Delphi accesses the Windows API directly (it is not Managed Code).














Classes



















| | Class | Description |
|--|---------|--|
|  | TUdsApi | Defines a class which represents the PCAN-UDS 2.x API. |

Structures

| | Structure | Description |
|---|-----------------|---|
|  | uds_msg | Represents the content of a UDS message. |
|  | uds_sessioninfo | Represents the diagnostic session information of a server (ECU). |
|  | uds_netaddrinfo | Represents the network addressing information of a UDS message. |
|  | uds_msgconfig | Represents a PUDS message (uds_msg) configuration. |
|  | uds_mapping | Represents a mapping between a network address information and a CAN identifier. |
|  | uds_msgaccess | This structure provides accessors to the corresponding data in the uds_msg structure. |

Enumerations



| | Name | Description |
|---|-------------------|--|
|  | uds_errstatus | Represents PUDS error codes (used in uds_status) |
|  | uds_status | Represents a PUDS status or error code. |
|  | uds_parameter | Represents a PUDS parameter to be read or set. |
|  | uds_service | Represents a service identifier defined in ISO 14229-1. |
|  | uds_address | Represents a standardized ISO-15765-4 address. |
|  | uds_can_id | Represents a standardized ISO-15765-4 CAN identifier. |
|  | uds_msgprotocol | Represents a standardized and supported network communication protocol. |
|  | uds_status_offset | Defines constants used by the uds_status enumeration. |
|  | uds_msgtype | Represents types and flags for a uds_msg. |
|  | uds_ | Represents a UDS negative response code (NRC). |
|  | uds_svc_param_dsc | Represents the subfunction parameter for the UDS service DiagnosticSessionControl. |
|  | uds_svc_param_er | Represents the subfunction parameter for the UDS service ECUReset. |
|  | uds_svc_param_cc | Represents the subfunction parameter for the UDS service CommunicationControl. |

| | Name | Description |
|---|--|---|
|  | uds_svc_param_tp | Represents the subfunction parameter for the UDS service TesterPresent. |
|  | uds_svc_param_cdtcs | Represents the subfunction parameter for the UDS service ControlDTCSetting. |
|  | uds_svc_param_roe | Represents the subfunction parameter for the UDS service ResponseOnEvent. |
|  | uds_svc_param_roe_recommended_service_id | Represents the recommended service to respond to for the UDS service ResponseOnEvent. |
|  | uds_svc_param_lc | Represents the subfunction parameter for the UDS service LinkControl. |
|  | uds_svc_param_lc_baudrate_identifier | Represents the standard baud rate identifier for the UDS service LinkControl. |
|  | uds_svc_param_di | Represents the data identifier parameter for the UDS services like ReadDataByIdentifier. |
|  | uds_svc_param_rdbpi | Represents the subfunction parameter for the UDS service ReadDataByPeriodicIdentifier. |
|  | uds_svc_param_ddd | Represents the subfunction parameter for the UDS service DynamicallyDefineDataIdentifier. |
|  | uds_svc_param_rdtci | Represents the subfunction parameter for the UDS service ReadDTCInformation. |
|  | uds_svc_param_rdtci_dtcsvm | Represents the DTC severity mask for the UDS service ReadDTCInformation. |
|  | uds_svc_param_iocbi | Represents the subfunction parameter for the UDS service InputOutputControlByIdentifier. |
|  | uds_svc_param_rc | Represents the subfunction parameter for the UDS service RoutineControl. |
|  | uds_svc_param_rc_rid | Represents the routine identifier for the UDS service RoutineControl. |
|  | uds_svc_param_atp | Represents the subfunction parameter for the UDS service AccessTimingParameter. |
|  | uds_svc_param_rft_moop | Represents the mode of operation parameter for the UDS service RequestFileTransfer. |
|  | uds_svc_authentication_subfunction | Represents the subfunction parameter for UDS service Authentication. |
|  | uds_svc_authentication_return_parameter | Represents the return parameter for UDS service Authentication. |

3.3 classes

The following classes are offered to make use of the PCAN-UDS 2.x API in a managed or unmanaged way.

Classes

| | Class | Description |
|---|---------|--|
|  | UDSApi | Defines a class to use the PCAN-UDS 2.x API within the Microsoft's .NET Framework programming environment. |
|  | TUdsApi | Defines a class to use the PCAN-UDS 2.x API within the Delphi programming environment. |

3.3.1 UDSApi

Defines a class that represents the PCAN-UDS 2.x API to be used within the Microsoft's .NET Framework.

Syntax

C#

```
public static class UDSApi
```

C++ / CLR

```
public ref class UDSApi abstract sealed
```

Visual Basic


```
Public NotInheritable Class UDSApi
```

Remarks

The `UDSApi` class collects and implements the PCAN-UDS 2.x API methods. Each method is called just like the API function with the exception that the prefix `UDS_` is not used. The structure and functionality of the methods and API functions are the same.

Within the .NET Framework from Microsoft, the `UDSApi` class is a static, not inheritable, class. It must directly be used, without any instance of it, e.g.:

```
uds_status res;
// Static use, without any instance
//
res = UDSApi.Initialize_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    cantp_baudrate.PCANTP_BAUDRATE_500K);
```

 **Note:** This class under Delphi is called `TUdsApi`.

See also: Methods on page 139, Definitions on page 765.

3.3.2 TUdsApi

Defines a class which represents the PCAN-UDS 2.x API to be used within the Delphi programming environment.

Syntax

Pascal OO

```
TUDSApi = class
```

Remarks

`TUDSApi` is a class containing only class-methods and constant members, allowing their use without the creation of any object, just like a static class of other programming languages. It collects and implements the PCAN-UDS 2.x API functions. Each method is called just like the API function with the exception that the prefix `UDS` is not used. The structure and functionality of the methods and API functions are the same.









Note: This class under .NET framework is called `UDSApi`.

See also: Methods on page 139, Definitions on page 765.

3.4 Structures

The PCAN-UDS 2.x API defines the following structures:

| | Name | Description |
|---|-----------------|---|
|  | uds_msg | Represents the content of a UDS message. |
|  | uds_sessioninfo | Represents the diagnostic session information of a server (ECU). |
|  | uds_netaddrinfo | Represents the network addressing information of a UDS message. |
|  | uds_msgconfig | Represents a PUDS message (uds_msg) configuration. |
|  | uds_mapping | Represents a mapping between a network address information and a CAN identifier. |
|  | uds_msgaccess | This structure provides accessors to the corresponding data in the uds_msg structure. |

3.4.1 uds_msg

Represents the content of a UDS message.

Syntax

C/C++

```
typedef struct _uds_msg {
    uds_msgtype type;
    uds_msgaccess links;
    cantp_msg msg;
} uds_msg;
```

Pascal OO

```
uds_msg = record
    typem: uds_msgtype;
    links: uds_msgaccess;
    msg: cantp_msg;
End;
```

C#

```
[StructLayout(LayoutKind.Sequential, Pack = 8)]
public struct uds_msg
{
    [MarshalAs(UnmanagedType.U4)]
    public uds_msgtype type;
    public uds_msgaccess links;
    public cantp_msg msg;
}
```

C++ / CLR

```
[StructLayout(LayoutKind::Sequential, Pack = 8)]
public value struct uds_msg
{
    [MarshalAs(UnmanagedType::U4)]
    uds_msgtype type;
    uds_msgaccess links;
    cantp_msg msg;
};
```

Visual Basic

```
<StructLayout(LayoutKind.Sequential, Pack:=8)>
Public Structure uds_msg
    <MarshalAs(UnmanagedType.U4)>
    Public type As uds_msgtype
    Public links As uds_msgaccess
    Public msg As cantp_msg
End Structure
```

Fields

| Name | Description |
|-------|--|
| type | Type and flags of the message (see uds_msgtype on page 63). |
| links | Quick accessors to the message data (see uds_msgaccess on page 28). |
| msg | The PCANTP message encapsulating the UDS data (see cantp_msg on page 118). |

Remarks

The `uds_msg` structure is automatically initialized and allocated by the PCAN-UDS 2.x API using:

- `UDS_MsgAlloc_2013` function or `MsgAlloc_2013` method
- PUDS services functions (suffixed `UDS_Svc`) or methods (suffixed `Svc`)
- `UDS_Read_2013` function or `Read_2013` method
- PUDS `UDS_WaitFor` functions or `WaitFor` methods

Once processed, the `uds_msg` structure should be released using `UDS_MsgFree_2013` function or `MsgFree_2013` method.

Some message related values are predefined in the API: see Messages Related Definitions on page 765.

3.4.2 uds_sessioninfo

Represents the diagnostic session information of a server (ECU).

Syntax

C/C++

```
typedef struct _uds_sessioninfo
{
    uds_netaddrinfo nai;
    cantp_can_msgtype can_msg_type;
    uint8_t session_type;
    uint16_t timeout_p2can_server_max;
    uint16_t timeout_enhanced_p2can_server_max;
    uint16_t s3_client_ms;
} uds_sessioninfo;
```

Pascal OO

```
uds_sessioninfo = record
    nai: uds_netaddrinfo;
    can_msg_type: cantp_can_msgtype;
    session_type: Byte;
    timeout_p2can_server_max: UInt16;
```

```

    timeout_enhanced_p2can_server_max: UInt16;
    s3_client_ms: UInt16;
End;

```

C#

```

[StructLayout(LayoutKind.Sequential, Pack = 8)]
public struct uds_sessioninfo
{
    public uds_netaddrinfo nai;
    [MarshalAs(UnmanagedType.U4)]
    public cantp_can_msgtype can_msg_type;
    public Byte session_type;
    public UInt16 timeout_p2can_server_max;
    public UInt16 timeout_enhanced_p2can_server_max;
    public UInt16 s3_client_ms;
}

```

C++ / CLR

```

[StructLayout(LayoutKind::Sequential, Pack = 8)]
public value struct uds_sessioninfo
{
    uds_netaddrinfo nai;
    [MarshalAs(UnmanagedType::U4)]
    cantp_can_msgtype can_msg_type;
    Byte session_type;
    UInt16 timeout_p2can_server_max;
    UInt16 timeout_enhanced_p2can_server_max;
    UInt16 s3_client_ms;
};

```

Visual Basic

```





<StructLayout(LayoutKind.Sequential, Pack:=8)>
Public Structure uds_sessioninfo
    Public nai As uds_netaddrinfo
    <MarshalAs(UnmanagedType.U4)>
    Public can_msg_type As cantp_can_msgtype
    Public session_type As Byte
    Public timeout_p2can_server_max As UInt16
    Public timeout_enhanced_p2can_server_max As UInt16
    Public s3_client_ms As UInt16
End Structure

```

Fields

| Name | Description |
|-----------------------------------|---|
| nai | Network Addressing Information (see uds_netaddrinfo on page 24). |
| can_msg_type | Type and flags of the CAN/CAN FD frame (see cantp_can_msgtype on page 121). |
| session_type | Currently activated diagnostic session type (see uds_svc_param_dsc on page 71). |
| timeout_p2can_server_max | Default P2 CAN server max timing for the activated session (i.e. maximum time allowed for the ECU to transmit a response indication). |
| timeout_enhanced_p2can_server_max | Enhanced P2 CAN server max timing for the activated session (i.e. maximum time allowed for the ECU to transmit a response indication after having sent a negative response code stating that more time is required: PUDS_NRC_EXTENDED_TIMING, 0x78). Warning: according to the ISO 14229-2, the resolution of this parameter is 10ms. |
| s3_client_ms | Time between two TesterPresents (using a null s3 timing will disable automatic TesterPresent). |

Predefined Parameters Values

| | Type | Name | Value | Description |
|---|-------|--|-------|---|
|  | Int32 | PUDS_P2CAN_SERVER_MAX_DEFAULT | 50 | Default value in milliseconds for the server performance requirement (see ISO_14229-2_2013 §7.2 table 4). |
|  | Int32 | PUDS_P2CAN_ENHANCED_SERVER_MAX_DEFAULT | 5000 | Default value in milliseconds for the enhanced server performance requirement (see ISO_14229-2_2013 §7.2 table 4). Warning: this parameter is in milliseconds, but the "timeout_enhanced_p2can_server_max" field in uds_sessioninfo has a resolution of 10ms. |
|  | Int32 | PUDS_S3_CLIENT_TIMEOUT_RECOMMENDED | 2000 | Default value in milliseconds for the S3 client performance requirement (see ISO_14229-2_2013 §7.2 table 4). |
|  | Int32 | PUDS_P3CAN_DEFAULT | 50 | Default value in milliseconds for the enhanced server performance requirement (see ISO_14229-2_2013 §7.2 table 4). |

See also: [uds_netaddrinfo](#) on page 24, [UDS_SvcDiagnosticSessionControl_2013](#) on page 661

Class-method: [SvcDiagnosticSessionControl_2013](#) on page 258.

3.4.3 uds_netaddrinfo

Represents the network addressing information of a UDS message.

Syntax

C/C++

```
typedef struct _uds_netaddrinfo {
    uds_msgprotocol protocol;
    cantp_isotp_addressing target_type;
    uint16_t source_addr;
    uint16_t target_addr;
    uint8_t extension_addr;
} uds_netaddrinfo;
```

Pascal OO

```
uds_netaddrinfo = record
    protocol: uds_msgprotocol;
    target_type: cantp_isotp_addressing;
    source_addr: UInt16;
    target_addr: UInt16;
    extension_addr: Byte;
End;
```

C#

```
[StructLayout(LayoutKind.Sequential, Pack = 8)]
public struct uds_netaddrinfo
{
    [MarshalAs(UnmanagedType.U4)]
    public uds_msgprotocol protocol;
    [MarshalAs(UnmanagedType.U4)]
    public cantp_isotp_addressing target_type;
    public UInt16 source_addr;
    public UInt16 target_addr;
    public Byte extension_addr;
}
```


C++ / CLR

```
[StructLayout(LayoutKind::Sequential, Pack = 8)]
public value struct uds_netaddrinfo
{
    [MarshalAs(UnmanagedType::U4)]
    uds_msgprotocol protocol;
    [MarshalAs(UnmanagedType::U4)]
    cantp_isotp_addressing target_type;
    UInt16 source_addr;
    UInt16 target_addr;
    Byte extension_addr;
};
```

Visual Basic

```
<StructLayout(LayoutKind.Sequential, Pack:=8)>
Public Structure uds_netaddrinfo
    <MarshalAs(UnmanagedType.U4)>
    Public protocol As uds_msgprotocol
    <MarshalAs(UnmanagedType.U4)>
    Public target_type As cantp_isotp_addressing
    Public source_addr As UInt16
    Public target_addr As UInt16
    Public extension_addr As Byte
End Structure
```

Fields

| Name | Description |
|----------------|--|
| protocol | Represents the protocol being used for communication (see uds_msgprotocol on page 61). |
| target_type | Represents the target address type (see cantp_isotp_addressing 115). |
| source_addr | Represents the source address (see uds_address on page 56). |
| target_addr | Represents the target address (see uds_address on page 56). |
| extension_addr | Represents the extension address (see uds_address on page 56). |

See also: [uds_address](#) on page 56, [cantp_isotp_addressing](#) on page 115, [uds_msgprotocol](#) on page 61.

3.4.4 uds_mapping

Represents a mapping between a network address information and a CAN identifier.

Syntax

C/C++

```
typedef struct _uds_mapping {
    uintptr_t uid;
    uint32_t can_id;
    uint32_t can_id_flow_ctrl;
    cantp_can_msgtype can_msgtype;
    uint8_t can_tx_dlc;
    uds_netaddrinfo nai;
} uds_mapping;
```

Pascal OO

```
uds_mapping = record
    uid: Pointer;
    can_id: UInt32;
```

```

can_id_flow_ctrl: UInt32;
can_msgtype: cantp_can_msgtype;
can_tx_dlc: Byte;
nai: uds_netaddrinfo;
End;

```

C#

```

[StructLayout(LayoutKind.Sequential, Pack = 8)]
public struct uds_mapping
{
    public UIntPtr uid;
    public UInt32 can_id;
    public UInt32 can_id_flow_ctrl;
    [MarshalAs(UnmanagedType.U4)]
    public cantp_can_msgtype can_msgtype;
    public Byte can_tx_dlc;
    public uds_netaddrinfo nai;
}

```

C++ / CLR

```

[StructLayout(LayoutKind::Sequential, Pack = 8)]
public value struct uds_mapping
{
    UIntPtr uid;
    UInt32 can_id;
    UInt32 can_id_flow_ctrl;
    [MarshalAs(UnmanagedType::U4)]
    cantp_can_msgtype can_msgtype;
    Byte can_tx_dlc;
    uds_netaddrinfo nai;
};

```

Visual Basic

```

<StructLayout(LayoutKind.Sequential, Pack:=8)>
Public Structure uds_mapping
    Public uid As UIntPtr
    Public can_id As UInt32
    Public can_id_flow_ctrl As UInt32
    <MarshalAs(UnmanagedType.U4)>
    Public can_msgtype As cantp_can_msgtype
    Public can_tx_dlc As Byte
    Public nai As uds_netaddrinfo
End Structure

```

Fields

| Name | Description |
|------------------|---|
| uid | Read only, mapping's unique identifier. |
| can_id | CAN identifier mapped to the network address information (see predefined uds_can_id values on page 58). |
| can_id_flow_ctrl | CAN identifier used for the flow control frame, formerly response CAN identifier (see predefined uds_can_id values on page 58). |
| can_msgtype | CAN frame message type, only PCANTP_CAN_MSGTYPE_STANDARD or PCANTP_CAN_MSGTYPE_EXTENDED is mandatory (see cantp_can_msgtype on page 121). |
| can_tx_dlc | Default CAN DLC value to use with segmented messages. Value can be 8 or more if CAN FD communication is supported. If non-zero, this value will supersede parameter PUDS_PARAMETER_CAN_TX_DL for communications involving this mapping. |
| nai | Network Addressing Information (see uds_netaddrinfo on page 24). |

Remark

By default, some mappings are initialized in the PCAN-UDS 2.x API. See UDS and ISO-TP Network Addressing Information on page 770.

See also: [UDS_AddMapping_2013](#) on page 631, [UDS_RemoveMapping_2013](#) on page 634, [AddMapping_2013](#) on page 164, [RemoveMapping_2013](#) on page 168.

3.4.5 uds_msgconfig

Represents a PUDS message ([uds_msg](#)) configuration.

Syntax

C/C++

```
typedef struct _uds_msgconfig {
    uds_msgtype type;
    uds_netaddrinfo nai;
    uint32_t can_id;
    cantp_can_msgtype can_msgtype;
    uint8_t can_tx_dlc;
} uds_msgconfig;
```

Pascal OO

```
uds_msgconfig = record
    typem: uds_msgtype;
    nai: uds_netaddrinfo;
    can_id: UInt32;
    can_msgtype: cantp_can_msgtype;
    can_tx_dlc: Byte;
End;
```

C#

```
[StructLayout(LayoutKind.Sequential, Pack = 8)]
public struct uds_msgconfig
{
    [MarshalAs(UnmanagedType.U4)]
    public uds_msgtype type;
    public uds_netaddrinfo nai;
    public UInt32 can_id;
    [MarshalAs(UnmanagedType.U4)]
    public cantp_can_msgtype can_msgtype;
    public Byte can_tx_dlc;
}
```

C++ / CLR

```
[StructLayout(LayoutKind::Sequential, Pack = 8)]
public value struct uds_msgconfig
{
    [MarshalAs(UnmanagedType::U4)]
    uds_msgtype type;
    uds_netaddrinfo nai;
    UInt32 can_id;
    [MarshalAs(UnmanagedType::U4)]
    cantp_can_msgtype can_msgtype;
    Byte can_tx_dlc;
};
```

Visual Basic

```
<StructLayout(LayoutKind.Sequential, Pack:=8)>
Public Structure uds_msgconfig
    <MarshalAs(UnmanagedType.U4)>
    Public type As uds_msgtype
    Public nai As uds_netaddrinfo
    Public can_id As UInt32
    <MarshalAs(UnmanagedType.U4)>
    Public can_msgtype As cantp_can_msgtype
    Public can_tx_dlc As Byte
End Structure
```

Fields

| Name | Description |
|-------------|---|
| type | Message specific flags (see uds_msgtype on page 63). |
| nai | Message Network Addressing Information (see uds_netaddrinfo on page 24). |
| can_id | Optional, CAN identifier (see predefined uds_can_id values on page 58). |
| can_msgtype | Optional flags for the CAN layer: 29 bits CAN-ID, FD, BRS etc. (see cantp_can_msgtype on page 121). |
| can_tx_dlc | If non-zero, Data Length Code of the frame (see Remarks below). |

Remarks

Specifying a non-zero dlc value in this structure will set message's `can_info.dlc` field with this value. That means when the message will be written the value `can_tx_dlc` of the corresponding mapping and the value of the parameter `PUDS_PARAMETER_CAN_TX_DL` will be overridden (see `uds_mapping` on page 25, `uds_parameter` on page 41, `cantp_can_info` on page 124).

See also: `UDS_MsgAlloc_2013` on page 643, `MsgAlloc_2013` on page 211, `UDS_Scv` functions, `Svc` methods.

3.4.6 uds_msgaccess

This structure provides accessors to the corresponding data in the `uds_msg` structure.

Syntax

C/C++

```
typedef struct _uds_msgaccess {
    uint8_t* service_id;
    uint8_t* param;
    uint8_t* nrc;
} uds_msgaccess;
```

Pascal OO

```
uds_msgaccess = record
    service_id: ^Byte;
    param: ^Byte;
    nrc: ^Byte;
End;
```

C#

```
[StructLayout(LayoutKind.Sequential, Pack = 8)]
public struct uds_msgaccess
{
    public IntPtr service_id;
```

```
public IntPtr param;  
public IntPtr nrc;  
}
```

C++ / CLR

```
[StructLayout(LayoutKind::Sequential, Pack = 8)]  
public value struct uds_msgaccess  
{  
    Byte *service_id;  
    Byte *param;  
    Byte *nrc;  
};
```

Visual Basic

```
<StructLayout(LayoutKind.Sequential, Pack:=8)>  
Public Structure uds_msgaccess  
    Public service_id As IntPtr  
    Public param As IntPtr  
    Public nrc As IntPtr  
End Structure
```








Fields

| Name | Description |
|------------|--|
| service_id | Pointer to the UDS Service Identifier in message's data (see <code>uds_service</code> on page 53). |
| param | Pointer to the first parameter in message's data. |
| nrc | Pointer to the Negative Response Code in message's data, NULL on positive response (see <code>uds_</code> on page 63). |

See also: `uds_msg` on page 21.

3.5 Types

The PCAN-UDS 2.x API defines the following types:

| Name | Description |
|---|---|
|  uds_errstatus | Represents PUDS error codes (used in uds_status) |
|  uds_status | Represents a PUDS status or error code. |
|  uds_parameter | Represents a PUDS parameter to be read or set. |
|  uds_service | Represents UDS Service Identifiers defined in ISO 14229-1. |
|  uds_address | Represents a standardized ISO-15765-4 address. |
|  uds_can_id | Represents a standardized ISO-15765-4 CAN identifier. |
|  uds_status_offset | Defines constants used by the uds_status enumeration. |
|  uds_msgprotocol | Represents a standardized and supported network communication protocol. |
|  uds_msgtype | Represents types and flags for a uds_msg. |
|  uds_nrc | Represents a UDS negative response code (NRC). |
|  uds_svc_param_dsc | Represents the subfunction parameter for the UDS service DiagnosticSessionControl. |
|  uds_svc_param_er | Represents the subfunction parameter for the UDS service ECUReset. |
|  uds_svc_param_cc | Represents the subfunction parameter for the UDS service CommunicationControl. |
|  uds_svc_param_tp | Represents the subfunction parameter for the UDS service TesterPresent. |
|  uds_svc_param_cdtcs | Represents the subfunction parameter for the UDS service ControlDTCSetting. |
|  uds_svc_param_roe | Represents the subfunction parameter for the UDS service ResponseOnEvent. |
|  uds_svc_param_roe_recommended_service_id | Represents the recommended service to respond to for the UDS service ResponseOnEvent. |
|  uds_svc_param_lc | Represents the subfunction parameter for the UDS service LinkControl. |
|  uds_svc_param_lc_baudrate_identifier | Represents the standard baud rate identifier for the UDS service LinkControl. |
|  uds_svc_param_di | Represents the data identifier parameter for the UDS services like ReadDataByIdentifier. |
|  uds_svc_param_rdbpi | Represents the subfunction parameter for the UDS service ReadDataByPeriodicIdentifier. |
|  uds_svc_param_ddd | Represents the subfunction parameter for the UDS service DynamicallyDefineDataIdentifier. |
|  uds_svc_param_rdtci | Represents the subfunction parameter for the UDS service ReadDTCInformation. |
|  uds_svc_param_rdtci_dtcsvm | Represents the DTC severity mask for the UDS service ReadDTCInformation. |
|  uds_svc_param_iocbi | Represents the subfunction parameter for the UDS service InputOutputControlByIdentifier. |
|  uds_svc_param_rc | Represents the subfunction parameter for the UDS service RoutineControl. |
|  uds_svc_param_rc_rid | Represents the routine identifier for the UDS service RoutineControl. |
|  uds_svc_param_atp | Represents the subfunction parameter for the UDS service AccessTimingParameter. It defines the access type. |
|  uds_svc_param_rft_moop | Represents the mode of operation parameter for the UDS service RequestFileTransfer. |
|  uds_svc_authentication_subfunction | Represents the subfunction parameter for UDS service Authentication. |
|  uds_svc_authentication_return_parameter | Represents the return parameter for UDS service Authentication. |

3.5.1 uds_errstatus

Represents a UDS error codes used in uds_status (see uds_status on page 32).

Syntax

C/C++

```
typedef enum _uds_errstatus {
    PUDS_ERRSTATUS_SERVICE_NO_MESSAGE = 1,
    PUDS_ERRSTATUS_SERVICE_TIMEOUT_CONFIRMATION,
    PUDS_ERRSTATUS_SERVICE_TIMEOUT_RESPONSE,
    PUDS_ERRSTATUS_RESET,
    PUDS_ERRSTATUS_ERROR_WAIT_FOR_P3_TIMING,
    PUDS_ERRSTATUS_SERVICE_ALREADY_PENDING,
    PUDS_ERRSTATUS_SERVICE_TX_ERROR,
    PUDS_ERRSTATUS_SERVICE_RX_ERROR,
    PUDS_ERRSTATUS_SERVICE_RX_OVERFLOW,
    PUDS_ERRSTATUS_MESSAGE_BUFFER_ALREADY_USED
}uds_errstatus;
```

Pascal OO

```
uds_errstatus = (
    PUDS_ERRSTATUS_SERVICE_NO_MESSAGE = 1,
    PUDS_ERRSTATUS_SERVICE_TIMEOUT_CONFIRMATION,
    PUDS_ERRSTATUS_SERVICE_TIMEOUT_RESPONSE,
    PUDS_ERRSTATUS_RESET,
    PUDS_ERRSTATUS_ERROR_WAIT_FOR_P3_TIMING,
    PUDS_ERRSTATUS_SERVICE_ALREADY_PENDING,
    PUDS_ERRSTATUS_SERVICE_TX_ERROR,
    PUDS_ERRSTATUS_SERVICE_RX_ERROR,
    PUDS_ERRSTATUS_SERVICE_RX_OVERFLOW,
);
```

C#

```
public enum uds_errstatus : byte
{
    PUDS_ERRSTATUS_SERVICE_NO_MESSAGE = 1,
    PUDS_ERRSTATUS_SERVICE_TIMEOUT_CONFIRMATION,
    PUDS_ERRSTATUS_SERVICE_TIMEOUT_RESPONSE,
    PUDS_ERRSTATUS_RESET,
    PUDS_ERRSTATUS_ERROR_WAIT_FOR_P3_TIMING,
    PUDS_ERRSTATUS_SERVICE_ALREADY_PENDING,
    PUDS_ERRSTATUS_SERVICE_TX_ERROR,
    PUDS_ERRSTATUS_SERVICE_RX_ERROR,
    PUDS_ERRSTATUS_SERVICE_RX_OVERFLOW,
    PUDS_ERRSTATUS_MESSAGE_BUFFER_ALREADY_USED
}
```

C++ / CLR

```
public enum uds_errstatus : Byte
{
    PUDS_ERRSTATUS_SERVICE_NO_MESSAGE = 1,
    PUDS_ERRSTATUS_SERVICE_TIMEOUT_CONFIRMATION,
    PUDS_ERRSTATUS_SERVICE_TIMEOUT_RESPONSE,
    PUDS_ERRSTATUS_RESET,
    PUDS_ERRSTATUS_ERROR_WAIT_FOR_P3_TIMING,
    PUDS_ERRSTATUS_SERVICE_ALREADY_PENDING,
    PUDS_ERRSTATUS_SERVICE_TX_ERROR,
```

```

PUDS_ERRSTATUS_SERVICE_RX_ERROR,
PUDS_ERRSTATUS_SERVICE_RX_OVERFLOW,
PUDS_ERRSTATUS_MESSAGE_BUFFER_ALREADY_USED
};

```

Visual Basic

```

Public Enum uds_errstatus As Byte
    PUDS_ERRSTATUS_SERVICE_NO_MESSAGE = 1
    PUDS_ERRSTATUS_SERVICE_TIMEOUT_CONFIRMATION
    PUDS_ERRSTATUS_SERVICE_TIMEOUT_RESPONSE
    PUDS_ERRSTATUS_RESET
    PUDS_ERRSTATUS_ERROR_WAIT_FOR_P3_TIMING
    PUDS_ERRSTATUS_SERVICE_ALREADY_PENDING
    PUDS_ERRSTATUS_SERVICE_TX_ERROR
    PUDS_ERRSTATUS_SERVICE_RX_ERROR
    PUDS_ERRSTATUS_SERVICE_RX_OVERFLOW
    PUDS_ERRSTATUS_MESSAGE_BUFFER_ALREADY_USED
End Enum

```

Values

| Name | Value | Description |
|---|-------|---|
| PUDS_ERRSTATUS_SERVICE_NO_MESSAGE | 1 | No message available. |
| PUDS_ERRSTATUS_SERVICE_TIMEOUT_CONFIRMATION | 2 | Timeout while waiting message confirmation (loopback). |
| PUDS_ERRSTATUS_SERVICE_TIMEOUT_RESPONSE | 3 | Timeout while waiting for request message response. |
| PUDS_ERRSTATUS_RESET | 4 | UDS reset error. |
| PUDS_ERRSTATUS_ERROR_WAIT_FOR_P3_TIMING | 5 | UDS wait for P3 timing error. |
| PUDS_ERRSTATUS_SERVICE_ALREADY_PENDING | 6 | A UDS request is already pending. A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with UDS_Reset_2013 (see UDS_Reset_2013 on page 651). |
| PUDS_ERRSTATUS_SERVICE_TX_ERROR | 7 | An error occurred during the transmission of the UDS request message. |
| PUDS_ERRSTATUS_SERVICE_RX_ERROR | 8 | An error occurred during the reception of the UDS response message. |
| PUDS_ERRSTATUS_SERVICE_RX_OVERFLOW | 9 | Service received more messages than input buffer expected. |
| PUDS_ERRSTATUS_MESSAGE_BUFFER_ALREADY_USED | 10 | Given message buffer is already allocated, user must release buffer with UDS_MsgFree_2013 or MsgFree_2013 before reusing it. |

3.5.2 uds_status

Represents a PUDS status or error code.

Syntax

C/C++

```

typedef enum _uds_status {
    PUDS_STATUS_OK = PCANTP_STATUS_OK,
    PUDS_STATUS_NOT_INITIALIZED = PCANTP_STATUS_NOT_INITIALIZED,
    PUDS_STATUS_ALREADY_INITIALIZED = PCANTP_STATUS_ALREADY_INITIALIZED,
    PUDS_STATUS_NO_MEMORY = PCANTP_STATUS_NO_MEMORY,
    PUDS_STATUS_OVERFLOW = PCANTP_STATUS_OVERFLOW,
    PUDS_STATUS_NO_MESSAGE = PCANTP_STATUS_NO_MESSAGE,
    PUDS_STATUS_PARAM_INVALID_TYPE = PCANTP_STATUS_PARAM_INVALID_TYPE,
    PUDS_STATUS_PARAM_INVALID_VALUE = PCANTP_STATUS_PARAM_INVALID_VALUE,
    PUDS_STATUS_MAPPING_NOT_INITIALIZED = PCANTP_STATUS_MAPPING_NOT_INITIALIZED,

```



```

PUDS_STATUS_MAPPING_INVALID = PCANTP_STATUS_MAPPING_INVALID,
PUDS_STATUS_MAPPING_ALREADY_INITIALIZED = PCANTP_STATUS_MAPPING_ALREADY_INITIALIZED,
PUDS_STATUS_PARAM_BUFFER_TOO_SMALL = PCANTP_STATUS_PARAM_BUFFER_TOO_SMALL,
PUDS_STATUS_QUEUE_TX_FULL = PCANTP_STATUS_QUEUE_TX_FULL,
PUDS_STATUS_LOCK_TIMEOUT = PCANTP_STATUS_LOCK_TIMEOUT,
PUDS_STATUS_HANDLE_INVALID = PCANTP_STATUS_HANDLE_INVALID,
PUDS_STATUS_UNKNOWN = PCANTP_STATUS_UNKNOWN,
PUDS_STATUS_FLAG_BUS_LIGHT = PCANTP_STATUS_FLAG_BUS_LIGHT,
PUDS_STATUS_FLAG_BUS_HEAVY = PCANTP_STATUS_FLAG_BUS_HEAVY,
PUDS_STATUS_FLAG_BUS_WARNING = PCANTP_STATUS_FLAG_BUS_WARNING,
PUDS_STATUS_FLAG_BUS_PASSIVE = PCANTP_STATUS_FLAG_BUS_PASSIVE,
PUDS_STATUS_FLAG_BUS_OFF = PCANTP_STATUS_FLAG_BUS_OFF,
PUDS_STATUS_FLAG_BUS_ANY = PCANTP_STATUS_FLAG_BUS_ANY,
PUDS_STATUS_FLAG_NETWORK_RESULT = PCANTP_STATUS_FLAG_NETWORK_RESULT,
PUDS_STATUS_NETWORK_TIMEOUT_A = PCANTP_STATUS_NETWORK_TIMEOUT_A,
PUDS_STATUS_NETWORK_TIMEOUT_Bs = PCANTP_STATUS_NETWORK_TIMEOUT_Bs,
PUDS_STATUS_NETWORK_TIMEOUT_Cr = PCANTP_STATUS_NETWORK_TIMEOUT_Cr,
PUDS_STATUS_NETWORK_WRONG_SN = PCANTP_STATUS_NETWORK_WRONG_SN,
PUDS_STATUS_NETWORK_INVALID_FS = PCANTP_STATUS_NETWORK_INVALID_FS,
PUDS_STATUS_NETWORK_UNEXP_PDU = PCANTP_STATUS_NETWORK_UNEXP_PDU,
PUDS_STATUS_NETWORK_WFT_OVRN = PCANTP_STATUS_NETWORK_WFT_OVRN,
PUDS_STATUS_NETWORK_BUFFER_OVFLW = PCANTP_STATUS_NETWORK_BUFFER_OVFLW,
PUDS_STATUS_NETWORK_ERROR = PCANTP_STATUS_NETWORK_ERROR,
PUDS_STATUS_NETWORK_IGNORED = PCANTP_STATUS_NETWORK_IGNORED,
PUDS_STATUS_CAUTION_INPUT_MODIFIED = PCANTP_STATUS_CAUTION_INPUT_MODIFIED,
PUDS_STATUS_CAUTION_DLC_MODIFIED = PCANTP_STATUS_CAUTION_DLC_MODIFIED,
PUDS_STATUS_CAUTION_DATA_LENGTH_MODIFIED = PCANTP_STATUS_CAUTION_DATA_LENGTH_MODIFIED,
PUDS_STATUS_CAUTION_FD_FLAG_MODIFIED = PCANTP_STATUS_CAUTION_FD_FLAG_MODIFIED,
PUDS_STATUS_CAUTION_RX_QUEUE_FULL = PCANTP_STATUS_CAUTION_RX_QUEUE_FULL,
PUDS_STATUS_CAUTION_BUFFER_IN_USE = PCANTP_STATUS_CAUTION_BUFFER_IN_USE,
PUDS_STATUS_FLAG_PCAN_STATUS = PCANTP_STATUS_FLAG_PCAN_STATUS,
PUDS_STATUS_MASK_ERROR = PCANTP_STATUS_MASK_ERROR,
PUDS_STATUS_MASK_BUS = PCANTP_STATUS_MASK_BUS,
PUDS_STATUS_MASK_ISOTP_NET = PCANTP_STATUS_MASK_ISOTP_NET,
PUDS_STATUS_MASK_INFO = PCANTP_STATUS_MASK_INFO,
PUDS_STATUS_MASK_PCAN = PCANTP_STATUS_MASK_PCAN,
PUDS_STATUS_FLAG_UDS_ERROR = 0x20 << PCANTP_STATUS_OFFSET_UDS,
PUDS_STATUS_MASK_UDS_ERROR = 0x3f << PCANTP_STATUS_OFFSET_UDS,
PUDS_STATUS_SERVICE_NO_MESSAGE = PUDS_STATUS_FLAG_UDS_ERROR |
    (PUDS_ERRSTATUS_SERVICE_NO_MESSAGE << PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_SERVICE_TIMEOUT_CONFIRMATION = PUDS_STATUS_FLAG_UDS_ERROR |
    (PUDS_ERRSTATUS_SERVICE_TIMEOUT_CONFIRMATION << PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_SERVICE_TIMEOUT_RESPONSE = PUDS_STATUS_FLAG_UDS_ERROR |
    (PUDS_ERRSTATUS_SERVICE_TIMEOUT_RESPONSE << PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_RESET = PUDS_STATUS_FLAG_UDS_ERROR | (PUDS_ERRSTATUS_RESET <<
    PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_ERROR_WAIT_FOR_P3_TIMING = PUDS_STATUS_FLAG_UDS_ERROR |
    (PUDS_ERRSTATUS_ERROR_WAIT_FOR_P3_TIMING << PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_SERVICE_ALREADY_PENDING = PUDS_STATUS_FLAG_UDS_ERROR |
    (PUDS_ERRSTATUS_SERVICE_ALREADY_PENDING << PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_SERVICE_TX_ERROR = PUDS_STATUS_FLAG_UDS_ERROR |
    (PUDS_ERRSTATUS_SERVICE_TX_ERROR << PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_SERVICE_RX_ERROR = PUDS_STATUS_FLAG_UDS_ERROR |
    (PUDS_ERRSTATUS_SERVICE_RX_ERROR << PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_SERVICE_RX_OVERFLOW = PUDS_STATUS_FLAG_UDS_ERROR |
    (PUDS_ERRSTATUS_SERVICE_RX_OVERFLOW << PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED = PUDS_STATUS_FLAG_UDS_ERROR |
    (PUDS_ERRSTATUS_MESSAGE_BUFFER_ALREADY_USED << PCANTP_STATUS_OFFSET_UDS),
} uds_status;

```

C++ / CLR

```

public enum uds_status : UInt32
{
    PUDS_STATUS_OK = cantp_status::PCANTP_STATUS_OK,
    PUDS_STATUS_NOT_INITIALIZED = cantp_status::PCANTP_STATUS_NOT_INITIALIZED,
    PUDS_STATUS_ALREADY_INITIALIZED = cantp_status::PCANTP_STATUS_ALREADY_INITIALIZED,
    PUDS_STATUS_NO_MEMORY = cantp_status::PCANTP_STATUS_NO_MEMORY,
    PUDS_STATUS_OVERFLOW = cantp_status::PCANTP_STATUS_OVERFLOW,
    PUDS_STATUS_NO_MESSAGE = cantp_status::PCANTP_STATUS_NO_MESSAGE,
    PUDS_STATUS_PARAM_INVALID_TYPE = cantp_status::PCANTP_STATUS_PARAM_INVALID_TYPE,
    PUDS_STATUS_PARAM_INVALID_VALUE = cantp_status::PCANTP_STATUS_PARAM_INVALID_VALUE,
    PUDS_STATUS_MAPPING_NOT_INITIALIZED = cantp_status::PCANTP_STATUS_MAPPING_NOT_INITIALIZED,
    PUDS_STATUS_MAPPING_INVALID = cantp_status::PCANTP_STATUS_MAPPING_INVALID,
    PUDS_STATUS_MAPPING_ALREADY_INITIALIZED =
        cantp_status::PCANTP_STATUS_MAPPING_ALREADY_INITIALIZED,
    PUDS_STATUS_PARAM_BUFFER_TOO_SMALL = cantp_status::PCANTP_STATUS_PARAM_BUFFER_TOO_SMALL,
    PUDS_STATUS_QUEUE_TX_FULL = cantp_status::PCANTP_STATUS_QUEUE_TX_FULL,
    PUDS_STATUS_LOCK_TIMEOUT = cantp_status::PCANTP_STATUS_LOCK_TIMEOUT,
    PUDS_STATUS_HANDLE_INVALID = cantp_status::PCANTP_STATUS_HANDLE_INVALID,
    PUDS_STATUS_UNKNOWN = cantp_status::PCANTP_STATUS_UNKNOWN,
    PUDS_STATUS_FLAG_BUS_LIGHT = cantp_status::PCANTP_STATUS_FLAG_BUS_LIGHT,
    PUDS_STATUS_FLAG_BUS_HEAVY = cantp_status::PCANTP_STATUS_FLAG_BUS_HEAVY,
    PUDS_STATUS_FLAG_BUS_WARNING = cantp_status::PCANTP_STATUS_FLAG_BUS_WARNING,
    PUDS_STATUS_FLAG_BUS_PASSIVE = cantp_status::PCANTP_STATUS_FLAG_BUS_PASSIVE,
    PUDS_STATUS_FLAG_BUS_OFF = cantp_status::PCANTP_STATUS_FLAG_BUS_OFF,
    PUDS_STATUS_FLAG_BUS_ANY = cantp_status::PCANTP_STATUS_FLAG_BUS_ANY,
    PUDS_STATUS_FLAG_NETWORK_RESULT = cantp_status::PCANTP_STATUS_FLAG_NETWORK_RESULT,
    PUDS_STATUS_NETWORK_TIMEOUT_A = cantp_status::PCANTP_STATUS_NETWORK_TIMEOUT_A,
    PUDS_STATUS_NETWORK_TIMEOUT_Bs = cantp_status::PCANTP_STATUS_NETWORK_TIMEOUT_Bs,
    PUDS_STATUS_NETWORK_TIMEOUT_Cr = cantp_status::PCANTP_STATUS_NETWORK_TIMEOUT_Cr,
    PUDS_STATUS_NETWORK_WRONG_SN = cantp_status::PCANTP_STATUS_NETWORK_WRONG_SN,
    PUDS_STATUS_NETWORK_INVALID_FS = cantp_status::PCANTP_STATUS_NETWORK_INVALID_FS,
    PUDS_STATUS_NETWORK_UNEXP_PDU = cantp_status::PCANTP_STATUS_NETWORK_UNEXP_PDU,
    PUDS_STATUS_NETWORK_WFT_OVRN = cantp_status::PCANTP_STATUS_NETWORK_WFT_OVRN,
    PUDS_STATUS_NETWORK_BUFFER_OVFLW = cantp_status::PCANTP_STATUS_NETWORK_BUFFER_OVFLW,
    PUDS_STATUS_NETWORK_ERROR = cantp_status::PCANTP_STATUS_NETWORK_ERROR,
    PUDS_STATUS_NETWORK_IGNORED = cantp_status::PCANTP_STATUS_NETWORK_IGNORED,
    PUDS_STATUS_CAUTION_INPUT_MODIFIED = cantp_status::PCANTP_STATUS_CAUTION_INPUT_MODIFIED,
    PUDS_STATUS_CAUTION_DLC_MODIFIED = cantp_status::PCANTP_STATUS_CAUTION_DLC_MODIFIED,
    PUDS_STATUS_CAUTION_DATA_LENGTH_MODIFIED =
        cantp_status::PCANTP_STATUS_CAUTION_DATA_LENGTH_MODIFIED,
    PUDS_STATUS_CAUTION_FD_FLAG_MODIFIED = cantp_status::PCANTP_STATUS_CAUTION_FD_FLAG_MODIFIED,
    PUDS_STATUS_CAUTION_RX_QUEUE_FULL = cantp_status::PCANTP_STATUS_CAUTION_RX_QUEUE_FULL,
    PUDS_STATUS_CAUTION_BUFFER_IN_USE = cantp_status::PCANTP_STATUS_CAUTION_BUFFER_IN_USE,
    PUDS_STATUS_FLAG_PCAN_STATUS = cantp_status::PCANTP_STATUS_FLAG_PCAN_STATUS,
    PUDS_STATUS_MASK_ERROR = cantp_status::PCANTP_STATUS_MASK_ERROR,
    PUDS_STATUS_MASK_BUS = cantp_status::PCANTP_STATUS_MASK_BUS,
    PUDS_STATUS_MASK_ISOTP_NET = cantp_status::PCANTP_STATUS_MASK_ISOTP_NET,
    PUDS_STATUS_MASK_INFO = cantp_status::PCANTP_STATUS_MASK_INFO,
    PUDS_STATUS_MASK_PCAN = cantp_status::PCANTP_STATUS_MASK_PCAN,
    PUDS_STATUS_FLAG_UDS_ERROR = 0x20 << uds_status_offset::PCANTP_STATUS_OFFSET_UDS,
    PUDS_STATUS_MASK_UDS_ERROR = (UInt32)0x3f << uds_status_offset::PCANTP_STATUS_OFFSET_UDS,
    PUDS_STATUS_SERVICE_NO_MESSAGE = PUDS_STATUS_FLAG_UDS_ERROR |
        (uds_errstatus::PUDS_ERRSTATUS_SERVICE_NO_MESSAGE <<
        uds_status_offset::PCANTP_STATUS_OFFSET_UDS),
    PUDS_STATUS_SERVICE_TIMEOUT_CONFIRMATION = PUDS_STATUS_FLAG_UDS_ERROR |
        (uds_errstatus::PUDS_ERRSTATUS_SERVICE_TIMEOUT_CONFIRMATION <<
        uds_status_offset::PCANTP_STATUS_OFFSET_UDS),
    PUDS_STATUS_SERVICE_TIMEOUT_RESPONSE = PUDS_STATUS_FLAG_UDS_ERROR |
        (uds_errstatus::PUDS_ERRSTATUS_SERVICE_TIMEOUT_RESPONSE <<
        uds_status_offset::PCANTP_STATUS_OFFSET_UDS),

```

```

PUDS_STATUS_RESET = PUDS_STATUS_FLAG_UDS_ERROR | (uds_errstatus::PUDS_ERRSTATUS_RESET <<
    uds_status_offset::PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_ERROR_WAIT_FOR_P3_TIMING = PUDS_STATUS_FLAG_UDS_ERROR |
    (uds_errstatus::PUDS_ERRSTATUS_ERROR_WAIT_FOR_P3_TIMING <<
    uds_status_offset::PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_SERVICE_ALREADY_PENDING = PUDS_STATUS_FLAG_UDS_ERROR |
    (uds_errstatus::PUDS_ERRSTATUS_SERVICE_ALREADY_PENDING <<
    uds_status_offset::PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_SERVICE_TX_ERROR = PUDS_STATUS_FLAG_UDS_ERROR |
    (uds_errstatus::PUDS_ERRSTATUS_SERVICE_TX_ERROR <<
    uds_status_offset::PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_SERVICE_RX_ERROR = PUDS_STATUS_FLAG_UDS_ERROR |
    (uds_errstatus::PUDS_ERRSTATUS_SERVICE_RX_ERROR <<
    uds_status_offset::PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_SERVICE_RX_OVERFLOW = PUDS_STATUS_FLAG_UDS_ERROR |
    (uds_errstatus::PUDS_ERRSTATUS_SERVICE_RX_OVERFLOW <<
    uds_status_offset::PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED = PUDS_STATUS_FLAG_UDS_ERROR |
    (uds_errstatus::PUDS_ERRSTATUS_MESSAGE_BUFFER_ALREADY_USED <<
    uds_status_offset::PCANTP_STATUS_OFFSET_UDS)
};

```

C#

```

public enum uds_status : UInt32
{
    PUDS_STATUS_OK = cantp_status.PCANTP_STATUS_OK,
    PUDS_STATUS_NOT_INITIALIZED = cantp_status.PCANTP_STATUS_NOT_INITIALIZED,
    PUDS_STATUS_ALREADY_INITIALIZED = cantp_status.PCANTP_STATUS_ALREADY_INITIALIZED,
    PUDS_STATUS_NO_MEMORY = cantp_status.PCANTP_STATUS_NO_MEMORY,
    PUDS_STATUS_OVERFLOW = cantp_status.PCANTP_STATUS_OVERFLOW,
    PUDS_STATUS_NO_MESSAGE = cantp_status.PCANTP_STATUS_NO_MESSAGE,
    PUDS_STATUS_PARAM_INVALID_TYPE = cantp_status.PCANTP_STATUS_PARAM_INVALID_TYPE,
    PUDS_STATUS_PARAM_INVALID_VALUE = cantp_status.PCANTP_STATUS_PARAM_INVALID_VALUE,
    PUDS_STATUS_MAPPING_NOT_INITIALIZED = cantp_status.PCANTP_STATUS_MAPPING_NOT_INITIALIZED,
    PUDS_STATUS_MAPPING_INVALID = cantp_status.PCANTP_STATUS_MAPPING_INVALID,
    PUDS_STATUS_MAPPING_ALREADY_INITIALIZED =
        cantp_status.PCANTP_STATUS_MAPPING_ALREADY_INITIALIZED,
    PUDS_STATUS_PARAM_BUFFER_TOO_SMALL = cantp_status.PCANTP_STATUS_PARAM_BUFFER_TOO_SMALL,
    PUDS_STATUS_QUEUE_TX_FULL = cantp_status.PCANTP_STATUS_QUEUE_TX_FULL,
    PUDS_STATUS_LOCK_TIMEOUT = cantp_status.PCANTP_STATUS_LOCK_TIMEOUT,
    PUDS_STATUS_HANDLE_INVALID = cantp_status.PCANTP_STATUS_HANDLE_INVALID,
    PUDS_STATUS_UNKNOWN = cantp_status.PCANTP_STATUS_UNKNOWN,
    PUDS_STATUS_FLAG_BUS_LIGHT = cantp_status.PCANTP_STATUS_FLAG_BUS_LIGHT,
    PUDS_STATUS_FLAG_BUS_HEAVY = cantp_status.PCANTP_STATUS_FLAG_BUS_HEAVY,
    PUDS_STATUS_FLAG_BUS_WARNING = cantp_status.PCANTP_STATUS_FLAG_BUS_WARNING,
    PUDS_STATUS_FLAG_BUS_PASSIVE = cantp_status.PCANTP_STATUS_FLAG_BUS_PASSIVE,
    PUDS_STATUS_FLAG_BUS_OFF = cantp_status.PCANTP_STATUS_FLAG_BUS_OFF,
    PUDS_STATUS_FLAG_BUS_ANY = cantp_status.PCANTP_STATUS_FLAG_BUS_ANY,
    PUDS_STATUS_FLAG_NETWORK_RESULT = cantp_status.PCANTP_STATUS_FLAG_NETWORK_RESULT,
    PUDS_STATUS_NETWORK_TIMEOUT_A = cantp_status.PCANTP_STATUS_NETWORK_TIMEOUT_A,
    PUDS_STATUS_NETWORK_TIMEOUT_Bs = cantp_status.PCANTP_STATUS_NETWORK_TIMEOUT_Bs,
    PUDS_STATUS_NETWORK_TIMEOUT_Cr = cantp_status.PCANTP_STATUS_NETWORK_TIMEOUT_Cr,
    PUDS_STATUS_NETWORK_WRONG_SN = cantp_status.PCANTP_STATUS_NETWORK_WRONG_SN,
    PUDS_STATUS_NETWORK_INVALID_FS = cantp_status.PCANTP_STATUS_NETWORK_INVALID_FS,
    PUDS_STATUS_NETWORK_UNEXP_PDU = cantp_status.PCANTP_STATUS_NETWORK_UNEXP_PDU,
    PUDS_STATUS_NETWORK_WFT_OVRN = cantp_status.PCANTP_STATUS_NETWORK_WFT_OVRN,
    PUDS_STATUS_NETWORK_BUFFER_OVFLW = cantp_status.PCANTP_STATUS_NETWORK_BUFFER_OVFLW,
    PUDS_STATUS_NETWORK_ERROR = cantp_status.PCANTP_STATUS_NETWORK_ERROR,
    PUDS_STATUS_NETWORK_IGNORED = cantp_status.PCANTP_STATUS_NETWORK_IGNORED,
    PUDS_STATUS_CAUTION_INPUT_MODIFIED = cantp_status.PCANTP_STATUS_CAUTION_INPUT_MODIFIED,
    PUDS_STATUS_CAUTION_DLC_MODIFIED = cantp_status.PCANTP_STATUS_CAUTION_DLC_MODIFIED,
}

```

```

PUDS_STATUS_CAUTION_DATA_LENGTH_MODIFIED =
    cantp_status.PCANTP_STATUS_CAUTION_DATA_LENGTH_MODIFIED,
PUDS_STATUS_CAUTION_FD_FLAG_MODIFIED = cantp_status.PCANTP_STATUS_CAUTION_FD_FLAG_MODIFIED,
PUDS_STATUS_CAUTION_RX_QUEUE_FULL = cantp_status.PCANTP_STATUS_CAUTION_RX_QUEUE_FULL,
PUDS_STATUS_CAUTION_BUFFER_IN_USE = cantp_status.PCANTP_STATUS_CAUTION_BUFFER_IN_USE,
PUDS_STATUS_FLAG_PCAN_STATUS = cantp_status.PCANTP_STATUS_FLAG_PCAN_STATUS,
PUDS_STATUS_MASK_ERROR = cantp_status.PCANTP_STATUS_MASK_ERROR,
PUDS_STATUS_MASK_BUS = cantp_status.PCANTP_STATUS_MASK_BUS,
PUDS_STATUS_MASK_ISOTP_NET = cantp_status.PCANTP_STATUS_MASK_ISOTP_NET,
PUDS_STATUS_MASK_INFO = cantp_status.PCANTP_STATUS_MASK_INFO,
PUDS_STATUS_MASK_PCAN = cantp_status.PCANTP_STATUS_MASK_PCAN,
PUDS_STATUS_FLAG_UDS_ERROR = 0x20 << uds_status_offset.PCANTP_STATUS_OFFSET_UDS,
PUDS_STATUS_MASK_UDS_ERROR = (UInt32)0x3f << uds_status_offset.PCANTP_STATUS_OFFSET_UDS,
PUDS_STATUS_SERVICE_NO_MESSAGE = PUDS_STATUS_FLAG_UDS_ERROR |
    (uds_errstatus.PUDS_ERRSTATUS_SERVICE_NO_MESSAGE <<
    uds_status_offset.PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_SERVICE_TIMEOUT_CONFIRMATION = PUDS_STATUS_FLAG_UDS_ERROR |
    (uds_errstatus.PUDS_ERRSTATUS_SERVICE_TIMEOUT_CONFIRMATION <<
    uds_status_offset.PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_SERVICE_TIMEOUT_RESPONSE = PUDS_STATUS_FLAG_UDS_ERROR |
    (uds_errstatus.PUDS_ERRSTATUS_SERVICE_TIMEOUT_RESPONSE <<
    uds_status_offset.PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_RESET = PUDS_STATUS_FLAG_UDS_ERROR | (uds_errstatus.PUDS_ERRSTATUS_RESET <<
    uds_status_offset.PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_ERROR_WAIT_FOR_P3_TIMING = PUDS_STATUS_FLAG_UDS_ERROR |
    (uds_errstatus.PUDS_ERRSTATUS_ERROR_WAIT_FOR_P3_TIMING <<
    uds_status_offset.PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_SERVICE_ALREADY_PENDING = PUDS_STATUS_FLAG_UDS_ERROR |
    (uds_errstatus.PUDS_ERRSTATUS_SERVICE_ALREADY_PENDING <<
    uds_status_offset.PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_SERVICE_TX_ERROR = PUDS_STATUS_FLAG_UDS_ERROR |
    (uds_errstatus.PUDS_ERRSTATUS_SERVICE_TX_ERROR <<
    uds_status_offset.PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_SERVICE_RX_ERROR = PUDS_STATUS_FLAG_UDS_ERROR |
    (uds_errstatus.PUDS_ERRSTATUS_SERVICE_RX_ERROR <<
    uds_status_offset.PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_SERVICE_RX_OVERFLOW = PUDS_STATUS_FLAG_UDS_ERROR |
    (uds_errstatus.PUDS_ERRSTATUS_SERVICE_RX_OVERFLOW <<
    uds_status_offset.PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED = PUDS_STATUS_FLAG_UDS_ERROR |
    (uds_errstatus.PUDS_ERRSTATUS_MESSAGE_BUFFER_ALREADY_USED <<
    uds_status_offset.PCANTP_STATUS_OFFSET_UDS)
}

```

Pascal OO

```

uds_status = (
    PUDS_STATUS_OK = UInt32(PCANTP_STATUS_OK),
    PUDS_STATUS_NOT_INITIALIZED = UInt32(PCANTP_STATUS_NOT_INITIALIZED),
    PUDS_STATUS_ALREADY_INITIALIZED = UInt32(PCANTP_STATUS_ALREADY_INITIALIZED),
    PUDS_STATUS_NO_MEMORY = UInt32(PCANTP_STATUS_NO_MEMORY),
    PUDS_STATUS_OVERFLOW = UInt32(PCANTP_STATUS_OVERFLOW),
    PUDS_STATUS_NO_MESSAGE = UInt32(PCANTP_STATUS_NO_MESSAGE),
    PUDS_STATUS_PARAM_INVALID_TYPE = UInt32(PCANTP_STATUS_PARAM_INVALID_TYPE),
    PUDS_STATUS_PARAM_INVALID_VALUE = UInt32(PCANTP_STATUS_PARAM_INVALID_VALUE),
    PUDS_STATUS_MAPPING_NOT_INITIALIZED = UInt32(PCANTP_STATUS_MAPPING_NOT_INITIALIZED),
    PUDS_STATUS_MAPPING_INVALID = UInt32(PCANTP_STATUS_MAPPING_INVALID),
    PUDS_STATUS_MAPPING_ALREADY_INITIALIZED = UInt32(PCANTP_STATUS_MAPPING_ALREADY_INITIALIZED),
    PUDS_STATUS_PARAM_BUFFER_TOO_SMALL = UInt32(PCANTP_STATUS_PARAM_BUFFER_TOO_SMALL),
    PUDS_STATUS_QUEUE_TX_FULL = UInt32(PCANTP_STATUS_QUEUE_TX_FULL),
    PUDS_STATUS_LOCK_TIMEOUT = UInt32(PCANTP_STATUS_LOCK_TIMEOUT),
    PUDS_STATUS_HANDLE_INVALID = UInt32(PCANTP_STATUS_HANDLE_INVALID),
    PUDS_STATUS_UNKNOWN = UInt32(PCANTP_STATUS_UNKNOWN),

```



```

PUDS_STATUS_FLAG_BUS_LIGHT = UInt32(PCANTP_STATUS_FLAG_BUS_LIGHT),
PUDS_STATUS_FLAG_BUS_HEAVY = UInt32(PCANTP_STATUS_FLAG_BUS_HEAVY),
PUDS_STATUS_FLAG_BUS_WARNING = UInt32(PCANTP_STATUS_FLAG_BUS_WARNING),
PUDS_STATUS_FLAG_BUS_PASSIVE = UInt32(PCANTP_STATUS_FLAG_BUS_PASSIVE),
PUDS_STATUS_FLAG_BUS_OFF = UInt32(PCANTP_STATUS_FLAG_BUS_OFF),
PUDS_STATUS_FLAG_BUS_ANY = UInt32(PCANTP_STATUS_FLAG_BUS_ANY),
PUDS_STATUS_FLAG_NETWORK_RESULT = UInt32(PCANTP_STATUS_FLAG_NETWORK_RESULT),
PUDS_STATUS_NETWORK_TIMEOUT_A = UInt32(PCANTP_STATUS_NETWORK_TIMEOUT_A),
PUDS_STATUS_NETWORK_TIMEOUT_Bs = UInt32(PCANTP_STATUS_NETWORK_TIMEOUT_Bs),
PUDS_STATUS_NETWORK_TIMEOUT_Cr = UInt32(PCANTP_STATUS_NETWORK_TIMEOUT_Cr),
PUDS_STATUS_NETWORK_WRONG_SN = UInt32(PCANTP_STATUS_NETWORK_WRONG_SN),
PUDS_STATUS_NETWORK_INVALID_FS = UInt32(PCANTP_STATUS_NETWORK_INVALID_FS),
PUDS_STATUS_NETWORK_UNEXP_PDU = UInt32(PCANTP_STATUS_NETWORK_UNEXP_PDU),
PUDS_STATUS_NETWORK_WFT_OVRN = UInt32(PCANTP_STATUS_NETWORK_WFT_OVRN),
PUDS_STATUS_NETWORK_BUFFER_OVFLW = UInt32(PCANTP_STATUS_NETWORK_BUFFER_OVFLW),
PUDS_STATUS_NETWORK_ERROR = UInt32(PCANTP_STATUS_NETWORK_ERROR),
PUDS_STATUS_NETWORK_IGNORED = UInt32(PCANTP_STATUS_NETWORK_IGNORED),
PUDS_STATUS_CAUTION_INPUT_MODIFIED = UInt32(PCANTP_STATUS_CAUTION_INPUT_MODIFIED),
PUDS_STATUS_CAUTION_DLC_MODIFIED = UInt32(PCANTP_STATUS_CAUTION_DLC_MODIFIED),
PUDS_STATUS_CAUTION_DATA_LENGTH_MODIFIED =
    UInt32(PCANTP_STATUS_CAUTION_DATA_LENGTH_MODIFIED),
PUDS_STATUS_CAUTION_FD_FLAG_MODIFIED = UInt32(PCANTP_STATUS_CAUTION_FD_FLAG_MODIFIED),
PUDS_STATUS_CAUTION_RX_QUEUE_FULL = UInt32(PCANTP_STATUS_CAUTION_RX_QUEUE_FULL),
PUDS_STATUS_CAUTION_BUFFER_IN_USE = UInt32(PCANTP_STATUS_CAUTION_BUFFER_IN_USE),
PUDS_STATUS_FLAG_PCAN_STATUS = UInt32(PCANTP_STATUS_FLAG_PCAN_STATUS),
PUDS_STATUS_MASK_ERROR = UInt32(PCANTP_STATUS_MASK_ERROR),
PUDS_STATUS_MASK_BUS = UInt32(PCANTP_STATUS_MASK_BUS),
PUDS_STATUS_MASK_ISOTP_NET = UInt32(PCANTP_STATUS_MASK_ISOTP_NET),
PUDS_STATUS_MASK_INFO = UInt32(PCANTP_STATUS_MASK_INFO),
PUDS_STATUS_MASK_PCAN = UInt32(PCANTP_STATUS_MASK_PCAN),
PUDS_STATUS_FLAG_UDS_ERROR = $20 Shl PCANTP_STATUS_OFFSET_UDS,
PUDS_STATUS_MASK_UDS_ERROR = $3F Shl PCANTP_STATUS_OFFSET_UDS,
PUDS_STATUS_SERVICE_NO_MESSAGE = PUDS_STATUS_FLAG_UDS_ERROR Or
    (UInt32(PUDS_ERRSTATUS_SERVICE_NO_MESSAGE) shl PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_SERVICE_TIMEOUT_CONFIRMATION = PUDS_STATUS_FLAG_UDS_ERROR Or
    (UInt32(PUDS_ERRSTATUS_SERVICE_TIMEOUT_CONFIRMATION) shl PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_SERVICE_TIMEOUT_RESPONSE = PUDS_STATUS_FLAG_UDS_ERROR Or
    (UInt32(PUDS_ERRSTATUS_SERVICE_TIMEOUT_RESPONSE) shl PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_RESET = PUDS_STATUS_FLAG_UDS_ERROR Or
    (UInt32(PUDS_ERRSTATUS_RESET) shl PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_ERROR_WAIT_FOR_P3_TIMING = PUDS_STATUS_FLAG_UDS_ERROR Or
    (UInt32(PUDS_ERRSTATUS_ERROR_WAIT_FOR_P3_TIMING) shl PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_SERVICE_ALREADY_PENDING = PUDS_STATUS_FLAG_UDS_ERROR Or
    (UInt32(PUDS_ERRSTATUS_SERVICE_ALREADY_PENDING) shl PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_SERVICE_TX_ERROR = PUDS_STATUS_FLAG_UDS_ERROR Or
    (UInt32(PUDS_ERRSTATUS_SERVICE_TX_ERROR) shl PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_SERVICE_RX_ERROR = PUDS_STATUS_FLAG_UDS_ERROR Or
    (UInt32(PUDS_ERRSTATUS_SERVICE_RX_ERROR) shl PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_SERVICE_RX_OVERFLOW = PUDS_STATUS_FLAG_UDS_ERROR Or
    (UInt32(PUDS_ERRSTATUS_SERVICE_RX_OVERFLOW) shl PCANTP_STATUS_OFFSET_UDS),
PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED = PUDS_STATUS_FLAG_UDS_ERROR Or
    (UInt32(PUDS_ERRSTATUS_MESSAGE_BUFFER_ALREADY_USED) shl PCANTP_STATUS_OFFSET_UDS)
);

```

Visual Basic

```

Public Enum uds_status As UInt32
    PUDS_STATUS_OK = cantp_status.PCANTP_STATUS_OK
    PUDS_STATUS_NOT_INITIALIZED = cantp_status.PCANTP_STATUS_NOT_INITIALIZED
    PUDS_STATUS_ALREADY_INITIALIZED = cantp_status.PCANTP_STATUS_ALREADY_INITIALIZED
    PUDS_STATUS_NO_MEMORY = cantp_status.PCANTP_STATUS_NO_MEMORY
    PUDS_STATUS_OVERFLOW = cantp_status.PCANTP_STATUS_OVERFLOW

```

```

PUDS_STATUS_NO_MESSAGE = cantp_status.PCANTP_STATUS_NO_MESSAGE
PUDS_STATUS_PARAM_INVALID_TYPE = cantp_status.PCANTP_STATUS_PARAM_INVALID_TYPE
PUDS_STATUS_PARAM_INVALID_VALUE = cantp_status.PCANTP_STATUS_PARAM_INVALID_VALUE
PUDS_STATUS_MAPPING_NOT_INITIALIZED = cantp_status.PCANTP_STATUS_MAPPING_NOT_INITIALIZED
PUDS_STATUS_MAPPING_INVALID = cantp_status.PCANTP_STATUS_MAPPING_INVALID
PUDS_STATUS_MAPPING_ALREADY_INITIALIZED =
    cantp_status.PCANTP_STATUS_MAPPING_ALREADY_INITIALIZED
PUDS_STATUS_PARAM_BUFFER_TOO_SMALL = cantp_status.PCANTP_STATUS_PARAM_BUFFER_TOO_SMALL
PUDS_STATUS_QUEUE_TX_FULL = cantp_status.PCANTP_STATUS_QUEUE_TX_FULL
PUDS_STATUS_LOCK_TIMEOUT = cantp_status.PCANTP_STATUS_LOCK_TIMEOUT
PUDS_STATUS_HANDLE_INVALID = cantp_status.PCANTP_STATUS_HANDLE_INVALID
PUDS_STATUS_UNKNOWN = cantp_status.PCANTP_STATUS_UNKNOWN
PUDS_STATUS_FLAG_BUS_LIGHT = cantp_status.PCANTP_STATUS_FLAG_BUS_LIGHT
PUDS_STATUS_FLAG_BUS_HEAVY = cantp_status.PCANTP_STATUS_FLAG_BUS_HEAVY
PUDS_STATUS_FLAG_BUS_WARNING = cantp_status.PCANTP_STATUS_FLAG_BUS_WARNING
PUDS_STATUS_FLAG_BUS_PASSIVE = cantp_status.PCANTP_STATUS_FLAG_BUS_PASSIVE
PUDS_STATUS_FLAG_BUS_OFF = cantp_status.PCANTP_STATUS_FLAG_BUS_OFF
PUDS_STATUS_FLAG_BUS_ANY = cantp_status.PCANTP_STATUS_FLAG_BUS_ANY
PUDS_STATUS_FLAG_NETWORK_RESULT = cantp_status.PCANTP_STATUS_FLAG_NETWORK_RESULT
PUDS_STATUS_NETWORK_TIMEOUT_A = cantp_status.PCANTP_STATUS_NETWORK_TIMEOUT_A
PUDS_STATUS_NETWORK_TIMEOUT_Bs = cantp_status.PCANTP_STATUS_NETWORK_TIMEOUT_Bs
PUDS_STATUS_NETWORK_TIMEOUT_Cr = cantp_status.PCANTP_STATUS_NETWORK_TIMEOUT_Cr
PUDS_STATUS_NETWORK_WRONG_SN = cantp_status.PCANTP_STATUS_NETWORK_WRONG_SN
PUDS_STATUS_NETWORK_INVALID_FS = cantp_status.PCANTP_STATUS_NETWORK_INVALID_FS
PUDS_STATUS_NETWORK_UNEXP_PDU = cantp_status.PCANTP_STATUS_NETWORK_UNEXP_PDU
PUDS_STATUS_NETWORK_WFT_OVRN = cantp_status.PCANTP_STATUS_NETWORK_WFT_OVRN
PUDS_STATUS_NETWORK_BUFFER_OVFLW = cantp_status.PCANTP_STATUS_NETWORK_BUFFER_OVFLW
PUDS_STATUS_NETWORK_ERROR = cantp_status.PCANTP_STATUS_NETWORK_ERROR
PUDS_STATUS_NETWORK_IGNORED = cantp_status.PCANTP_STATUS_NETWORK_IGNORED
PUDS_STATUS_CAUTION_INPUT_MODIFIED = cantp_status.PCANTP_STATUS_CAUTION_INPUT_MODIFIED
PUDS_STATUS_CAUTION_DLC_MODIFIED = cantp_status.PCANTP_STATUS_CAUTION_DLC_MODIFIED
PUDS_STATUS_CAUTION_DATA_LENGTH_MODIFIED =
    cantp_status.PCANTP_STATUS_CAUTION_DATA_LENGTH_MODIFIED
PUDS_STATUS_CAUTION_FD_FLAG_MODIFIED = cantp_status.PCANTP_STATUS_CAUTION_FD_FLAG_MODIFIED
PUDS_STATUS_CAUTION_RX_QUEUE_FULL = cantp_status.PCANTP_STATUS_CAUTION_RX_QUEUE_FULL
PUDS_STATUS_CAUTION_BUFFER_IN_USE = cantp_status.PCANTP_STATUS_CAUTION_BUFFER_IN_USE
PUDS_STATUS_FLAG_PCAN_STATUS = cantp_status.PCANTP_STATUS_FLAG_PCAN_STATUS
PUDS_STATUS_MASK_ERROR = cantp_status.PCANTP_STATUS_MASK_ERROR
PUDS_STATUS_MASK_BUS = cantp_status.PCANTP_STATUS_MASK_BUS
PUDS_STATUS_MASK_ISOTP_NET = cantp_status.PCANTP_STATUS_MASK_ISOTP_NET
PUDS_STATUS_MASK_INFO = cantp_status.PCANTP_STATUS_MASK_INFO
PUDS_STATUS_MASK_PCAN = cantp_status.PCANTP_STATUS_MASK_PCAN
PUDS_STATUS_FLAG_UDS_ERROR = &H20 << uds_status_offset.PCANTP_STATUS_OFFSET_UDS
PUDS_STATUS_MASK_UDS_ERROR = CType(&H3F, UInt32) <<
    uds_status_offset.PCANTP_STATUS_OFFSET_UDS
PUDS_STATUS_SERVICE_NO_MESSAGE = PUDS_STATUS_FLAG_UDS_ERROR Or
    (uds_errstatus.PUDS_ERRSTATUS_SERVICE_NO_MESSAGE <<
        uds_status_offset.PCANTP_STATUS_OFFSET_UDS)
PUDS_STATUS_SERVICE_TIMEOUT_CONFIRMATION = PUDS_STATUS_FLAG_UDS_ERROR Or
    (uds_errstatus.PUDS_ERRSTATUS_SERVICE_TIMEOUT_CONFIRMATION <<
        uds_status_offset.PCANTP_STATUS_OFFSET_UDS)
PUDS_STATUS_SERVICE_TIMEOUT_RESPONSE = PUDS_STATUS_FLAG_UDS_ERROR Or
    (uds_errstatus.PUDS_ERRSTATUS_SERVICE_TIMEOUT_RESPONSE <<
        uds_status_offset.PCANTP_STATUS_OFFSET_UDS)
PUDS_STATUS_RESET = PUDS_STATUS_FLAG_UDS_ERROR Or (uds_errstatus.PUDS_ERRSTATUS_RESET <<
    uds_status_offset.PCANTP_STATUS_OFFSET_UDS)
PUDS_STATUS_ERROR_WAIT_FOR_P3_TIMING = PUDS_STATUS_FLAG_UDS_ERROR Or
    (uds_errstatus.PUDS_ERRSTATUS_ERROR_WAIT_FOR_P3_TIMING <<
        uds_status_offset.PCANTP_STATUS_OFFSET_UDS)
PUDS_STATUS_SERVICE_ALREADY_PENDING = PUDS_STATUS_FLAG_UDS_ERROR Or
    (uds_errstatus.PUDS_ERRSTATUS_SERVICE_ALREADY_PENDING <<
        uds_status_offset.PCANTP_STATUS_OFFSET_UDS)

```

```

PUDS_STATUS_SERVICE_TX_ERROR = PUDS_STATUS_FLAG_UDS_ERROR Or
    (uds_errstatus.PUDS_ERRSTATUS_SERVICE_TX_ERROR <<
    uds_status_offset.PCANTP_STATUS_OFFSET_UDS)
PUDS_STATUS_SERVICE_RX_ERROR = PUDS_STATUS_FLAG_UDS_ERROR Or
    (uds_errstatus.PUDS_ERRSTATUS_SERVICE_RX_ERROR <<
    uds_status_offset.PCANTP_STATUS_OFFSET_UDS)
PUDS_STATUS_SERVICE_RX_OVERFLOW = PUDS_STATUS_FLAG_UDS_ERROR Or
    (uds_errstatus.PUDS_ERRSTATUS_SERVICE_RX_OVERFLOW <<
    uds_status_offset.PCANTP_STATUS_OFFSET_UDS)
PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED = PUDS_STATUS_FLAG_UDS_ERROR Or
    (uds_errstatus.PUDS_ERRSTATUS_MESSAGE_BUFFER_ALREADY_USED <<
    uds_status_offset.PCANTP_STATUS_OFFSET_UDS)

```

End Enum

values

| Name | Value | Description |
|---|------------------|--|
| PUDS_STATUS_OK | 0x0 (0) | No error, success. |
| PUDS_STATUS_NOT_INITIALIZED | 0x1 (1) | Not initialized. |
| PUDS_STATUS_ALREADY_INITIALIZED | 0x2 (2) | Already initialized. |
| PUDS_STATUS_NO_MEMORY | 0x3 (3) | Could not obtain memory. |
| PUDS_STATUS_OVERFLOW | 0x4 (4) | Input buffer overflow. |
| PUDS_STATUS_NO_MESSAGE | 0x7 (7) | No message available. |
| PUDS_STATUS_PARAM_INVALID_TYPE | 0x8 (8) | Parameter has an invalid or unexpected type. |
| PUDS_STATUS_PARAM_INVALID_VALUE | 0x9 (9) | Parameter has an invalid value. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | 0xd (13) | PUDS mapping not initialized. |
| PUDS_STATUS_MAPPING_INVALID | 0xe (14) | PUDS mapping parameters are invalid. |
| PUDS_STATUS_MAPPING_ALREADY_INITIALIZED | 0xf (15) | PUDS mapping already defined. |
| PUDS_STATUS_PARAM_BUFFER_TOO_SMALL | 0x10 (16) | Buffer is too small. |
| PUDS_STATUS_QUEUE_TX_FULL | 0x11 (17) | Tx queue is full. |
| PUDS_STATUS_LOCK_TIMEOUT | 0x12 (18) | Failed to get an access to the internal lock. |
| PUDS_STATUS_HANDLE_INVALID | 0x13 (19) | Invalid cantp_handle. |
| PUDS_STATUS_UNKNOWN | 0xff (255) | Unknown/generic error. |
| PUDS_STATUS_FLAG_BUS_LIGHT | 0x100 (256) | PUDS channel is in BUS - LIGHT error state. |
| PUDS_STATUS_FLAG_BUS_HEAVY | 0x200 (512) | PUDS channel is in BUS - HEAVY error state. |
| PUDS_STATUS_FLAG_BUS_WARNING | 0x200 (512) | PUDS channel is in BUS - HEAVY error state. |
| PUDS_STATUS_FLAG_BUS_PASSIVE | 0x400 (1024) | PUDS channel is in error passive state. |
| PUDS_STATUS_FLAG_BUS_OFF | 0x800 (2048) | PUDS channel is in BUS - OFF error state. |
| PUDS_STATUS_FLAG_BUS_ANY | 0xf00 (3840) | Mask for all bus errors. |
| PUDS_STATUS_FLAG_NETWORK_RESULT | 0x2000 (8192) | This flag states if one of the following network errors occurred with the fetched message. |
| PUDS_STATUS_NETWORK_TIMEOUT_A | 0x6000 (24576) | Timeout occurred between 2 frames transmission (sender and receiver side). |
| PUDS_STATUS_NETWORK_TIMEOUT_Bs | 0xa000 (40960) | Sender side timeout while waiting for flow control frame. |
| PUDS_STATUS_NETWORK_TIMEOUT_Cr | 0xe000 (57344) | Receiver side timeout while waiting for consecutive frame. |
| PUDS_STATUS_NETWORK_WRONG_SN | 0x12000 (73728) | Unexpected sequence number. |
| PUDS_STATUS_NETWORK_INVALID_FS | 0x16000 (90112) | Invalid or unknown Flow Status. |
| PUDS_STATUS_NETWORK_UNEXP_PDU | 0x1a000 (106496) | Unexpected protocol data unit. |
| PUDS_STATUS_NETWORK_WFT_OVRN | 0x1e000 (122880) | Reception of flow control WAIT frame that exceeds the maximum counter defined by PUDS_PARAMETER_WFT_MAX. |
| PUDS_STATUS_NETWORK_BUFFER_OVFLW | 0x22000 (139264) | Buffer on the receiver side cannot store the data length (server side only). |
| PUDS_STATUS_NETWORK_ERROR | 0x26000 (155648) | General error. |
| PUDS_STATUS_NETWORK_IGNORED | 0x2a000 (172032) | Message was invalid and ignored. |

| Name | Value | Description |
|--|-------------------------|---|
| PUDS_STATUS_NETWORK_TIMEOUT_Ar | 0x46000 (286720) | Receiver side timeout while transmitting. |
| PUDS_STATUS_NETWORK_TIMEOUT_As | 0x42000 (270336) | Sender side timeout while transmitting. |
| PUDS_STATUS_CAUTION_INPUT_MODIFIED | 0x40000 (262144) | Input was modified by the API. |
| PUDS_STATUS_CAUTION_DLC_MODIFIED | 0x80000 (524288) | DLC value of the input was modified by the API. |
| PUDS_STATUS_CAUTION_DATA_LENGTH_MODIFIED | 0x100000 (1048576) | Data Length value of the input was modified by the API. |
| PUDS_STATUS_CAUTION_FD_FLAG_MODIFIED | 0x200000 (2097152) | FD flags of the input was modified by the API. |
| PUDS_STATUS_CAUTION_RX_QUEUE_FULL | 0x400000 (4194304) | Receive queue is full. |
| PUDS_STATUS_CAUTION_BUFFER_IN_USE | 0x800000 (8388608) | Buffer is used by another thread or API. |
| PUDS_STATUS_FLAG_PCAN_STATUS | 0x80000000 (2147483648) | PCAN error flag, remove flag to get a usable PCAN error/status code (cf. PCANBasic API). |
| PUDS_STATUS_MASK_ERROR | 0x3f (63) | Filter general error. |
| PUDS_STATUS_MASK_BUS | 0x1f00 (7936) | Filter bus error. |
| PUDS_STATUS_MASK_ISOTP_NET | 0x3e000 (253952) | Filter network error. |
| PUDS_STATUS_MASK_INFO | 0xfc0000 (16515072) | Filter extra information. |
| PUDS_STATUS_MASK_PCAN | 0x7FFFFFFF (2147483647) | Filter PCAN error (encapsulated PCAN-Basic status). |
| PUDS_STATUS_FLAG_UDS_ERROR | 0x20000000 (536870912) | PUDS error flag. |
| PUDS_STATUS_MASK_UDS_ERROR | 0xff000000 (4278190080) | Filter PUDS error. |
| PUDS_STATUS_SERVICE_NO_MESSAGE | 0x21000000 (553648128) | UDS, no message available. |
| PUDS_STATUS_SERVICE_TIMEOUT_CONFIRMATION | 0x22000000 (570425344) | Timeout while waiting message confirmation (loopback). |
| PUDS_STATUS_SERVICE_TIMEOUT_RESPONSE | 0x23000000 (587202560) | Timeout while waiting for request message response. |
| PUDS_STATUS_RESET | 0x24000000 (603979776) | UDS reset error. |
| PUDS_STATUS_ERROR_WAIT_FOR_P3_TIMING | 0x25000000 (620756992) | UDS wait for P3 timing error. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | 0x26000000 (637534208) | A UDS request is already pending. A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with UDS_Reset_2013 (see UDS_Reset_2013 on page 651). |
| PUDS_STATUS_SERVICE_TX_ERROR | 0x27000000 (654311424) | An error occurred during the transmission of the UDS request message. |
| PUDS_STATUS_SERVICE_RX_ERROR | 0x28000000 (671088640) | An error occurred during the reception of the UDS response message. |
| PUDS_STATUS_SERVICE_RX_OVERFLOW | 0x29000000 (687865856) | Service received more messages than input buffer expected. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | 0x2a000000 (704643072) | Given message buffer is already allocated, user must release buffer with UDS_MsgFree_2013 or MsgFree_2013 before reusing it. |

Remarks

The `PUDS_STATUS_FLAG_PCAN_STATUS` status is a generic error code that is used to identify PCAN-Basic errors (as PCAN-Basic API is used internally by the PCAN-UDS 2.x API). When a PCAN-Basic error occurs, the API performs a bitwise combination of the `PUDS_STATUS_MASK_PCAN` and the PCAN-Basic (`TPCANStatus`) error.

`UDS_WaitForService_2013` function (or `WaitForService_2013` method) may return exactly `PUDS_STATUS_NETWORK_ERROR`, in this specific case the error occurred in the PCAN-ISO-TP 3.x layer. The network status result of the UDS message will provide extra information (see `uds_msg` on page 21).

See also: `UDS_StatusIsOk_2013` on page 640, `StatusIsOk_2013` on page 203.

3.5.3 uds_parameter

Represents a PUDS parameter or a PUDS value that can be read or set. With some exceptions, a channel must first be initialized before their parameters can be read or set.

Syntax

C/C++

```
typedef enum _uds_parameter {
    PUDS_PARAMETER_API_VERSION = 0x201,
    PUDS_PARAMETER_DEBUG = 0x203,
    PUDS_PARAMETER_RECEIVE_EVENT = 0x204,
    PUDS_PARAMETER_SERVER_ADDRESS = 0x207,
    PUDS_PARAMETER_SESSION_INFO = 0x209,
    PUDS_PARAMETER_TIMEOUT_REQUEST = 0x20A,
    PUDS_PARAMETER_TIMEOUT_RESPONSE = 0x20B,
    PUDS_PARAMETER_AUTOMATIC_TESTER_PRESENT = 0x20C,
    PUDS_PARAMETER_USE_NO_RESPONSE_AUTOMATIC_TESTER_PRESENT = 0x213,
    PUDS_PARAMETER_AUTO_P3_TIMING_MANAGEMENT = 0x20D,
    PUDS_PARAMETER_LISTENED_ADDRESSES = 0x210,
    PUDS_PARAMETER_ADD_LISTENED_ADDRESS = 0x211,
    PUDS_PARAMETER_REMOVE_LISTENED_ADDRESS = 0x212,
    PUDS_PARAMETER_CHANNEL_CONDITION = PCANTP_PARAMETER_CHANNEL_CONDITION,
    PUDS_PARAMETER_CAN_TX_DL = PCANTP_PARAMETER_CAN_TX_DL,
    PUDS_PARAMETER_CAN_DATA_PADDING = PCANTP_PARAMETER_CAN_DATA_PADDING,
    PUDS_PARAMETER_CAN_PADDING_VALUE = PCANTP_PARAMETER_CAN_PADDING_VALUE,
    PUDS_PARAMETER_J1939_PRIORITY = PCANTP_PARAMETER_J1939_PRIORITY,
    PUDS_PARAMETER_BLOCK_SIZE = PCANTP_PARAMETER_BLOCK_SIZE,
    PUDS_PARAMETER_SEPARATION_TIME = PCANTP_PARAMETER_SEPARATION_TIME,
    PUDS_PARAMETER_WFT_MAX = PCANTP_PARAMETER_WFT_MAX,
    PUDS_PARAMETER_ISO_TIMEOUTS = PCANTP_PARAMETER_ISO_TIMEOUTS,
    PUDS_PARAMETER_RESET_HARD = PCANTP_PARAMETER_RESET_HARD,
    PUDS_PARAMETER_HARDWARE_NAME = PCAN_HARDWARE_NAME,
    PUDS_PARAMETER_DEVICE_NUMBER = PCAN_DEVICE_NUMBER,
    PUDS_PARAMETER_CONTROLLER_NUMBER = PCAN_CONTROLLER_NUMBER,
    PUDS_PARAMETER_CHANNEL_FEATURES = PCAN_CHANNEL_FEATURES
} uds_parameter;
```

Pascal OO

```
uds_parameter = (
    PUDS_PARAMETER_API_VERSION = $201,
    PUDS_PARAMETER_DEBUG = $203,
    PUDS_PARAMETER_RECEIVE_EVENT = $204,
    PUDS_PARAMETER_SERVER_ADDRESS = $207,
    PUDS_PARAMETER_SESSION_INFO = $209,
    PUDS_PARAMETER_TIMEOUT_REQUEST = $20A,
    PUDS_PARAMETER_TIMEOUT_RESPONSE = $20B,
    PUDS_PARAMETER_AUTOMATIC_TESTER_PRESENT = $20C,
    PUDS_PARAMETER_USE_NO_RESPONSE_AUTOMATIC_TESTER_PRESENT = $213,
    PUDS_PARAMETER_AUTO_P3_TIMING_MANAGEMENT = $20D,
    PUDS_PARAMETER_LISTENED_ADDRESSES = $210,
    PUDS_PARAMETER_ADD_LISTENED_ADDRESS = $211,
    PUDS_PARAMETER_REMOVE_LISTENED_ADDRESS = $212,
    PUDS_PARAMETER_CHANNEL_CONDITION = UInt32(PCANTP_PARAMETER_CHANNEL_CONDITION),
    PUDS_PARAMETER_CAN_TX_DL = UInt32(PCANTP_PARAMETER_CAN_TX_DL),
    PUDS_PARAMETER_CAN_DATA_PADDING = UInt32(PCANTP_PARAMETER_CAN_DATA_PADDING),
    PUDS_PARAMETER_CAN_PADDING_VALUE = UInt32(PCANTP_PARAMETER_CAN_PADDING_VALUE),
    PUDS_PARAMETER_J1939_PRIORITY = UInt32(PCANTP_PARAMETER_J1939_PRIORITY),
    PUDS_PARAMETER_BLOCK_SIZE = UInt32(PCANTP_PARAMETER_BLOCK_SIZE),
    PUDS_PARAMETER_SEPARATION_TIME = UInt32(PCANTP_PARAMETER_SEPARATION_TIME),
```

```

PUDS_PARAMETER_WFT_MAX = UInt32(PCANTP_PARAMETER_WFT_MAX),
PUDS_PARAMETER_ISO_TIMEOUTS = UInt32(PCANTP_PARAMETER_ISO_TIMEOUTS),
PUDS_PARAMETER_RESET_HARD = UInt32(PCANTP_PARAMETER_RESET_HARD),
PUDS_PARAMETER_HARDWARE_NAME = UInt32(PCAN_HARDWARE_NAME),
PUDS_PARAMETER_DEVICE_NUMBER = UInt32(PCAN_DEVICE_NUMBER),
PUDS_PARAMETER_CONTROLLER_NUMBER = UInt32(PCAN_CONTROLLER_NUMBER),
PUDS_PARAMETER_CHANNEL_FEATURES = UInt32(PCAN_CHANNEL_FEATURES)
);

```

C#

```

public enum uds_parameter : UInt32
{
    PUDS_PARAMETER_API_VERSION = 0x201,
    PUDS_PARAMETER_DEBUG = 0x203,
    PUDS_PARAMETER_RECEIVE_EVENT = 0x204,
    PUDS_PARAMETER_SERVER_ADDRESS = 0x207,
    PUDS_PARAMETER_SESSION_INFO = 0x209,
    PUDS_PARAMETER_TIMEOUT_REQUEST = 0x20A,
    PUDS_PARAMETER_TIMEOUT_RESPONSE = 0x20B,
    PUDS_PARAMETER_AUTOMATIC_TESTER_PRESENT = 0x20C,
    PUDS_PARAMETER_USE_NO_RESPONSE_AUTOMATIC_TESTER_PRESENT = 0x213,
    PUDS_PARAMETER_AUTO_P3_TIMING_MANAGEMENT = 0x20D,
    PUDS_PARAMETER_LISTENED_ADDRESSES = 0x210,
    PUDS_PARAMETER_ADD_LISTENED_ADDRESS = 0x211,
    PUDS_PARAMETER_REMOVE_LISTENED_ADDRESS = 0x212,
    PUDS_PARAMETER_CHANNEL_CONDITION = cantp_parameter.PCANTP_PARAMETER_CHANNEL_CONDITION,
    PUDS_PARAMETER_CAN_TX_DL = cantp_parameter.PCANTP_PARAMETER_CAN_TX_DL,
    PUDS_PARAMETER_CAN_DATA_PADDING = cantp_parameter.PCANTP_PARAMETER_CAN_DATA_PADDING,
    PUDS_PARAMETER_CAN_PADDING_VALUE = cantp_parameter.PCANTP_PARAMETER_CAN_PADDING_VALUE,
    PUDS_PARAMETER_J1939_PRIORITY = cantp_parameter.PCANTP_PARAMETER_J1939_PRIORITY,
    PUDS_PARAMETER_BLOCK_SIZE = cantp_parameter.PCANTP_PARAMETER_BLOCK_SIZE,
    PUDS_PARAMETER_SEPARATION_TIME = cantp_parameter.PCANTP_PARAMETER_SEPARATION_TIME,
    PUDS_PARAMETER_WFT_MAX = cantp_parameter.PCANTP_PARAMETER_WFT_MAX,
    PUDS_PARAMETER_ISO_TIMEOUTS = cantp_parameter.PCANTP_PARAMETER_ISO_TIMEOUTS,
    PUDS_PARAMETER_RESET_HARD = cantp_parameter.PCANTP_PARAMETER_RESET_HARD,
    PUDS_PARAMETER_HARDWARE_NAME = TPCANParameter.PCAN_HARDWARE_NAME,
    PUDS_PARAMETER_DEVICE_NUMBER = TPCANParameter.PCAN_DEVICE_NUMBER,
    PUDS_PARAMETER_CONTROLLER_NUMBER = TPCANParameter.PCAN_CONTROLLER_NUMBER,
    PUDS_PARAMETER_CHANNEL_FEATURES = TPCANParameter.PCAN_CHANNEL_FEATURES
}

```

C++ / CLR

```

public enum uds_parameter : UInt32
{
    PUDS_PARAMETER_API_VERSION = 0x201,
    PUDS_PARAMETER_DEBUG = 0x203,
    PUDS_PARAMETER_RECEIVE_EVENT = 0x204,
    PUDS_PARAMETER_SERVER_ADDRESS = 0x207,
    PUDS_PARAMETER_SESSION_INFO = 0x209,
    PUDS_PARAMETER_TIMEOUT_REQUEST = 0x20A,
    PUDS_PARAMETER_TIMEOUT_RESPONSE = 0x20B,
    PUDS_PARAMETER_AUTOMATIC_TESTER_PRESENT = 0x20C,
    PUDS_PARAMETER_USE_NO_RESPONSE_AUTOMATIC_TESTER_PRESENT = 0x213,
    PUDS_PARAMETER_AUTO_P3_TIMING_MANAGEMENT = 0x20D,
    PUDS_PARAMETER_LISTENED_ADDRESSES = 0x210,
    PUDS_PARAMETER_ADD_LISTENED_ADDRESS = 0x211,
    PUDS_PARAMETER_REMOVE_LISTENED_ADDRESS = 0x212,
    PUDS_PARAMETER_CHANNEL_CONDITION = cantp_parameter::PCANTP_PARAMETER_CHANNEL_CONDITION,
    PUDS_PARAMETER_CAN_TX_DL = cantp_parameter::PCANTP_PARAMETER_CAN_TX_DL,
    PUDS_PARAMETER_CAN_DATA_PADDING = cantp_parameter::PCANTP_PARAMETER_CAN_DATA_PADDING,

```

```

PUDS_PARAMETER_CAN_PADDING_VALUE = cantp_parameter::PCANTP_PARAMETER_CAN_PADDING_VALUE,
PUDS_PARAMETER_J1939_PRIORITY = cantp_parameter::PCANTP_PARAMETER_J1939_PRIORITY,
PUDS_PARAMETER_BLOCK_SIZE = cantp_parameter::PCANTP_PARAMETER_BLOCK_SIZE,
PUDS_PARAMETER_SEPARATION_TIME = cantp_parameter::PCANTP_PARAMETER_SEPARATION_TIME,
PUDS_PARAMETER_WFT_MAX = cantp_parameter::PCANTP_PARAMETER_WFT_MAX,
PUDS_PARAMETER_ISO_TIMEOUTS = cantp_parameter::PCANTP_PARAMETER_ISO_TIMEOUTS,
PUDS_PARAMETER_RESET_HARD = cantp_parameter::PCANTP_PARAMETER_RESET_HARD,
PUDS_PARAMETER_HARDWARE_NAME = (UInt32)TPCANParameter::PCAN_HARDWARE_NAME,
PUDS_PARAMETER_DEVICE_NUMBER = (UInt32)TPCANParameter::PCAN_DEVICE_NUMBER,
PUDS_PARAMETER_CONTROLLER_NUMBER = (UInt32)TPCANParameter::PCAN_CONTROLLER_NUMBER,
PUDS_PARAMETER_CHANNEL_FEATURES = (UInt32)TPCANParameter::PCAN_CHANNEL_FEATURES
};

```

Visual Basic

```

Public Enum uds_parameter As UInt32
    PUDS_PARAMETER_API_VERSION = &H201
    PUDS_PARAMETER_DEBUG = &H203
    PUDS_PARAMETER_RECEIVE_EVENT = &H204
    PUDS_PARAMETER_SERVER_ADDRESS = &H207
    PUDS_PARAMETER_SESSION_INFO = &H209
    PUDS_PARAMETER_TIMEOUT_REQUEST = &H20A
    PUDS_PARAMETER_TIMEOUT_RESPONSE = &H20B
    PUDS_PARAMETER_AUTOMATIC_TESTER_PRESENT = &H20C
    PUDS_PARAMETER_USE_NO_RESPONSE_AUTOMATIC_TESTER_PRESENT = &H213
    PUDS_PARAMETER_AUTO_P3_TIMING_MANAGEMENT = &H20D
    PUDS_PARAMETER_LISTENED_ADDRESSES = &H210
    PUDS_PARAMETER_ADD_LISTENED_ADDRESS = &H211
    PUDS_PARAMETER_REMOVE_LISTENED_ADDRESS = &H212
    PUDS_PARAMETER_CHANNEL_CONDITION = cantp_parameter.PCANTP_PARAMETER_CHANNEL_CONDITION
    PUDS_PARAMETER_CAN_TX_DL = cantp_parameter.PCANTP_PARAMETER_CAN_TX_DL
    PUDS_PARAMETER_CAN_DATA_PADDING = cantp_parameter.PCANTP_PARAMETER_CAN_DATA_PADDING
    PUDS_PARAMETER_CAN_PADDING_VALUE = cantp_parameter.PCANTP_PARAMETER_CAN_PADDING_VALUE
    PUDS_PARAMETER_J1939_PRIORITY = cantp_parameter.PCANTP_PARAMETER_J1939_PRIORITY
    PUDS_PARAMETER_BLOCK_SIZE = cantp_parameter.PCANTP_PARAMETER_BLOCK_SIZE
    PUDS_PARAMETER_SEPARATION_TIME = cantp_parameter.PCANTP_PARAMETER_SEPARATION_TIME
    PUDS_PARAMETER_WFT_MAX = cantp_parameter.PCANTP_PARAMETER_WFT_MAX
    PUDS_PARAMETER_ISO_TIMEOUTS = cantp_parameter.PCANTP_PARAMETER_ISO_TIMEOUTS
    PUDS_PARAMETER_RESET_HARD = cantp_parameter.PCANTP_PARAMETER_RESET_HARD
    PUDS_PARAMETER_HARDWARE_NAME = TPCANParameter.PCAN_HARDWARE_NAME
    PUDS_PARAMETER_DEVICE_NUMBER = TPCANParameter.PCAN_DEVICE_NUMBER
    PUDS_PARAMETER_CONTROLLER_NUMBER = TPCANParameter.PCAN_CONTROLLER_NUMBER
    PUDS_PARAMETER_CHANNEL_FEATURES = TPCANParameter.PCAN_CHANNEL_FEATURES
End Enum

```

values

| Name | Value | Data type | Description |
|-------------------------------|-------|-----------------|--|
| PUDS_PARAMETER_API_VERSION | 0x201 | String | API version of the PCAN-UDS API. |
| PUDS_PARAMETER_DEBUG | 0x203 | Byte | Debug mode. |
| PUDS_PARAMETER_RECEIVE_EVENT | 0x204 | Pointer | Defines PUDS receive-event handler, require a pointer to an event HANDLE. |
| PUDS_PARAMETER_SERVER_ADDRESS | 0x207 | UInt16 | Physical address of the server (ECU) |
| PUDS_PARAMETER_SESSION_INFO | 0x209 | uds_sessioninfo | UDS current server (ECU) session information (see uds_sessioninfo on page 22). |

| Name | Value | Data type | Description |
|---|-------|-----------------|--|
| PUDS_PARAMETER_TIMEOUT_REQUEST | 0x20A | UInt32 | Maximum time in milliseconds allowed by the client to transmit a request. |
| PUDS_PARAMETER_TIMEOUT_RESPONSE | 0x20B | UInt32 | Maximum time in milliseconds allowed by the client to receive a response. |
| PUDS_PARAMETER_AUTOMATIC_TESTER_PRESENT | 0x20C | Boolean | Automatic tester present (default value: true). |
| PUDS_PARAMETER_USE_NO_RESPONSE_AUTOMATIC_TESTER_PRESENT | 0x213 | Boolean | Use no response flag for automatic tester present (default value: true) |
| PUDS_PARAMETER_AUTO_P3_TIMING_MANAGEMENT | 0x20D | Boolean | Waits for P3 timing (default value: true). |
| PUDS_PARAMETER_LISTENED_ADDRESSES | 0x210 | Array of UInt16 | Lists of physical addresses to listen to. |
| PUDS_PARAMETER_ADD_LISTENED_ADDRESS | 0x211 | UInt16 | Adds a listening address to the list of physical addresses to listen to. |
| PUDS_PARAMETER_REMOVE_LISTENED_ADDRESS | 0x212 | UInt16 | Removes a listening address from the list of physical addresses to listen to. |
| PUDS_PARAMETER_CHANNEL_CONDITION | 0x102 | Byte | PUDS channel condition. |
| PUDS_PARAMETER_CAN_TX_DL | 0x106 | Byte | Data stating the default DLC to use when transmitting messages with CAN FD. |
| PUDS_PARAMETER_CAN_DATA_PADDING | 0x107 | Byte | Defines if CAN frame DLC uses padding or not. |
| PUDS_PARAMETER_CAN_PADDING_VALUE | 0x108 | Byte | Value used when CAN Data padding is enabled. |
| PUDS_PARAMETER_J1939_PRIORITY | 0x10A | Byte | Defines the default priority value for normal fixed, mixed, and enhanced addressing (default=6). |
| PUDS_PARAMETER_BLOCK_SIZE | 0x10C | Byte | Block size (BS) parameter. |
| PUDS_PARAMETER_SEPARATION_TIME | 0x10E | Byte | Separation Time (Stmin) parameter. |
| PUDS_PARAMETER_WFT_MAX | 0x110 | UInt32 | "N_WFTmax" parameter. |
| PUDS_PARAMETER_ISO_TIMEOUTS | 0x116 | Byte | Predefined ISO values for timeouts. |
| PUDS_PARAMETER_RESET_HARD | 0x11F | Byte | Resets the CAN controller and clears Rx/Tx queues without uninitializing mapping and defined settings. |
| PUDS_PARAMETER_HARDWARE_NAME | 0x0E | String | PCAN hardware name parameter. |
| PUDS_PARAMETER_DEVICE_NUMBER | 0x01 | UInt32 | PCAN-USB device number parameter. |
| PUDS_PARAMETER_CONTROLLER_NUMBER | 0x10 | UInt32 | CAN-Controller number of a PCAN-Channel. |
| PUDS_PARAMETER_CHANNEL_FEATURES | 0x16 | UInt32 | Capabilities of a PCAN device (FEATURE_***). |

Detailed Parameters Characteristics

PUDS_PARAMETER_API_VERSION

Access: **R**

Description: This parameter is used to get information about the PCAN-UDS API implementation version.

Possible Values: The value is a null-terminated string indication the version number of the API implementation. The returned text has the following form: x,x,x,x for major, minor, release and build. It represents the binary version of the API, within two 32-bit integers, defined by four 16-bit integers. The length of this text value will have a maximum length of 24 bytes, 5 bytes for represent each 16-bit value, three separator characters (, or .) and the null-termination.

Default Value: NA.








PCAN-Device: NA. Any PCAN device can be used, including the `PCANTP_HANDLE_NONEBUS` channel.

PUDS_PARAMETER_DEBUG

Access: **R W**

Description: This parameter is used to control debug mode. If enabled, any received or transmitted CAN frames will be logged in PCANBasic log file (default filename is PCANBasic.log located inside the current directory).

Possible Values:

| | Type | Constant | Value | Description |
|---|------|----------------------------|-------|---|
|  | Byte | PUDS_DEBUG_LVL_NONE | 0x00 | No debug messages are being generated. |
|  | Byte | PUDS_DEBUG_LVL_ERROR | 0xF1 | Enable debug messages (only errors) |
|  | Byte | PUDS_DEBUG_LVL_WARNING | 0xF2 | Enable debug messages (only warnings, errors) |
|  | Byte | PUDS_DEBUG_LVL_INFORMATION | 0xF3 | Enable debug messages (only informations, warnings, errors) |
|  | Byte | PUDS_DEBUG_LVL_NOTICE | 0xF4 | Enable debug messages (only notices, informations, warnings, errors) |
|  | Byte | PUDS_DEBUG_LVL_DEBUG | 0xF5 | Enable debug messages (only debug, notices, informations, warnings, errors) |
|  | Byte | PUDS_DEBUG_LVL_TRACE | 0xF6 | Enable all debug messages |

Default Value: `PUDS_DEBUG_LVL_NONE` (no debug messages are being generated).

PCAN-Device: All PCAN devices (excluding `PCANTP_HANDLE_NONEBUS` channel).

PUDS_PARAMETER_RECEIVE_EVENT

Access: **R W**

Description: This parameter is used to let the PCAN-UDS 2.x API notify an application when UDS messages are available to be read. In this form, message processing tasks of an application can react faster and make a more efficient use of the processor time.

Possible Values: This value has to be a handle for an event object returned by the Windows API function `CreateEvent` or the value 0 (`IntPtr.Zero` in a managed environment). When setting this parameter, the value of 0 resets the parameter in the PCAN-UDS 2.x API. Reading a value of 0 indicates that no event handle is set. For more information about reading with events, please refer to the topic Using Events on page 777.



Note: .NET environment should use the prototype `SetValue_2013`, as in the following C# example:

```
System.Threading.AutoResetEvent receive_event = new System.Threading.AutoResetEvent(false);

UInt32 ibuf_event =
    Convert.ToInt32(receive_event.SafeWaitHandle.DangerousGetHandle().ToInt32());
UDSApi.SetValue_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    uds_parameter.PUDS_PARAMETER_RECEIVE_EVENT, ref ibuf_event, sizeof(UInt32));
```

Default Value: Disabled (0).

PCAN-Device: All PCAN devices (excluding `PCANTP_HANDLE_NONEBUS` channel).

PUDS_PARAMETER_SERVER_ADDRESS

Access: **R W**

Description: This value is used to define the address of the server. Defining this parameter is required as it allows configuration of the underlying ISO-TP 3.0 API. By default the address is set to the standard address of external test equipment (0xF1, see `uds_address` on page 56). Although most UDS addresses are 1 byte data, the parameter requires a 2 bytes value in order to be compatible with ISO 15765-3:2004 addresses (see `PUDS_SERVER_ADDR_MASK_ENHANCED_ISO_15765_3`, 0x7FF).

Note that with ISO 14229-1:2013 enhanced addresses are considered deprecated, yet if you need to define an ISO 15765-3:2004 address, add the flag `PUDS_SERVER_ADDR_FLAG_ENHANCED_ISO_15765_3` (0x1000) to the address (using a bitwise OR operation, i.e.: `(PUDS_SERVER_ADDR_FLAG_ENHANCED_ISO_15765_3 | 0x123)`).

Possible Values: 0x00 to 0xFF, or 0x1000 to 0x17FF for enhanced ISO 14229-1:2013 addresses (see predefined `uds_address` values on page 56).

Default Value: `PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT` (0xF1).

PCAN-Device: All PCAN devices (excluding `PCANTP_HANDLE_NONEBUS` channel).

PUDS_PARAMETER_SESSION_INFO

Access: **R W**

Description: This parameter is used to read the current diagnostic session information. The diagnostic session information is a value which is internally updated when the UDS service `UDS_SvcDiagnosticSessionControl_2013` is invoked: it keeps track of the active session and its associated timeouts. If a non-default diagnostic session is active, a keep-alive mechanism (using physically addressed TesterPresent service requests with the “suppress positive response message” flag) is automatically activated according to UDS standard (see below `PUDS_PARAMETER_AUTOMATIC_TESTER_PRESENT` to enable/disable this mechanism). This parameter can also be used to override the current session information (for example, some custom ECU may be in a non default session after reset, in this case the current session information must be overridden).

Possible Values: any `uds_sessioninfo` structure (see `uds_sessioninfo` on page 22).

Default Value: automatically updated when the response of the service `UDS_SvcDiagnosticSessionControl_2013` is processed (with the `UDS_WaitForService_2013` or `UDS_WaitForServiceFunctional_2013` functions).

PCAN-Device: All PCAN devices (excluding `PCANTP_HANDLE_NONEBUS` channel).

See also: `uds_sessioninfo` on page 22, `UDS_SvcDiagnosticSessionControl_2013` function on page 661, `SvcDiagnosticSessionControl_2013` class method on page 258.

PUDS_PARAMETER_TIMEOUT_REQUEST**Access:** **R W**

Description: This value defines the maximum waiting time in milliseconds to receive a confirmation of a transmission (or request loopback). It is used in PCAN-UDS 2.x API utility functions like `UDS_WaitForService_2013` or method like `WaitForService_2013`.

Possible Values: 0x00 (unlimited) to 0xFFFFFFFF.

Default Value: `PUDS_TIMEOUT_REQUEST` (10000 milliseconds).

PCAN-Device: All PCAN devices (excluding `PCANTP_HANDLE_NONEBUS` channel).

PUDS_PARAMETER_TIMEOUT_RESPONSE**Access:** **R W**

Description: This value defines the maximum waiting time in milliseconds to receive a response indication. Note that the exact timeout value is the sum of this parameter and the timeout defined in the active diagnostic session. It is used in PCAN-UDS 2.x API utility functions like `UDS_WaitForService_2013` or methods like `WaitForService_2013`.

Possible Values: 0x00 (unlimited) to 0xFFFFFFFF.

Default Value: `PUDS_TIMEOUT_RESPONSE` (10000 milliseconds).

PCAN-Device: All PCAN devices (excluding `PCANTP_HANDLE_NONEBUS` channel).

PUDS_PARAMETER_AUTOMATIC_TESTER_PRESENT**Access:** **R W**

Description: Activates or not the automatic UDS Tester Present request that should occur when a non-default UDS session is set.

Possible Values: true (1) or false (0).

Default Value: true (activated).

PCAN-Device: All PCAN devices (excluding `PCANTP_HANDLE_NONEBUS` channel).

PUDS_PARAMETER_USE_NO_RESPONSE_AUTOMATIC_TESTER_PRESENT**Access:** **R W**

Description: Activates or not the no response flag for automatic UDS Tester Present requests.

Possible Values: true (1) or false (0).

Default Value: true (activated).

PCAN-Device: All PCAN devices (excluding `PCANTP_HANDLE_NONEBUS` channel).

PUDS_PARAMETER_AUTO_P3_TIMING_MANAGEMENT**Access:** **R W**

Description: Activates or not automatic P3 timing management (P3 client/server is the time to wait before sending a new UDS request to the same ECU). If activated, required waiting timing between consecutive

requests (functional or physical request without response) will be included in the call to `UDS_Write_2013` function. If not `UDS_Write_2013` will return the status `PUDS_STATUS_ERROR_WAIT_FOR_P3_TIMING`.

Possible Values: true (1) or false (0).

Default Value: true (activated).

PCAN-Device: All PCAN devices (excluding `PCANTP_HANDLE_NONEBUS` channel).

See also: ISO-14229-2_2013 §8.3 Minimum time between client request messages, p.36.

PUDS_PARAMETER_LISTENED_ADDRESSES

Access: 

Description: Gets the list of physical addresses to listen to. The length of the array must be specified in the `buffer_size` parameter of the `UDS_GetValue_2013` function or `GetValue_2013` method. If a UDS message is received and its target address does not match an item in the list of physical addresses, then this message is discarded.

Possible Values: 0x00 to 0xFF (see predefined `uds_address` values on page 56), or 0x1000 to 0x17FF for enhanced ISO 14229-1:2013 addresses (with `PUDS_SERVER_ADDR_FLAG_ENHANCED_ISO_15765_3` flag).

PCAN-Device: All PCAN devices (excluding `PCANTP_HANDLE_NONEBUS` channel).

PUDS_PARAMETER_ADD_LISTENED_ADDRESS

Access: 

Description: Adds a listening address to the list of physical addresses to listen to. If a UDS message is received and its target address does not match an item in the list of physical addresses, then this message is discarded. **Note:** if a call to `UDS_Write_2013` function is requested with a source address that is not in this list, it will be automatically added to this list.

Possible Values: 0x00 to 0xFF (see predefined `uds_address` values on page 56), or 0x1000 to 0x17FF for enhanced ISO 14229-1:2013 addresses (with `PUDS_SERVER_ADDR_FLAG_ENHANCED_ISO_15765_3` flag).

PCAN-Device: All PCAN devices (excluding `PCANTP_HANDLE_NONEBUS` channel).

PUDS_PARAMETER_REMOVE_LISTENED_ADDRESS

Access: 

Description: Removes a listened address from the list of physical addresses to listen to.

Possible Values: 0x00 to 0xFF (see predefined `uds_address` values on page 56), or 0x1000 to 0x17FF for enhanced ISO 14229-1:2013 addresses (with `PUDS_SERVER_ADDR_FLAG_ENHANCED_ISO_15765_3` flag).




PCAN-Device: All PCAN devices (excluding `PCANTP_HANDLE_NONEBUS` channel).

PUDS_PARAMETER_CHANNEL_CONDITION

Access: 


Description: This parameter is used to check and detect available PCAN hardware on a computer, even before trying to connect any of them. This is useful when an application wants the user to select which hardware should be using in a communication session.

Possible Values: This parameter can have one of these values: `PUDS_CHANNEL_UNAVAILABLE`, `PUDS_CHANNEL_AVAILABLE` and `PUDS_CHANNEL_OCCUPIED`.

| | Type | Constant | Value | Description |
|---|------|--------------------------|-------|--|
|  | Byte | PUDS_CHANNEL_UNAVAILABLE | 0 | The PUDS channel handle is illegal, or its associated hardware is not available. |
|  | Byte | PUDS_CHANNEL_AVAILABLE | 1 | The PUDS channel handle is valid to connect/initialize. Furthermore, for Plug and Play hardware, this means that the hardware is plugged in. |
|  | Byte | PUDS_CHANNEL_OCCUPIED | 2 | The PUDS channel handle is valid and is currently being used. |

Default Value: NA.

PCAN-Device: All PCAN devices (excluding `PCANTP_HANDLE_NONEBUS` channel).

 **Note:** It is not needed to have a PUDS channel initialized before asking for its condition.

PUDS_PARAMETER_CAN_TX_DL

Access:  

Description: This parameter is used to define the default Data Length Code (DLC) used when transmitting ISO-TP messages CAN FD enabled: the fragmented CAN FD frames composing the full CAN ISO-TP message will have at most a length corresponding to that DLC (It depends if the data fit in a lower DLC value). Note that member `can_tx_dlc` in `uds_mapping` or `uds_msgconfig` or message's `can_info.dlc` can be used to override this parameter locally (see `uds_mapping` on page 25, `uds_msgconfig` on page 27, `cantp_can_info` on page 124).

Possible values: 8 (0x08) to 15 (0x0F).

Default value: 8 (0x08).



PCAN-Device: All PCAN devices (excluding `PCANTP_HANDLE_NONEBUS` channel).

PUDS_PARAMETER_CAN_DATA_PADDING

Access:  

Description: This parameter is used to define if the API should use CAN data optimization or CAN data padding: the first case will optimize the CAN DLC to avoid sending unnecessary data, on the other hand with CAN data padding the API will always send CAN frames with a DLC of 8 and pads the data with the padding value.

Possible values: `PUDS_CAN_DATA_PADDING_NONE` disables data padding (enabling CAN data optimization) and `PUDS_CAN_DATA_PADDING_ON` (enabling data padding).

| | Type | Constant | Value | Description |
|---|------|----------------------------|-------|---|
|  | Byte | PUDS_CAN_DATA_PADDING_NONE | 0x00 | CAN frame data optimization is enabled. |
|  | Byte | PUDS_CAN_DATA_PADDING_ON | 0x01 | CAN frame data optimization is disabled: CAN data length is always 8 and data is padded with zeros. |

Default value: `PUDS_CAN_DATA_PADDING_ON` since ECUs that do not support CAN data optimization may not respond to UDS/CAN-TP messages.

PCAN-Device: All PCAN devices (excluding the `PCANTP_HANDLE_NONEBUS` channel).

PUDS_PARAMETER_CAN_PADDING_VALUE

Access:  

Description: This parameter is used to define the value (or pattern) for CAN data padding when it is enabled.

Possible values: Any value from 0x00 to 0xFF.

Default value: 0x55 (`PUDS_CAN_DATA_PADDING_VALUE`).

PCAN-Device: All PCAN devices (excluding `PCANTP_HANDLE_NONEBUS` channel).

PUDS_PARAMETER_J1939_PRIORITY

Access: **R W**

Description: This parameter is used to define the default priority for messages compliant with SAE J1939 data link layer (i.e. 29-bit CAN identifier messages with normal fixed, mixed, or enhanced addressing).

Possible Values: Any value from 0x00 to 0x07.

Default Value: 0x06 (`PCANTP_J1939_PRIORITY_DEFAULT`).

PCAN-Device: All PCAN devices (excluding `PCANTP_HANDLE_NONEBUS` channel).

PUDS_PARAMETER_BLOCK_SIZE

Access: **R W**

Description: This value is used to set the BlockSize (BS) parameter defined in the ISO-TP standard: it indicates to the sender the maximum number of consecutive frames that can be received without an intermediate FlowControl frame from the receiving network entity. A value of 0 indicates that no limit is set, and the sending network layer entity shall send all remaining consecutive frames.

Possible Values: 0x00 (unlimited) to 0xFF.

Default Value: 10.


PCAN-Device: All PCAN devices (excluding `PCANTP_HANDLE_NONEBUS` channel).

PUDS_PARAMETER_SEPARATION_TIME

Access: **R W**

Description: This value is used to set the Separation Time (STmin) parameter defined in the ISO-TP standard: it indicates the minimum time the sender is to wait between the transmissions of two Consecutive Frames.

Possible values: 0x00 to 0x7F (range from 0 to 127 milliseconds) and 0xF1 to 0xF9 (range from 100 to 900 µs).

 **Note:** Values between 0xF1 to 0xF3 should define a minimum time of 100 to 300 µs, but in practice the time to transmit effectively a frame takes about 300 µs (which is to send the message to the CAN controller and to assert that the message is physically emitted on the CAN bus). Other values than the ones stated above are ISO reserved.

Default value: 10ms.



PCAN-Device: All PCAN devices (excluding `PCANTP_HANDLE_NONEBUS` channel).

PUDS_PARAMETER_WFT_MAX

Access: **R W**


Description: This parameter is used to set the maximum number of FlowControl Wait frame transmission (N_WFTmax) defined in the ISO-TP standard: it indicates how many FlowControl Wait frames with the wait status can be transmitted by a receiver in a row.

Possible value: Any positive number.

| | Type | Constant | Value | Description |
|---|-------|--------------------------|-------|---|
|  | Int32 | PCANTP_WFT_MAX_UNLIMITED | 0x00 | Disables checks for ISO-TP WaitForFrames overrun when receiving a FlowControl frames (PUDS_STATUS_NETWORK_WFT_OVRN error will never occur). |
|  | Int32 | PCANTP_WFT_MAX_DEFAULT | 0x10 | The default value used by the API: if the number of consecutive FlowControl frame with the wait status exceeds this value, a PUDS_STATUS_NETWORK_WFT_OVRN error will occur. |

Default value: `PCANTP_WFT_MAX_DEFAULT` (0x10).

PCAN-Device: NA. Any PCAN device can be used, including the `PCANTP_HANDLE_NONEBUS` channel.



 **Note:** Also, this parameter is set globally, channels will use the value sets when they are initialized, so it is possible to define different values of N_WFTmax on separate channels. Consequently, once a channel is initialized, changing the WFTmax parameter will not affect that channel.

PUDS_PARAMETER_ISO_TIMEOUTS

Access:  

Description: Sets predefined ISO values for timeouts.

Possible values:

| | Type | Constant | Value | Description |
|--|------|-----------------------------|-------|---|
|  | Byte | PCANTP_ISO_TIMEOUTS_15765_2 | 0 | Sets timeouts according to ISO-15765-2. |
|  | Byte | PCANTP_ISO_TIMEOUTS_15765_4 | 1 | Sets timeouts according to ISO-15765-4 (OBDII). |

Default value: 0 (`PCANTP_ISO_TIMEOUTS_15765_2`).

PCAN-Device: All PCAN devices (excluding `PCANTP_HANDLE_NONEBUS` channel).

PUDS_PARAMETER_RESET_HARD

Access: 

Description: Resets the CAN controller and clears internal reception and transmission queues. This parameter provides a way to uninitialized then initialize the CAN channel without losing any configured mappings and PUDS/PCANTP related settings.

Possible values: 1.

Default value: NA.

PCAN-Device: All PCAN devices (excluding `PCANTP_HANDLE_NONEBUS` channel).

PUDS_PARAMETER_HARDWARE_NAME

Access: 

Description: This parameter is used to retrieve the name of the hardware represented by a PUDS channel. This is useful when an application wants to differentiate between several models of the same device, e.g. a PCAN-USB and a PCAN-USB Pro.

Possible values: The value is a null-terminated string which contains the name of the hardware specified by the given PUDS channel. The length of this text will have a maximum length of 32 bytes (null termination included).

Default value: N/A.

PCAN-Device: All PCAN devices (excluding `PCANTP_HANDLE_NONEBUS` channel).

PUDS_PARAMETER_DEVICE_NUMBER**Access:** R W

Description: This parameter is used on PCAN hardware to distinguish between 2 (or more) devices of the same type connected to the same computer. This value is persistent, i.e. the identifier will not be lost after disconnecting and connecting again a device.

Possible values: According to the firmware version, this value can be a number in the range [1..255] or [1..4294967295]. If the firmware has a resolution of one byte and the specified value is bigger, the value will be truncated.

Default value: If this parameter was never set before, the value is the maximum value possible for the used resolution. For 8-bits: 255 (0xFF), for 32 bits: 429496729 (0xFFFFFFFF).

PCAN-Device: All PCAN devices (excluding `PCANTP_HANDLE_NONEBUS` channel).

PUDS_PARAMETER_CONTROLLER_NUMBER**Access:** R

Description: This parameter is a zero-based index used to identify the CAN controllers built in a hardware. This parameter is useful when it is needed to communicate with a specific physical channel on a multichannel CAN Hardware, e.g. "0" or "1" on a PCAN-USB Pro device.

Possible values: A number in the range [0..n-1], where n is the number of physical channels on the device being used.




Default value: N/A.

PCAN-Device: All PCAN devices (excluding `PCANTP_HANDLE_NONEBUS` channel).

PUDS_PARAMETER_CHANNEL_FEATURES**Access:** R

Description: This value is used to read the particularities of a PUDS channel.

Possible values: The value can be one of the following values or a combination of them:

| | Type | Constant | Value | Description |
|---|--------|-----------------------|-------|---|
|  | UInt32 | FEATURE_FD_CAPABLE | 1 | This value indicates that the hardware represented by a PUDS channel is FD capable (it supports flexible data rate). |
|  | UInt32 | FEATURE_DELAY_CAPABLE | 2 | This value indicates that the hardware represented by a PUDS channel allows the configuration of a delay between sending frames. |
|  | UInt32 | FEATURE_IO_CAPABLE | 4 | This value indicates that the hardware represented by a PUDS channel supports I/O functionality for electronic circuits (USB-Chip devices). |

Default value: A value of 0, indicating "no special features".

PCAN-Device: All PCAN devices (excluding `PCANTP_HANDLE_NONEBUS` channel).

3.5.4 uds_service

Represents a UDS service identifier. According to the programming language, this type can be a group of defined values or an enumeration.

Syntax

C/C++

```
#define PUDS_SI_DiagnosticSessionControl    0x10
#define PUDS_SI_ECUReset                    0x11
#define PUDS_SI_SecurityAccess              0x27
#define PUDS_SI_CommunicationControl        0x28
#define PUDS_SI_TesterPresent               0x3E
#define PUDS_SI_AccessTimingParameter       0x83
#define PUDS_SI_SecuredDataTransmission    0x84
#define PUDS_SI_ControlDTCSetting           0x85
#define PUDS_SI_ResponseOnEvent             0x86
#define PUDS_SI_LinkControl                 0x87
#define PUDS_SI_ReadDataByIdentifier        0x22
#define PUDS_SI_ReadMemoryByAddress         0x23
#define PUDS_SI_ReadScalingDataByIdentifier 0x24
#define PUDS_SI_ReadDataByPeriodicIdentifier 0x2A
#define PUDS_SI_DynamicallyDefineDataIdentifier 0x2C
#define PUDS_SI_WriteDataByIdentifier       0x2E
#define PUDS_SI_WriteMemoryByAddress        0x3D
#define PUDS_SI_ClearDiagnosticInformation   0x14
#define PUDS_SI_ReadDTCInformation          0x19
#define PUDS_SI_InputOutputControlByIdentifier 0x2F
#define PUDS_SI_RoutineControl              0x31
#define PUDS_SI_RequestDownload             0x34
#define PUDS_SI_RequestUpload               0x35
#define PUDS_SI_TransferData                0x36
#define PUDS_SI_RequestTransferExit         0x37
#define PUDS_SI_RequestFileTransfer         0x38
#define PUDS_SI_Authentication              0x29
#define PUDS_NR_SI                          0x7F
```

Pascal OO

```
uds_service = (
  PUDS_SERVICE_SI_DiagnosticSessionControl = $10,
  PUDS_SERVICE_SI_ECUReset = $11,
  PUDS_SERVICE_SI_SecurityAccess = $27,
  PUDS_SERVICE_SI_CommunicationControl = $28,
  PUDS_SERVICE_SI_TesterPresent = $3E,
  PUDS_SERVICE_SI_AccessTimingParameter = $83,
  PUDS_SERVICE_SI_SecuredDataTransmission = $84,
  PUDS_SERVICE_SI_ControlDTCSetting = $85,
  PUDS_SERVICE_SI_ResponseOnEvent = $86,
  PUDS_SERVICE_SI_LinkControl = $87,
  PUDS_SERVICE_SI_ReadDataByIdentifier = $22,
  PUDS_SERVICE_SI_ReadMemoryByAddress = $23,
  PUDS_SERVICE_SI_ReadScalingDataByIdentifier = $24,
  PUDS_SERVICE_SI_ReadDataByPeriodicIdentifier = $2A,
  PUDS_SERVICE_SI_DynamicallyDefineDataIdentifier = $2C,
  PUDS_SERVICE_SI_WriteDataByIdentifier = $2E,
  PUDS_SERVICE_SI_WriteMemoryByAddress = $3D,
  PUDS_SERVICE_SI_ClearDiagnosticInformation = $14,
  PUDS_SERVICE_SI_ReadDTCInformation = $19,
  PUDS_SERVICE_SI_InputOutputControlByIdentifier = $2F,
  PUDS_SERVICE_SI_RoutineControl = $31,
  PUDS_SERVICE_SI_RequestDownload = $34,
```

```

PUDS_SERVICE_SI_RequestUpload = $35,
PUDS_SERVICE_SI_TransferData = $36,
PUDS_SERVICE_SI_RequestTransferExit = $37,
PUDS_SERVICE_SI_RequestFileTransfer = $38,
PUDS_SERVICE_SI_Authentication = $29,
PUDS_SERVICE_NR_SI = $7F
);

```

C#

```

public enum uds_service : Byte
{
    PUDS_SERVICE_SI_DiagnosticSessionControl = 0x10,
    PUDS_SERVICE_SI_ECUReset = 0x11,
    PUDS_SERVICE_SI_SecurityAccess = 0x27,
    PUDS_SERVICE_SI_CommunicationControl = 0x28,
    PUDS_SERVICE_SI_TesterPresent = 0x3E,
    PUDS_SERVICE_SI_AccessTimingParameter = 0x83,
    PUDS_SERVICE_SI_SecuredDataTransmission = 0x84,
    PUDS_SERVICE_SI_ControlDTCSetting = 0x85,
    PUDS_SERVICE_SI_ResponseOnEvent = 0x86,
    PUDS_SERVICE_SI_LinkControl = 0x87,
    PUDS_SERVICE_SI_ReadDataByIdentifier = 0x22,
    PUDS_SERVICE_SI_ReadMemoryByAddress = 0x23,
    PUDS_SERVICE_SI_ReadScalingDataByIdentifier = 0x24,
    PUDS_SERVICE_SI_ReadDataByPeriodicIdentifier = 0x2A,
    PUDS_SERVICE_SI_DynamicallyDefineDataIdentifier = 0x2C,
    PUDS_SERVICE_SI_WriteDataByIdentifier = 0x2E,
    PUDS_SERVICE_SI_WriteMemoryByAddress = 0x3D,
    PUDS_SERVICE_SI_ClearDiagnosticInformation = 0x14,
    PUDS_SERVICE_SI_ReadDTCInformation = 0x19,
    PUDS_SERVICE_SI_InputOutputControlByIdentifier = 0x2F,
    PUDS_SERVICE_SI_RoutineControl = 0x31,
    PUDS_SERVICE_SI_RequestDownload = 0x34,
    PUDS_SERVICE_SI_RequestUpload = 0x35,
    PUDS_SERVICE_SI_TransferData = 0x36,
    PUDS_SERVICE_SI_RequestTransferExit = 0x37,
    PUDS_SERVICE_SI_RequestFileTransfer = 0x38,
    PUDS_SERVICE_SI_Authentication = 0x29,
    PUDS_SERVICE_NR_SI = 0x7F,
}

```

C++ / CLR

```

public enum uds_service : Byte
{
    PUDS_SERVICE_SI_DiagnosticSessionControl = 0x10,
    PUDS_SERVICE_SI_ECUReset = 0x11,
    PUDS_SERVICE_SI_SecurityAccess = 0x27,
    PUDS_SERVICE_SI_CommunicationControl = 0x28,
    PUDS_SERVICE_SI_TesterPresent = 0x3E,
    PUDS_SERVICE_SI_AccessTimingParameter = 0x83,
    PUDS_SERVICE_SI_SecuredDataTransmission = 0x84,
    PUDS_SERVICE_SI_ControlDTCSetting = 0x85,
    PUDS_SERVICE_SI_ResponseOnEvent = 0x86,
    PUDS_SERVICE_SI_LinkControl = 0x87,
    PUDS_SERVICE_SI_ReadDataByIdentifier = 0x22,
    PUDS_SERVICE_SI_ReadMemoryByAddress = 0x23,
    PUDS_SERVICE_SI_ReadScalingDataByIdentifier = 0x24,
    PUDS_SERVICE_SI_ReadDataByPeriodicIdentifier = 0x2A,
    PUDS_SERVICE_SI_DynamicallyDefineDataIdentifier = 0x2C,
    PUDS_SERVICE_SI_WriteDataByIdentifier = 0x2E,
}

```

```

PUDS_SERVICE_SI_WriteMemoryByAddress = 0x3D,
PUDS_SERVICE_SI_ClearDiagnosticInformation = 0x14,
PUDS_SERVICE_SI_ReadDTCInformation = 0x19,
PUDS_SERVICE_SI_InputOutputControlByIdentifier = 0x2F,
PUDS_SERVICE_SI_RoutineControl = 0x31,
PUDS_SERVICE_SI_RequestDownload = 0x34,
PUDS_SERVICE_SI_RequestUpload = 0x35,
PUDS_SERVICE_SI_TransferData = 0x36,
PUDS_SERVICE_SI_RequestTransferExit = 0x37,
PUDS_SERVICE_SI_RequestFileTransfer = 0x38,
PUDS_SERVICE_SI_Authentication = 0x29,
PUDS_SERVICE_NR_SI = 0x7F,
};

```

Visual Basic

```

Public Enum uds_service As Byte
    PUDS_SERVICE_SI_DiagnosticSessionControl = &H10
    PUDS_SERVICE_SI_ECUReset = &H11
    PUDS_SERVICE_SI_SecurityAccess = &H27
    PUDS_SERVICE_SI_CommunicationControl = &H28
    PUDS_SERVICE_SI_TesterPresent = &H3E
    PUDS_SERVICE_SI_AccessTimingParameter = &H83
    PUDS_SERVICE_SI_SecuredDataTransmission = &H84
    PUDS_SERVICE_SI_ControlDTCSetting = &H85
    PUDS_SERVICE_SI_ResponseOnEvent = &H86
    PUDS_SERVICE_SI_LinkControl = &H87
    PUDS_SERVICE_SI_ReadDataByIdentifier = &H22
    PUDS_SERVICE_SI_ReadMemoryByAddress = &H23
    PUDS_SERVICE_SI_ReadScalingDataByIdentifier = &H24
    PUDS_SERVICE_SI_ReadDataByPeriodicIdentifier = &H2A
    PUDS_SERVICE_SI_DynamicallyDefineDataIdentifier = &H2C
    PUDS_SERVICE_SI_WriteDataByIdentifier = &H2E
    PUDS_SERVICE_SI_WriteMemoryByAddress = &H3D
    PUDS_SERVICE_SI_ClearDiagnosticInformation = &H14
    PUDS_SERVICE_SI_ReadDTCInformation = &H19
    PUDS_SERVICE_SI_InputOutputControlByIdentifier = &H2F
    PUDS_SERVICE_SI_RoutineControl = &H31
    PUDS_SERVICE_SI_RequestDownload = &H34
    PUDS_SERVICE_SI_RequestUpload = &H35
    PUDS_SERVICE_SI_TransferData = &H36
    PUDS_SERVICE_SI_RequestTransferExit = &H37
    PUDS_SERVICE_SI_RequestFileTransfer = &H38
    PUDS_SERVICE_SI_Authentication = &H29
    PUDS_SERVICE_NR_SI = &H7F
End Enum

```

Values

| Name | Value | Description |
|----------------------------------|------------|---|
| PUDS_SI_DiagnosticSessionControl | 0x10 (16) | Identifier of the UDS Service DiagnosticSessionControl. |
| PUDS_SI_ECUReset | 0x11 (17) | Identifier of the UDS Service ECUReset. |
| PUDS_SI_SecurityAccess | 0x27 (39) | Identifier of the UDS Service SecurityAccess. |
| PUDS_SI_CommunicationControl | 0x28 (40) | Identifier of the UDS Service CommunicationControl. |
| PUDS_SI_TesterPresent | 0x3E (62) | Identifier of the UDS Service TesterPresent. |
| PUDS_SI_AccessTimingParameter | 0x83 (131) | Identifier of the UDS Service AccessTimingParameter. |
| PUDS_SI_SecuredDataTransmission | 0x84 (132) | Identifier of the UDS Service SecuredDataTransmission. |
| PUDS_SI_ControlDTCSetting | 0x85 (133) | Identifier of the UDS Service ControlDTCSetting. |
| PUDS_SI_ResponseOnEvent | 0x86 (134) | Identifier of the UDS Service ResponseOnEvent. |
| PUDS_SI_LinkControl | 0x87 (135) | Identifier of the UDS Service LinkControl. |

| Name | Value | Description |
|---|------------|--|
| PUDS_SI_ReadDataByIdentifier | 0x22 (34) | Identifier of the UDS Service ReadDataByIdentifier. |
| PUDS_SI_ReadMemoryByAddress | 0x23 (35) | Identifier of the UDS Service ReadMemoryByAddress. |
| PUDS_SI_ReadScalingDataByIdentifier | 0x24 (36) | Identifier of the UDS Service ReadScalingDataByIdentifier. |
| PUDS_SI_ReadDataByPeriodicIdentifier | 0x2A (42) | Identifier of the UDS Service ReadDataByPeriodicIdentifier. |
| PUDS_SI_DynamicallyDefineDataIdentifier | 0x2C (44) | Identifier of the UDS Service DynamicallyDefineDataIdentifier. |
| PUDS_SI_WriteDataByIdentifier | 0x2E (46) | Identifier of the UDS Service WriteDataByIdentifier. |
| PUDS_SI_WriteMemoryByAddress | 0x3D (61) | Identifier of the UDS Service WriteMemoryByAddress. |
| PUDS_SI_ClearDiagnosticInformation | 0x14 (20) | Identifier of the UDS Service ClearDiagnosticInformation. |
| PUDS_SI_ReadDTCInformation | 0x19 (25) | Identifier of the UDS Service ReadDTCInformation. |
| PUDS_SI_InputOutputControlByIdentifier | 0x2F (47) | Identifier of the UDS Service InputOutputControlByIdentifier. |
| PUDS_SI_RoutineControl | 0x31 (49) | Identifier of the UDS Service RoutineControl. |
| PUDS_SI_RequestDownload | 0x34 (52) | Identifier of the UDS Service RequestDownload. |
| PUDS_SI_RequestUpload | 0x35 (53) | Identifier of the UDS Service RequestUpload. |
| PUDS_SI_TransferData | 0x36 (54) | Identifier of the UDS Service TransferData. |
| PUDS_SI_RequestTransferExit | 0x37 (55) | Identifier of the UDS Service RequestTransferExit. |
| PUDS_SI_RequestFileTransfer | 0x38 (56) | Identifier of the UDS Service RequestFileTransfer. |
| PUDS_SI_Authentication | 0x29 (41) | Identifier of the UDS Service Authentication. |
| PUDS_NR_SI | 0x7F (127) | UDS Negative response identifier. |

3.5.5 uds_address

Represents the legislated addresses used with OBD (ISO-15765-4) communication. According to the programming language, this type can be a group of defined values or an enumeration.

Syntax

C/C++

```
#define PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT      0xF1
#define PUDS_ISO_15765_4_ADDR_OBD_FUNCTIONAL      0x33
#define PUDS_ISO_15765_4_ADDR_ECU_1               0x01
#define PUDS_ISO_15765_4_ADDR_ECU_2               0x02
#define PUDS_ISO_15765_4_ADDR_ECU_3               0x03
#define PUDS_ISO_15765_4_ADDR_ECU_4               0x04
#define PUDS_ISO_15765_4_ADDR_ECU_5               0x05
#define PUDS_ISO_15765_4_ADDR_ECU_6               0x06
#define PUDS_ISO_15765_4_ADDR_ECU_7               0x07
#define PUDS_ISO_15765_4_ADDR_ECU_8               0x08
```

Pascal OO

```
uds_address = (
  PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT = $F1,
  PUDS_ADDRESS_ISO_15765_4_ADDR_OBD_FUNCTIONAL = $33,
  PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1 = $1,
  PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_2 = $2,
  PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_3 = $3,
  PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_4 = $4,
  PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_5 = $5,
  PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_6 = $6,
  PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_7 = $7,
  PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_8 = $8
);
```


C#

```
public enum uds_address : UInt16
{
    PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT = 0xF1,
    PUDS_ADDRESS_ISO_15765_4_ADDR_OBD_FUNCTIONAL = 0x33,
    PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1 = 0x01,
    PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_2 = 0x02,
    PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_3 = 0x03,
    PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_4 = 0x04,
    PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_5 = 0x05,
    PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_6 = 0x06,
    PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_7 = 0x07,
    PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_8 = 0x08
}
```

C++ / CLR

```
public enum uds_address : UInt16
{
    PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT = 0xF1,
    PUDS_ADDRESS_ISO_15765_4_ADDR_OBD_FUNCTIONAL = 0x33,
    PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1 = 0x01,
    PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_2 = 0x02,
    PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_3 = 0x03,
    PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_4 = 0x04,
    PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_5 = 0x05,
    PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_6 = 0x06,
    PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_7 = 0x07,
    PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_8 = 0x08
};
```

Visual Basic

```
Public Enum uds_address As UInt16
    PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT = &HF1
    PUDS_ADDRESS_ISO_15765_4_ADDR_OBD_FUNCTIONAL = &H33
    PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1 = &H1
    PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_2 = &H2
    PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_3 = &H3
    PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_4 = &H4
    PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_5 = &H5
    PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_6 = &H6
    PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_7 = &H7
    PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_8 = &H8
End Enum
```

Values

| Name | Value | Description |
|--------------------------------------|------------|---|
| PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT | 0xF1 (241) | Legislated physical address for external equipment. |
| PUDS_ISO_15765_4_ADDR_OBD_FUNCTIONAL | 0x33 (51) | Functional address for legislated OBD system. |
| PUDS_ISO_15765_4_ADDR_ECU_1 | 0x01 (1) | Legislated-OBD ECU #1. |
| PUDS_ISO_15765_4_ADDR_ECU_2 | 0x02 (2) | Legislated-OBD ECU #2. |
| PUDS_ISO_15765_4_ADDR_ECU_3 | 0x03 (3) | Legislated-OBD ECU #3. |
| PUDS_ISO_15765_4_ADDR_ECU_4 | 0x04 (4) | Legislated-OBD ECU #4. |
| PUDS_ISO_15765_4_ADDR_ECU_5 | 0x05 (5) | Legislated-OBD ECU #5. |
| PUDS_ISO_15765_4_ADDR_ECU_6 | 0x06 (6) | Legislated-OBD ECU #6. |
| PUDS_ISO_15765_4_ADDR_ECU_7 | 0x07 (7) | Legislated-OBD ECU #7. |
| PUDS_ISO_15765_4_ADDR_ECU_8 | 0x08 (8) | Legislated-OBD ECU #8. |

See also: `uds_netaddrinfo` on page 24.

3.5.6 uds_can_id

Represents the legislated CAN Identifiers used with OBD (ISO-15765-4) communication. Each of these CAN identifiers is assigned to a specific network addressing information (see `uds_mapping` on page 25 and default mappings in UDS and ISO-TP Network Addressing Information on page 770). According to the programming language, this type can be a group of defined values or an enumeration.

Syntax

C/C++

```
#define PUDS_ISO_15765_4_CAN_ID_FUNCTIONAL_REQUEST      0x7DF
#define PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1     0x7E0
#define PUDS_ISO_15765_4_CAN_ID_PHYSICAL_RESPONSE_1   0x7E8
#define PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_2     0x7E1
#define PUDS_ISO_15765_4_CAN_ID_PHYSICAL_RESPONSE_2   0x7E9
#define PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_3     0x7E2
#define PUDS_ISO_15765_4_CAN_ID_PHYSICAL_RESPONSE_3   0x7EA
#define PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_4     0x7E3
#define PUDS_ISO_15765_4_CAN_ID_PHYSICAL_RESPONSE_4   0x7EB
#define PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_5     0x7E4
#define PUDS_ISO_15765_4_CAN_ID_PHYSICAL_RESPONSE_5   0x7EC
#define PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_6     0x7E5
#define PUDS_ISO_15765_4_CAN_ID_PHYSICAL_RESPONSE_6   0x7ED
#define PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_7     0x7E6
#define PUDS_ISO_15765_4_CAN_ID_PHYSICAL_RESPONSE_7   0x7EE
#define PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_8     0x7E7
#define PUDS_ISO_15765_4_CAN_ID_PHYSICAL_RESPONSE_8   0x7EF
```

Pascal OO

```
uds_can_id = (
  PUDS_CAN_ID_ISO_15765_4_FUNCTIONAL_REQUEST = $7DF,
  PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1 = $7E0,
  PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_1 = $7E8,
  PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_2 = $7E1,
  PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_2 = $7E9,
  PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_3 = $7E2,
  PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_3 = $7EA,
  PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_4 = $7E3,
  PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_4 = $7EB,
  PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_5 = $7E4,
  PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_5 = $7EC,
  PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_6 = $7E5,
  PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_6 = $7ED,
  PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_7 = $7E6,
  PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_7 = $7EE,
  PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_8 = $7E7,
  PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_8 = $7EF
);
```

C#

```
public enum uds_can_id : UInt32
{
  PUDS_CAN_ID_ISO_15765_4_FUNCTIONAL_REQUEST = 0x7DF,
  PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1 = 0x7E0,
  PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_1 = 0x7E8,
  PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_2 = 0x7E1,
```

```

PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_2 = 0x7E9,
PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_3 = 0x7E2,
PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_3 = 0x7EA,
PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_4 = 0x7E3,
PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_4 = 0x7EB,
PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_5 = 0x7E4,
PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_5 = 0x7EC,
PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_6 = 0x7E5,
PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_6 = 0x7ED,
PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_7 = 0x7E6,
PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_7 = 0x7EE,
PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_8 = 0x7E7,
PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_8 = 0x7EF
}

```

C++ / CLR

```

public enum uds_can_id : UInt32
{
    PUDS_CAN_ID_ISO_15765_4_FUNCTIONAL_REQUEST = 0x7DF,
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1 = 0x7E0,
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_1 = 0x7E8,
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_2 = 0x7E1,
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_2 = 0x7E9,
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_3 = 0x7E2,
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_3 = 0x7EA,
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_4 = 0x7E3,
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_4 = 0x7EB,
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_5 = 0x7E4,
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_5 = 0x7EC,
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_6 = 0x7E5,
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_6 = 0x7ED,
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_7 = 0x7E6,
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_7 = 0x7EE,
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_8 = 0x7E7,
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_8 = 0x7EF
};

```

Visual Basic

```

Public Enum uds_can_id As UInt32
    PUDS_CAN_ID_ISO_15765_4_FUNCTIONAL_REQUEST = &H7DF
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1 = &H7E0
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_1 = &H7E8
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_2 = &H7E1
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_2 = &H7E9
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_3 = &H7E2
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_3 = &H7EA
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_4 = &H7E3
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_4 = &H7EB
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_5 = &H7E4
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_5 = &H7EC
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_6 = &H7E5
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_6 = &H7ED
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_7 = &H7E6
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_7 = &H7EE
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_8 = &H7E7
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_8 = &H7EF
End Enum

```

values

| Name | Value | Description |
|---|-------|---|
| PUDS_ISO_15765_4_CAN_ID_FUNCTIONAL_REQUEST | 0x7DF | CAN identifier for functionally addressed request messages sent by external test equipment. |
| PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1 | 0x7E0 | Physical request CAN ID from external test equipment to ECU #1. |
| PUDS_ISO_15765_4_CAN_ID_PHYSICAL_RESPONSE_1 | 0x7E8 | Physical response CAN ID from ECU #1 to external test equipment. |
| PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_2 | 0x7E1 | Physical request CAN ID from external test equipment to ECU #2. |
| PUDS_ISO_15765_4_CAN_ID_PHYSICAL_RESPONSE_2 | 0x7E9 | Physical response CAN ID from ECU #2 to external test equipment. |
| PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_3 | 0x7E2 | Physical request CAN ID from external test equipment to ECU #3. |
| PUDS_ISO_15765_4_CAN_ID_PHYSICAL_RESPONSE_3 | 0x7EA | Physical response CAN ID from ECU #3 to external test equipment. |
| PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_4 | 0x7E3 | Physical request CAN ID from external test equipment to ECU #4. |
| PUDS_ISO_15765_4_CAN_ID_PHYSICAL_RESPONSE_4 | 0x7EB | Physical response CAN ID from ECU #4 to external test equipment. |
| PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_5 | 0x7E4 | Physical request CAN ID from external test equipment to ECU #5. |
| PUDS_ISO_15765_4_CAN_ID_PHYSICAL_RESPONSE_5 | 0x7EC | Physical response CAN ID from ECU #5 to external test equipment. |
| PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_6 | 0x7E5 | Physical request CAN ID from external test equipment to ECU #6. |
| PUDS_ISO_15765_4_CAN_ID_PHYSICAL_RESPONSE_6 | 0x7ED | Physical response CAN ID from ECU #6 to external test equipment. |
| PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_7 | 0x7E6 | Physical request CAN ID from external test equipment to ECU #7. |
| PUDS_ISO_15765_4_CAN_ID_PHYSICAL_RESPONSE_7 | 0x7EE | Physical response CAN ID from ECU #7 to external test equipment. |
| PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_8 | 0x7E7 | Physical request CAN ID from external test equipment to ECU #8. |
| PUDS_ISO_15765_4_CAN_ID_PHYSICAL_RESPONSE_8 | 0x7EF | Physical response CAN ID from ECU #8 to external test equipment. |

See also: [uds_netaddrinfo](#) on page 24.

3.5.7 uds_status_offset

Defines constants used by the [uds_status](#) enumeration. According to the programming language, this type can be a group of defined values or an enumeration.

Syntax

C/C++

```
#define PCANTP_STATUS_OFFSET_BUS 8
#define PCANTP_STATUS_OFFSET_NET (PCANTP_STATUS_OFFSET_BUS + 5)
#define PCANTP_STATUS_OFFSET_INFO (PCANTP_STATUS_OFFSET_NET + 5)
#define PCANTP_STATUS_OFFSET_UDS (PCANTP_STATUS_OFFSET_INFO + 6)
```

Pascal OO

```
const
  PCANTP_STATUS_OFFSET_BUS = 8;
  PCANTP_STATUS_OFFSET_NET = (PCANTP_STATUS_OFFSET_BUS + 5);
  PCANTP_STATUS_OFFSET_INFO = (PCANTP_STATUS_OFFSET_NET + 5);
  PCANTP_STATUS_OFFSET_UDS = (PCANTP_STATUS_OFFSET_INFO + 6);
```

C#

```
public enum uds_status_offset : byte
{
    PCANTP_STATUS_OFFSET_BUS = 8,
```

```

PCANTP_STATUS_OFFSET_NET = (PCANTP_STATUS_OFFSET_BUS + 5),
PCANTP_STATUS_OFFSET_INFO = (PCANTP_STATUS_OFFSET_NET + 5),
PCANTP_STATUS_OFFSET_UDS = (PCANTP_STATUS_OFFSET_INFO + 6)
}

```

C++ / CLR

```

public enum uds_status_offset : Byte
{
    PCANTP_STATUS_OFFSET_BUS = 8,
    PCANTP_STATUS_OFFSET_NET = (PCANTP_STATUS_OFFSET_BUS + 5),
    PCANTP_STATUS_OFFSET_INFO = (PCANTP_STATUS_OFFSET_NET + 5),
    PCANTP_STATUS_OFFSET_UDS = (PCANTP_STATUS_OFFSET_INFO + 6)
};

```

Visual Basic

```

Public Enum uds_status_offset As Byte
    PCANTP_STATUS_OFFSET_BUS = 8
    PCANTP_STATUS_OFFSET_NET = (PCANTP_STATUS_OFFSET_BUS + 5)
    PCANTP_STATUS_OFFSET_INFO = (PCANTP_STATUS_OFFSET_NET + 5)
    PCANTP_STATUS_OFFSET_UDS = (PCANTP_STATUS_OFFSET_INFO + 6)
End Enum

```

Values

| Name | Value | Description |
|---------------------------|-------|---|
| PCANTP_STATUS_OFFSET_BUS | 8 | Bit shifting offset used for bus status code (see uds_status on page 32). |
| PCANTP_STATUS_OFFSET_NET | 13 | Bit shifting offset used for network status code (see uds_status on page 32). |
| PCANTP_STATUS_OFFSET_INFO | 18 | Bit shifting offset used for extra information status code (see uds_status on page 32). |
| PCANTP_STATUS_OFFSET_UDS | 24 | Bit shifting offset used for PUDS status code (see uds_status on page 32). |

See also: [uds_status](#) on page 32.

3.5.8 uds_msgprotocol

Represents a standardized and supported network communication protocol. It is a combination of the ISO-TP network addressing information supported in UDS.

Syntax

C/C++

```

typedef enum _uds_msgprotocol {
    PUDS_MSGPROTOCOL_NONE = 0x00,
    PUDS_MSGPROTOCOL_ISO_15765_2_11B_EXTENDED = 0x07,
    PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL = 0x01,
    PUDS_MSGPROTOCOL_ISO_15765_2_11B_REMOTE = 0x02,
    PUDS_MSGPROTOCOL_ISO_15765_2_29B_EXTENDED = 0x08,
    PUDS_MSGPROTOCOL_ISO_15765_2_29B_FIXED_NORMAL = 0x03,
    PUDS_MSGPROTOCOL_ISO_15765_2_29B_NORMAL = 0x06,
    PUDS_MSGPROTOCOL_ISO_15765_2_29B_REMOTE = 0x04,
    PUDS_MSGPROTOCOL_ISO_15765_3_29B_ENHANCED = 0x05,
} uds_msgprotocol;

```

Pascal OO

```

uds_msgprotocol = (
    PUDS_MSGPROTOCOL_NONE = UInt32($0),
    PUDS_MSGPROTOCOL_ISO_15765_2_11B_EXTENDED = UInt32($07),

```

```

PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL = UInt32($01),
PUDS_MSGPROTOCOL_ISO_15765_2_11B_REMOTE = UInt32($02),
PUDS_MSGPROTOCOL_ISO_15765_2_29B_EXTENDED = UInt32($08),
PUDS_MSGPROTOCOL_ISO_15765_2_29B_FIXED_NORMAL = UInt32($03),
PUDS_MSGPROTOCOL_ISO_15765_2_29B_NORMAL = UInt32($06),
PUDS_MSGPROTOCOL_ISO_15765_2_29B_REMOTE = UInt32($04),
PUDS_MSGPROTOCOL_ISO_15765_3_29B_ENHANCED = UInt32($05));

```

C#

```

public enum uds_msgprotocol : UInt32
{
    PUDS_MSGPROTOCOL_NONE = 0x00,
    PUDS_MSGPROTOCOL_ISO_15765_2_11B_EXTENDED = 0x07,
    PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL = 0x01,
    PUDS_MSGPROTOCOL_ISO_15765_2_11B_REMOTE = 0x02,
    PUDS_MSGPROTOCOL_ISO_15765_2_29B_EXTENDED = 0x08,
    PUDS_MSGPROTOCOL_ISO_15765_2_29B_FIXED_NORMAL = 0x03,
    PUDS_MSGPROTOCOL_ISO_15765_2_29B_NORMAL = 0x06,
    PUDS_MSGPROTOCOL_ISO_15765_2_29B_REMOTE = 0x04,
    PUDS_MSGPROTOCOL_ISO_15765_3_29B_ENHANCED = 0x05
}

```

C++ / CLR

```

public enum uds_msgprotocol : UInt32
{
    PUDS_MSGPROTOCOL_NONE = 0x00,
    PUDS_MSGPROTOCOL_ISO_15765_2_11B_EXTENDED = 0x07,
    PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL = 0x01,
    PUDS_MSGPROTOCOL_ISO_15765_2_11B_REMOTE = 0x02,
    PUDS_MSGPROTOCOL_ISO_15765_2_29B_EXTENDED = 0x08,
    PUDS_MSGPROTOCOL_ISO_15765_2_29B_FIXED_NORMAL = 0x03,
    PUDS_MSGPROTOCOL_ISO_15765_2_29B_NORMAL = 0x06,
    PUDS_MSGPROTOCOL_ISO_15765_2_29B_REMOTE = 0x04,
    PUDS_MSGPROTOCOL_ISO_15765_3_29B_ENHANCED = 0x05
};

```

Visual Basic

```

Public Enum uds_msgprotocol As UInt32
    PUDS_MSGPROTOCOL_NONE = &H0
    PUDS_MSGPROTOCOL_ISO_15765_2_11B_EXTENDED = &H7
    PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL = &H1
    PUDS_MSGPROTOCOL_ISO_15765_2_11B_REMOTE = &H2
    PUDS_MSGPROTOCOL_ISO_15765_2_29B_EXTENDED = &H8
    PUDS_MSGPROTOCOL_ISO_15765_2_29B_FIXED_NORMAL = &H3
    PUDS_MSGPROTOCOL_ISO_15765_2_29B_NORMAL = &H6
    PUDS_MSGPROTOCOL_ISO_15765_2_29B_REMOTE = &H4
    PUDS_MSGPROTOCOL_ISO_15765_3_29B_ENHANCED = &H5
End Enum

```

Values

| Name | Value | Description |
|---|-------|--|
| PUDS_MSGPROTOCOL_NONE | 0 | Network layer configuration for Unacknowledged Unsegmented Data Transfer (UUDT). |
| PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL | 1 | Network layer configuration for the PCAN-ISO-TP 3.x API: 11 BIT CAN identifier, NORMAL addressing, and diagnostic message type (mapping required). |

| Name | Value | Description |
|---|-------|--|
| PUDS_MSGPROTOCOL_ISO_15765_2_11B_EXTENDED | 7 | Network layer configuration for the PCAN-ISO-TP 3.x API: 11 BIT CAN identifier, EXTENDED addressing, and diagnostic message type. Note that specific CAN identifier mappings must be configured via the PCAN-ISO-TP 3.x API to transmit and receive such messages. |
| PUDS_MSGPROTOCOL_ISO_15765_2_11B_REMOTE | 2 | Network layer configuration for the PCAN-ISO-TP 3.x API: 11 BIT CAN identifier, MIXED addressing, and remote diagnostic message type (mapping required). |
| PUDS_MSGPROTOCOL_ISO_15765_2_29B_FIXED_NORMAL | 3 | Network layer configuration for the PCAN-ISO-TP 3.x API: 29 BIT CAN identifier, FIXED NORMAL addressing, and diagnostic message type. |
| PUDS_MSGPROTOCOL_ISO_15765_2_29B_EXTENDED | 8 | Network layer configuration for the PCAN-ISO-TP 3.x API: 29 BIT CAN identifier, EXTENDED addressing, and diagnostic message. Note that specific CAN identifier mappings must be configured via the PCAN-ISO-TP 3.x API to transmit and receive such messages. |
| PUDS_MSGPROTOCOL_ISO_15765_2_29B_NORMAL | 6 | Network layer configuration for the PCAN-ISO-TP 3.x API: 29 BIT CAN identifier, NORMAL addressing, and diagnostic message type. Note that specific CAN identifier mappings must be configured via the PCAN-ISO-TP 3.x API to transmit and receive such messages. |
| PUDS_MSGPROTOCOL_ISO_15765_2_29B_REMOTE | 4 | Network layer configuration for the PCAN-ISO-TP 3.x API: 29 BIT CAN identifier, MIXED addressing, and remote diagnostic message type. |
| PUDS_MSGPROTOCOL_ISO_15765_3_29B_ENHANCED | 5 | Network layer configuration for the PCAN-ISO-TP 3.x API: Enhanced diagnostics 29-bit CAN identifiers. Note with ISO-15765:2016, this addressing is considered deprecated and disabled by default. See PCAN-ISO-TP documentation on parameter: PCANTP_PARAMETER_SUPPORT_29B_ENHANCED. |

See also: [uds_netaddrinfo](#) on page 24 and [uds_mapping](#) on page 25.

3.5.9 uds_msgtype

Represents type and flags for a [uds_msg](#).

Syntax

C/C++

```
typedef enum _uds_msgtype {
    PUDS_MSGTYPE_USDT = 0,
    PUDS_MSGTYPE_UUDT = 1,
    PUDS_MSGTYPE_FLAG_NO_POSITIVE_RESPONSE = 2,
    PUDS_MSGTYPE_FLAG_LOOPBACK = 4,
    PUDS_MSGTYPE_MASK_TYPE = 0x01,
} uds_msgtype;
```

Pascal OO

```
uds_msgtype = (
    PUDS_MSGTYPE_USDT = 0,
    PUDS_MSGTYPE_UUDT = 1,
    PUDS_MSGTYPE_FLAG_NO_POSITIVE_RESPONSE = 2,
    PUDS_MSGTYPE_FLAG_LOOPBACK = 4,
    PUDS_MSGTYPE_MASK_TYPE = $1
);
```

C#

```
[Flags]
public enum uds_msgtype : UInt32
{
    PUDS_MSGTYPE_USDT = 0,
    PUDS_MSGTYPE_UUDT = 1,
    PUDS_MSGTYPE_FLAG_NO_POSITIVE_RESPONSE = 2,
    PUDS_MSGTYPE_FLAG_LOOPBACK = 4,
    PUDS_MSGTYPE_MASK_TYPE = 0x01,
}
```

C++ / CLR

```
public enum uds_msgtype : UInt32
{
    PUDS_MSGTYPE_USDT = 0,
    PUDS_MSGTYPE_UUDT = 1,
    PUDS_MSGTYPE_FLAG_NO_POSITIVE_RESPONSE = 2,
    PUDS_MSGTYPE_FLAG_LOOPBACK = 4,
    PUDS_MSGTYPE_MASK_TYPE = 0x01,
};
```

Visual Basic

```
<Flags()>
Public Enum uds_msgtype As UInt32
    PUDS_MSGTYPE_USDT = 0
    PUDS_MSGTYPE_UUDT = 1
    PUDS_MSGTYPE_FLAG_NO_POSITIVE_RESPONSE = 2
    PUDS_MSGTYPE_FLAG_LOOPBACK = 4
    PUDS_MSGTYPE_MASK_TYPE = &H1
End Enum
```

values

| Name | Value | Description |
|--|-------|---|
| PUDS_MSGTYPE_USDT | 0 | Unacknowledged Segmented Data Transfer (ISO-TP message). |
| PUDS_MSGTYPE_UUDT | 1 | Unacknowledged Unsegmented Data Transfer (physical message will use a single CAN/CAN FD frame without ISO-TP protocol control information). |
| PUDS_MSGTYPE_FLAG_NO_POSITIVE_RESPONSE | 2 | ECU(s) shall not reply to the request on a positive response. |
| PUDS_MSGTYPE_FLAG_LOOPBACK | 4 | Message is a loopback. |
| PUDS_MSGTYPE_MASK_TYPE | 1 | Mask to get the type (USDT or UUDT). |

See also: [uds_msgconfig](#) on page 25, [uds_msg](#) on page 21.

3.5.10 uds_nrc

Represents UDS negative response codes (see ISO 14229-1:2013 §A.1 Negative response codes p.325).

Syntax**C/C++**

```
typedef enum _uds_nrc
{
    PUDS_NRC_PR = 0x00,
    PUDS_NRC_GR = 0x10,
    PUDS_NRC_SNS = 0x11,
```



```
PUDS_NRC_SFNS = 0x12,  
PUDS_NRC_IMLOIF = 0x13,  
PUDS_NRC_RTL = 0x14,  
PUDS_NRC_BRR = 0x21,  
PUDS_NRC_CNC = 0x22,  
PUDS_NRC_RSE = 0x24,  
PUDS_NRC_NRFSC = 0x25,  
PUDS_NRC_FPEORA = 0x26,  
PUDS_NRC_ROOR = 0x31,  
PUDS_NRC_SAD = 0x33,  
PUDS_NRC_AR = 0x34,  
PUDS_NRC_IK = 0x35,  
PUDS_NRC_ENOA = 0x36,  
PUDS_NRC_RTDNE = 0x37,  
PUDS_NRC_SDTR = 0x38,  
PUDS_NRC_SDTNA = 0x39,  
PUDS_NRC_SDTF = 0x3A,  
PUDS_NRC_CVFITP = 0x50,  
PUDS_NRC_CVFISIG = 0x51,  
PUDS_NRC_CVFICOT = 0x52,  
PUDS_NRC_CVFIT = 0x53,  
PUDS_NRC_CVFIF = 0x54,  
PUDS_NRC_CVFIC = 0x55,  
PUDS_NRC_CVFISCP = 0x56,  
PUDS_NRC_CVFICERT = 0x57,  
PUDS_NRC_OVF = 0x58,  
PUDS_NRC_CCF = 0x59,  
PUDS_NRC_SARF = 0x5A,  
PUDS_NRC_SKCDF = 0x5B,  
PUDS_NRC_CDUF = 0x5C,  
PUDS_NRC_DAF = 0x5D,  
PUDS_NRC_UDNA = 0x70,  
PUDS_NRC_TDS = 0x71,  
PUDS_NRC_GPF = 0x72,  
PUDS_NRC_WBSC = 0x73,  
PUDS_NRC_RCRRP = 0x78,  
PUDS_NRC_SFNSIAS = 0x7E,  
PUDS_NRC_SNSIAS = 0x7F,  
PUDS_NRC_RPMTH = 0x81,  
PUDS_NRC_RPMTL = 0x82,  
PUDS_NRC_EIR = 0x83,  
PUDS_NRC_EINR = 0x84,  
PUDS_NRC_ERTTL = 0x85,  
PUDS_NRC_TEMPTH = 0x86,  
PUDS_NRC_TEMPTL = 0x87,  
PUDS_NRC_VSTH = 0x88,  
PUDS_NRC_VSTL = 0x89,  
PUDS_NRC_TPTH = 0x8A,  
PUDS_NRC_TPTL = 0x8B,  
PUDS_NRC_TRNIN = 0x8C,  
PUDS_NRC_TRNIG = 0x8D,  
PUDS_NRC_BSNC = 0x8F,  
PUDS_NRC_SLNIP = 0x90,  
PUDS_NRC_TCCL = 0x91,  
PUDS_NRC_VTH = 0x92,  
PUDS_NRC_VTL = 0x93,  
PUDS_NRC_RTNA = 0x94  
} uds_nrc;
```

Pascal OO

```
uds_nrc = (  
    PUDS_NRC_PR = $00,  
    PUDS_NRC_GR = $10,  
    PUDS_NRC_SNS = $11,  
    PUDS_NRC_SFNS = $12,  
    PUDS_NRC_IMLOIF = $13,  
    PUDS_NRC_RTL = $14,  
    PUDS_NRC_BRR = $21,  
    PUDS_NRC_CNC = $22,  
    PUDS_NRC_RSE = $24,  
    PUDS_NRC_NRFSC = $25,  
    PUDS_NRC_FPEORA = $26,  
    PUDS_NRC_ROOR = $31,  
    PUDS_NRC_SAD = $33,  
    PUDS_NRC_AR = $34,  
    PUDS_NRC_IK = $35,  
    PUDS_NRC_ENOA = $36,  
    PUDS_NRC_RTDNE = $37,  
    PUDS_NRC_SDTR = $38,  
    PUDS_NRC_SDTNA = $39,  
    PUDS_NRC_SDTF = $3A,  
    PUDS_NRC_CVFITP = $50,  
    PUDS_NRC_CVFISIG = $51,  
    PUDS_NRC_CVFICOT = $52,  
    PUDS_NRC_CVFIT = $53,  
    PUDS_NRC_CVFIF = $54,  
    PUDS_NRC_CVFIC = $55,  
    PUDS_NRC_CVFISCP = $56,  
    PUDS_NRC_CVFICERT = $57,  
    PUDS_NRC_OVF = $58,  
    PUDS_NRC_CCF = $59,  
    PUDS_NRC_SARF = $5A,  
    PUDS_NRC_SKCDF = $5B,  
    PUDS_NRC_CDUF = $5C,  
    PUDS_NRC_DAF = $5D,  
    PUDS_NRC_UDNA = $70,  
    PUDS_NRC_TDS = $71,  
    PUDS_NRC_GPF = $72,  
    PUDS_NRC_WBSC = $73,  
    PUDS_NRC_RCRRP = $78,  
    PUDS_NRC_SFNSIAS = $7E,  
    PUDS_NRC_SNSIAS = $7F,  
    PUDS_NRC_RPMTH = $81,  
    PUDS_NRC_RPMTL = $82,  
    PUDS_NRC_EIR = $83,  
    PUDS_NRC_EINR = $84,  
    PUDS_NRC_ERTTL = $85,  
    PUDS_NRC_TEMPTH = $86,  
    PUDS_NRC_TEMPTL = $87,  
    PUDS_NRC_VSTH = $88,  
    PUDS_NRC_VSTL = $89,  
    PUDS_NRC_TPTH = $8A,  
    PUDS_NRC_TPTL = $8B,  
    PUDS_NRC_TRNIN = $8C,  
    PUDS_NRC_TRNIG = $8D,  
    PUDS_NRC_BSNC = $8F,  
    PUDS_NRC_SLNIP = $90,  
    PUDS_NRC_TCCL = $91,
```

```
PUDS_NRC_VTH = $92,  
PUDS_NRC_VTL = $93,  
PUDS_NRC_RTNA = $94);
```

C#

```
public enum uds_nrc : Byte  
{  
    PUDS_NRC_PR = 0x00,  
    PUDS_NRC_GR = 0x10,  
    PUDS_NRC_SNS = 0x11,  
    PUDS_NRC_SFNS = 0x12,  
    PUDS_NRC_IMLOIF = 0x13,  
    PUDS_NRC_RTL = 0x14,  
    PUDS_NRC_BRR = 0x21,  
    PUDS_NRC_CNC = 0x22,  
    PUDS_NRC_RSE = 0x24,  
    PUDS_NRC_NRFSC = 0x25,  
    PUDS_NRC_FPEORA = 0x26,  
    PUDS_NRC_ROOR = 0x31,  
    PUDS_NRC_SAD = 0x33,  
    PUDS_NRC_AR = 0x34,  
    PUDS_NRC_IK = 0x35,  
    PUDS_NRC_ENOA = 0x36,  
    PUDS_NRC_RTDNE = 0x37,  
    PUDS_NRC_SDTR = 0x38,  
    PUDS_NRC_SDTNA = 0x39,  
    PUDS_NRC_SDTF = 0x3A,  
    PUDS_NRC_CVFITP = 0x50,  
    PUDS_NRC_CVFISIG = 0x51,  
    PUDS_NRC_CVFICOT = 0x52,  
    PUDS_NRC_CVFIT = 0x53,  
    PUDS_NRC_CVFIF = 0x54,  
    PUDS_NRC_CVFIC = 0x55,  
    PUDS_NRC_CVFISCP = 0x56,  
    PUDS_NRC_CVFICERT = 0x57,  
    PUDS_NRC_OVF = 0x58,  
    PUDS_NRC_CCF = 0x59,  
    PUDS_NRC_SARF = 0x5A,  
    PUDS_NRC_SKCDF = 0x5B,  
    PUDS_NRC_CDUF = 0x5C,  
    PUDS_NRC_DAF = 0x5D,  
    PUDS_NRC_UDNA = 0x70,  
    PUDS_NRC_TDS = 0x71,  
    PUDS_NRC_GPF = 0x72,  
    PUDS_NRC_WBSC = 0x73,  
    PUDS_NRC_RCRRP = 0x78,  
    PUDS_NRC_SFNSIAS = 0x7E,  
    PUDS_NRC_SNSIAS = 0x7F,  
    PUDS_NRC_RPMTH = 0x81,  
    PUDS_NRC_RPMTL = 0x82,  
    PUDS_NRC_EIR = 0x83,  
    PUDS_NRC_EINR = 0x84,  
    PUDS_NRC_ERTTL = 0x85,  
    PUDS_NRC_TEMPTH = 0x86,  
    PUDS_NRC_TEMPTL = 0x87,  
    PUDS_NRC_VSTH = 0x88,  
    PUDS_NRC_VSTL = 0x89,  
    PUDS_NRC_TPTH = 0x8A,  
    PUDS_NRC_TPTL = 0x8B,  
    PUDS_NRC_TRNIN = 0x8C,  
    PUDS_NRC_TRNIG = 0x8D,
```

```

PUDS_NRC_BSNC = 0x8F,
PUDS_NRC_SLNIP = 0x90,
PUDS_NRC_TCCL = 0x91,
PUDS_NRC_VTH = 0x92,
PUDS_NRC_VTL = 0x93,
PUDS_NRC_RTNA = 0x94
}

```

C++ / CLR

```

public enum uds_nrc : Byte
{
    PUDS_NRC_PR = 0x00,
    PUDS_NRC_GR = 0x10,
    PUDS_NRC_SNS = 0x11,
    PUDS_NRC_SFNS = 0x12,
    PUDS_NRC_IMLOIF = 0x13,
    PUDS_NRC_RTL = 0x14,
    PUDS_NRC_BRR = 0x21,
    PUDS_NRC_CNC = 0x22,
    PUDS_NRC_RSE = 0x24,
    PUDS_NRC_NRFSC = 0x25,
    PUDS_NRC_FPEORA = 0x26,
    PUDS_NRC_RORR = 0x31,
    PUDS_NRC_SAD = 0x33,
    PUDS_NRC_AR = 0x34,
    PUDS_NRC_IK = 0x35,
    PUDS_NRC_ENOA = 0x36,
    PUDS_NRC_RTDNE = 0x37,
    PUDS_NRC_SDTR = 0x38,
    PUDS_NRC_SDTNA = 0x39,
    PUDS_NRC_SDTF = 0x3A,
    PUDS_NRC_CVFITP = 0x50,
    PUDS_NRC_CVFISIG = 0x51,
    PUDS_NRC_CVFICOT = 0x52,
    PUDS_NRC_CVFIT = 0x53,
    PUDS_NRC_CVFIF = 0x54,
    PUDS_NRC_CVFIC = 0x55,
    PUDS_NRC_CVFISCP = 0x56,
    PUDS_NRC_CVFICERT = 0x57,
    PUDS_NRC_OVF = 0x58,
    PUDS_NRC_CCF = 0x59,
    PUDS_NRC_SARF = 0x5A,
    PUDS_NRC_SKCDF = 0x5B,
    PUDS_NRC_CDUF = 0x5C,
    PUDS_NRC_DAF = 0x5D,
    PUDS_NRC_UDNA = 0x70,
    PUDS_NRC_TDS = 0x71,
    PUDS_NRC_GPF = 0x72,
    PUDS_NRC_WBSC = 0x73,
    PUDS_NRC_RCRRP = 0x78,
    PUDS_NRC_SFNSIAS = 0x7E,
    PUDS_NRC_SNSIAS = 0x7F,
    PUDS_NRC_RPMTH = 0x81,
    PUDS_NRC_RPMTL = 0x82,
    PUDS_NRC_EIR = 0x83,
    PUDS_NRC_EINR = 0x84,
    PUDS_NRC_ERTTL = 0x85,
    PUDS_NRC_TEMPTH = 0x86,
    PUDS_NRC_TEMPTL = 0x87,
    PUDS_NRC_VSTH = 0x88,
    PUDS_NRC_VSTL = 0x89,
}

```

```

PUDS_NRC_TPTH = 0x8A,
PUDS_NRC_TPTL = 0x8B,
PUDS_NRC_TRNIN = 0x8C,
PUDS_NRC_TRNIG = 0x8D,
PUDS_NRC_BSNC = 0x8F,
PUDS_NRC_SLNIP = 0x90,
PUDS_NRC_TCCL = 0x91,
PUDS_NRC_VTH = 0x92,
PUDS_NRC_VTL = 0x93,
PUDS_NRC_RTNA = 0x94
};

```

Visual Basic

```

Public Enum uds_nrc As Byte
    PUDS_NRC_PR = &H00
    PUDS_NRC_GR = &H10
    PUDS_NRC_SNS = &H11
    PUDS_NRC_SFNS = &H12
    PUDS_NRC_IMLOIF = &H13
    PUDS_NRC_RTL = &H14
    PUDS_NRC_BRR = &H21
    PUDS_NRC_CNC = &H22
    PUDS_NRC_RSE = &H24
    PUDS_NRC_NRFSC = &H25
    PUDS_NRC_FPEORA = &H26
    PUDS_NRC_ROOR = &H31
    PUDS_NRC_SAD = &H33
    PUDS_NRC_AR = &H34
    PUDS_NRC_IK = &H35
    PUDS_NRC_ENOA = &H36
    PUDS_NRC_RTDNE = &H37
    PUDS_NRC_SDTR = &H38
    PUDS_NRC_SDTNA = &H39
    PUDS_NRC_SDTF = &H3A
    PUDS_NRC_CVFITP = &H50
    PUDS_NRC_CVFISIG = &H51
    PUDS_NRC_CVFICOT = &H52
    PUDS_NRC_CVFIT = &H53
    PUDS_NRC_CVFIF = &H54
    PUDS_NRC_CVFIC = &H55
    PUDS_NRC_CVFISCP = &H56
    PUDS_NRC_CVFICERT = &H57
    PUDS_NRC_OVF = &H58
    PUDS_NRC_CCF = &H59
    PUDS_NRC_SARF = &H5A
    PUDS_NRC_SKCDF = &H5B
    PUDS_NRC_CDUF = &H5C
    PUDS_NRC_DAF = &H5D
    PUDS_NRC_UDNA = &H70
    PUDS_NRC_TDS = &H71
    PUDS_NRC_GPF = &H72
    PUDS_NRC_WBSC = &H73
    PUDS_NRC_RCRRP = &H78
    PUDS_NRC_SFNSIAS = &H7E
    PUDS_NRC_SNSIAS = &H7F
    PUDS_NRC_RPMTH = &H81
    PUDS_NRC_RPMTL = &H82
    PUDS_NRC_EIR = &H83
    PUDS_NRC_EINR = &H84
    PUDS_NRC_ERTTL = &H85
    PUDS_NRC_TEMPTH = &H86

```

```

PUDS_NRC_TEMPTL = &H87
PUDS_NRC_VSTH = &H88
PUDS_NRC_VSTL = &H89
PUDS_NRC_TPTH = &H8A
PUDS_NRC_TPTL = &H8B
PUDS_NRC_TRNIN = &H8C
PUDS_NRC_TRNIG = &H8D
PUDS_NRC_BSNC = &H8F
PUDS_NRC_SLNIP = &H90
PUDS_NRC_TCCL = &H91
PUDS_NRC_VTH = &H92
PUDS_NRC_VTL = &H93
PUDS_NRC_RTNA = &H94

```

End Enum

values

| Name | Value | Description |
|-------------------|-------|--|
| PUDS_NRC_PR | 0x00 | Positive Response |
| PUDS_NRC_GR | 0x10 | General Reject |
| PUDS_NRC_SNS | 0x11 | Service Not Supported |
| PUDS_NRC_SFNS | 0x12 | Sub Function Not Supported |
| PUDS_NRC_IMLOIF | 0x13 | Incorrect Message Length Or Invalid Format |
| PUDS_NRC_RTL | 0x14 | Response Too Long |
| PUDS_NRC_BRR | 0x21 | Busy Repeat Request |
| PUDS_NRC_CNC | 0x22 | Conditions Not Correct |
| PUDS_NRC_RSE | 0x24 | Request Sequence Error |
| PUDS_NRC_NRFSC | 0x25 | No Response From Subnet Component |
| PUDS_NRC_FPEORA | 0x26 | Failure Prevents Execution Of Requested Action |
| PUDS_NRC_ROOR | 0x31 | Request Out Of Range |
| PUDS_NRC_SAD | 0x33 | Security Access Denied |
| PUDS_NRC_AR | 0x34 | Authentication Required |
| PUDS_NRC_IK | 0x35 | Invalid Key |
| PUDS_NRC_ENOA | 0x36 | Exceeded Number Of Attempts |
| PUDS_NRC_RTDNE | 0x37 | Required Time Delay Not Expired |
| PUDS_NRC_SDTR | 0x38 | Secure Data Transmission Required |
| PUDS_NRC_SDTNA | 0x39 | Secure Data Transmission Not Allowed |
| PUDS_NRC_SDTF | 0x3A | Secure Data Verification Failed |
| PUDS_NRC_CVFITP | 0x50 | Certificate Verification Failed Invalid Time Period |
| PUDS_NRC_CVFISIG | 0x51 | Certificate Verification Failed Invalid SIGNature |
| PUDS_NRC_CVFICOT | 0x52 | Certificate Verification Failed Invalid Chain of Trust |
| PUDS_NRC_CVFIT | 0x53 | Certificate Verification Failed Invalid Type |
| PUDS_NRC_CVFIF | 0x54 | Certificate Verification Failed Invalid Format |
| PUDS_NRC_CVFIC | 0x55 | Certificate Verification Failed Invalid Content |
| PUDS_NRC_CVFISCP | 0x56 | Certificate Verification Failed Invalid ScoPe |
| PUDS_NRC_CVFICERT | 0x57 | Certificate Verification Failed Invalid CERTificate(revoked) |
| PUDS_NRC_OVF | 0x58 | Ownership Verification Failed |
| PUDS_NRC_CCF | 0x59 | Challenge Calculation Failed |
| PUDS_NRC_SARF | 0x5A | Setting Access Rights Failed |
| PUDS_NRC_SKCDF | 0x5B | Session Key Creation / Derivation Failed |
| PUDS_NRC_CDUF | 0x5C | Configuration Data Usage Failed |
| PUDS_NRC_DAF | 0x5D | DeAuthentication Failed |
| PUDS_NRC_UDNA | 0x70 | Upload Download Not Accepted |
| PUDS_NRC_TDS | 0x71 | Transfer Data Suspended |
| PUDS_NRC_GPF | 0x72 | General Programming Failure |
| PUDS_NRC_WBSC | 0x73 | Wrong Block Sequence Counter |

| Name | Value | Description |
|------------------|-------|---|
| PUDS_NRC_RCRRP | 0x78 | Request Correctly Received – Response Pending |
| PUDS_NRC_SFNSIAS | 0x7E | Sub Function Not Supported In Active Session |
| PUDS_NRC_SNSIAS | 0x7F | Service Not Supported In Active Session |
| PUDS_NRC_RPMTH | 0x81 | RPM Too High |
| PUDS_NRC_RPMTL | 0x82 | RPM Too Low |
| PUDS_NRC_EIR | 0x83 | Engine Is Running |
| PUDS_NRC_EINR | 0x84 | Engine Is Not Running |
| PUDS_NRC_ERTTL | 0x85 | Engine Run Time Too Low |
| PUDS_NRC_TEMPTH | 0x86 | TEMPerature Too High |
| PUDS_NRC_TEMPTL | 0x87 | TEMPerature Too Low |
| PUDS_NRC_VSTH | 0x88 | Vehicle Speed Too High |
| PUDS_NRC_VSTL | 0x89 | Vehicle Speed Too Low |
| PUDS_NRC_TPTH | 0x8A | Throttle / Pedal Too High |
| PUDS_NRC_TPTL | 0x8B | Throttle / Pedal Too Low |
| PUDS_NRC_TRNIN | 0x8C | Transmission Range Not In Neutral |
| PUDS_NRC_TRNIG | 0x8D | Transmission Range Not In Gear |
| PUDS_NRC_BSNC | 0x8F | Brake Switch(es) Not Closed(brake pedal not pressed or not applied) |
| PUDS_NRC_SLNIP | 0x90 | Shifter Lever Not In Park |
| PUDS_NRC_TCCL | 0x91 | Torque Converter Clutch Locked |
| PUDS_NRC_VTH | 0x92 | Voltage Too High |
| PUDS_NRC_VTL | 0x93 | Voltage Too Low |
| PUDS_NRC_RTNA | 0x94 | Resource Temporarily Not Available |

See also: `uds_msgaccess` on page 28, `uds_msg` on page 21.

3.5.11 uds_svc_param_dsc

Represents the subfunction parameter for UDS service DiagnosticSessionControl. According to the programming language, this type can be a group of defined values or an enumeration.

Syntax

C/C++

```
#define PUDS_SVC_PARAM_DSC_DS      0x01
#define PUDS_SVC_PARAM_DSC_ECUPS   0x02
#define PUDS_SVC_PARAM_DSC_ECUEDS 0x03
#define PUDS_SVC_PARAM_DSC_SSDS   0x04
```

Pascal OO

```
uds_svc_param_dsc = (
    PUDS_SVC_PARAM_DSC_DS = $1,
    PUDS_SVC_PARAM_DSC_ECUPS = $2,
    PUDS_SVC_PARAM_DSC_ECUEDS = $3,
    PUDS_SVC_PARAM_DSC_SSDS = $4
);
```

C#

```
public enum uds_svc_param_dsc : Byte
{
    PUDS_SVC_PARAM_DSC_DS = 0x01,
    PUDS_SVC_PARAM_DSC_ECUPS = 0x02,
    PUDS_SVC_PARAM_DSC_ECUEDS = 0x03,
    PUDS_SVC_PARAM_DSC_SSDS = 0x04
}
```

C++ / CLR

```
enum struct uds_svc_param_dsc : Byte
{
    PUDS_SVC_PARAM_DSC_DS = 0x01,
    PUDS_SVC_PARAM_DSC_ECUPS = 0x02,
    PUDS_SVC_PARAM_DSC_ECUEDS = 0x03,
    PUDS_SVC_PARAM_DSC_SSDS = 0x04
};
```

Visual Basic

```
Public Enum uds_svc_param_dsc As Byte
    PUDS_SVC_PARAM_DSC_DS = &H1
    PUDS_SVC_PARAM_DSC_ECUPS = &H2
    PUDS_SVC_PARAM_DSC_ECUEDS = &H3
    PUDS_SVC_PARAM_DSC_SSDS = &H4
End Enum
```

Values

| Name | Value | Description |
|---------------------------|-------|-----------------------------------|
| PUDS_SVC_PARAM_DSC_DS | 1 | Default Session. |
| PUDS_SVC_PARAM_DSC_ECUPS | 2 | ECU Programming Session. |
| PUDS_SVC_PARAM_DSC_ECUEDS | 3 | ECU Extended Diagnostic Session. |
| PUDS_SVC_PARAM_DSC_SSDS | 4 | Safety System Diagnostic Session. |

See also: [UDS_SvcDiagnosticSessionControl_2013](#) on page 661 (**class-method:** [SvcDiagnosticSessionControl_2013](#) on page 258).

3.5.12 uds_svc_param_er

Represents the subfunction parameter for UDS service ECUReset. According to the programming language, this type can be a group of defined values or an enumeration.

Syntax**C/C++**

```
#define PUDS_SVC_PARAM_ER_HR 0x01
#define PUDS_SVC_PARAM_ER_KOFFONR 0x02
#define PUDS_SVC_PARAM_ER_SR 0x03
#define PUDS_SVC_PARAM_ER_ERPSD 0x04
#define PUDS_SVC_PARAM_ER_DRPSD 0x05
```


Pascal OO

```
uds_svc_param_er = (
  PUDS_SVC_PARAM_ER_HR = $1,
  PUDS_SVC_PARAM_ER_KOFFONR = $2,
  PUDS_SVC_PARAM_ER_SR = $3,
  PUDS_SVC_PARAM_ER_ERPSD = $4,
  PUDS_SVC_PARAM_ER_DRPSD = $5
);
```

C#

```
public enum uds_svc_param_er : Byte
{
  PUDS_SVC_PARAM_ER_HR = 0x01,
  PUDS_SVC_PARAM_ER_KOFFONR = 0x02,
  PUDS_SVC_PARAM_ER_SR = 0x03,
  PUDS_SVC_PARAM_ER_ERPSD = 0x04,
  PUDS_SVC_PARAM_ER_DRPSD = 0x05,
}
```

C++ / CLR

```
enum struct uds_svc_param_er : Byte
{
  PUDS_SVC_PARAM_ER_HR = 0x01,
  PUDS_SVC_PARAM_ER_KOFFONR = 0x02,
  PUDS_SVC_PARAM_ER_SR = 0x03,
  PUDS_SVC_PARAM_ER_ERPSD = 0x04,
  PUDS_SVC_PARAM_ER_DRPSD = 0x05,
};
```

Visual Basic

```
Public Enum uds_svc_param_er As Byte
  PUDS_SVC_PARAM_ER_HR = &H1
  PUDS_SVC_PARAM_ER_KOFFONR = &H2
  PUDS_SVC_PARAM_ER_SR = &H3
  PUDS_SVC_PARAM_ER_ERPSD = &H4
  PUDS_SVC_PARAM_ER_DRPSD = &H5
End Enum
```

Values

| Name | Value | Description |
|---------------------------|-------|-------------------------------|
| PUDS_SVC_PARAM_ER_HR | 1 | Hard Reset. |
| PUDS_SVC_PARAM_ER_KOFFONR | 2 | Key Off on Reset. |
| PUDS_SVC_PARAM_ER_SR | 3 | Soft Reset. |
| PUDS_SVC_PARAM_ER_ERPSD | 4 | Enable Rapid Power Shutdown. |
| PUDS_SVC_PARAM_ER_DRPSD | 5 | Disable Rapid Power Shutdown. |

See also: [UDS_SvcECUReset_2013](#) on page 663 (**class-method:** [SvcECUReset_2013](#) on page 262).

3.5.13 uds_svc_param_cc

Represents the subfunction parameter for UDS service CommunicationControl. According to the programming language, this type can be a group of defined values or an enumeration.

Syntax

C/C++

```
#define PUDS_SVC_PARAM_CC_ERXTX          0x00
#define PUDS_SVC_PARAM_CC_ERXDTX        0x01
#define PUDS_SVC_PARAM_CC_DRXETX        0x02
#define PUDS_SVC_PARAM_CC_DRXTX         0x03
#define PUDS_SVC_PARAM_CC_ERXDTXWEAI    0x04
#define PUDS_SVC_PARAM_CC_ERXTXWEAI     0x05
```

Pascal OO

```
uds_svc_param_cc = (
    PUDS_SVC_PARAM_CC_ERXTX = $0,
    PUDS_SVC_PARAM_CC_ERXDTX = $1,
    PUDS_SVC_PARAM_CC_DRXETX = $2,
    PUDS_SVC_PARAM_CC_DRXTX = $3,
    PUDS_SVC_PARAM_CC_ERXDTXWEAI = $4,
    PUDS_SVC_PARAM_CC_ERXTXWEAI = $5
);
```

C#

```
public enum uds_svc_param_cc : Byte
{
    PUDS_SVC_PARAM_CC_ERXTX = 0x00,
    PUDS_SVC_PARAM_CC_ERXDTX = 0x01,
    PUDS_SVC_PARAM_CC_DRXETX = 0x02,
    PUDS_SVC_PARAM_CC_DRXTX = 0x03,
    PUDS_SVC_PARAM_CC_ERXDTXWEAI = 0x04,
    PUDS_SVC_PARAM_CC_ERXTXWEAI = 0x05
}
```

C++ / CLR

```
enum struct uds_svc_param_cc : Byte
{
    PUDS_SVC_PARAM_CC_ERXTX = 0x00,
    PUDS_SVC_PARAM_CC_ERXDTX = 0x01,
    PUDS_SVC_PARAM_CC_DRXETX = 0x02,
    PUDS_SVC_PARAM_CC_DRXTX = 0x03,
    PUDS_SVC_PARAM_CC_ERXDTXWEAI = 0x04,
    PUDS_SVC_PARAM_CC_ERXTXWEAI = 0x05
};
```

Visual Basic

```
Public Enum uds_svc_param_cc As Byte
    PUDS_SVC_PARAM_CC_ERXTX = &H0
    PUDS_SVC_PARAM_CC_ERXDTX = &H1
    PUDS_SVC_PARAM_CC_DRXETX = &H2
    PUDS_SVC_PARAM_CC_DRXTX = &H3
    PUDS_SVC_PARAM_CC_ERXDTXWEAI = &H4
    PUDS_SVC_PARAM_CC_ERXTXWEAI = &H5
End Enum
```

Values

| Name | Value | Description |
|------------------------------|-------|---|
| PUDS_SVC_PARAM_CC_ERXTX | 0 | Enable Rx and Tx. |
| PUDS_SVC_PARAM_CC_ERXDTX | 1 | Enable Rx and Disable Tx. |
| PUDS_SVC_PARAM_CC_DRXETX | 2 | Disable Rx and Enable Tx. |
| PUDS_SVC_PARAM_CC_DRXTX | 3 | Disable Rx and Tx. |
| PUDS_SVC_PARAM_CC_ERXDTXWEAI | 4 | Enable Rx And Disable Tx With Enhanced Address Information. |
| PUDS_SVC_PARAM_CC_ERXTXWEAI | 5 | Enable Rx And Tx With Enhanced Address Information. |

See also: [UDS_SvcCommunicationControl_2013](#) on page 666 (**class-method:** [SvcCommunicationControl_2013](#) on page 276).

3.5.14 uds_svc_param_tp

Represents the subfunction parameter for UDS service TesterPresent. According to the programming language, this type can be a group of defined values or an enumeration.

Syntax

C/C++

```
#define PUDS_SVC_PARAM_TP_ZSUBF 0x00
```

Pascal OO

```
uds_svc_param_tp = (  
    PUDS_SVC_PARAM_TP_ZSUBF = $0  
);
```

C#

```
public enum uds_svc_param_tp : Byte  
{  
    PUDS_SVC_PARAM_TP_ZSUBF = 0x00  
}
```

C++ / CLR

```
enum struct uds_svc_param_tp : Byte  
{  
    PUDS_SVC_PARAM_TP_ZSUBF = 0x00  
};
```

Visual Basic

```
Public Enum uds_svc_param_tp As Byte  
    PUDS_SVC_PARAM_TP_ZSUBF = &H0  
End Enum
```

Values

| Name | Value | Description |
|-------------------------|-------|-------------------|
| PUDS_SVC_PARAM_TP_ZSUBF | 0 | Zero subfunction. |

See also: [UDS_SvcTesterPresent_2013](#) on page 668 (**class-method:** [SvcTesterPresent_2013](#) on page 285).

3.5.15 uds_svc_param_cdtcs

Represents the subfunction parameter for UDS service ControlDTCSetting. According to the programming language, this type can be a group of defined values or an enumeration.

Syntax

C/C++

```
#define PUDS_SVC_PARAM_CDTCS_ON      0x01
#define PUDS_SVC_PARAM_CDTCS_OFF    0x02
```

Pascal OO

```
uds_svc_param_cdtcs = (
    PUDS_SVC_PARAM_CDTCS_ON = $1,
    PUDS_SVC_PARAM_CDTCS_OFF = $2
);
```

C#

```
public enum uds_svc_param_cdtcs : Byte
{
    PUDS_SVC_PARAM_CDTCS_ON = 0x01,
    PUDS_SVC_PARAM_CDTCS_OFF = 0x02
}
```

C++ / CLR

```
enum struct uds_svc_param_cdtcs : Byte
{
    PUDS_SVC_PARAM_CDTCS_ON = 0x01,
    PUDS_SVC_PARAM_CDTCS_OFF = 0x02
};
```

Visual Basic

```
Public Enum uds_svc_param_cdtcs As Byte
    PUDS_SVC_PARAM_CDTCS_ON = &H1
    PUDS_SVC_PARAM_CDTCS_OFF = &H2
End Enum
```

Values

| Name | Value | Description |
|--------------------------|-------|--|
| PUDS_SVC_PARAM_CDTCS_ON | 1 | The server(s)/ECU(s) shall resume the setting of diagnostic trouble codes. |
| PUDS_SVC_PARAM_CDTCS_OFF | 2 | The server(s)/ECU(s) shall stop the setting of diagnostic trouble codes. |

See also: [UDS_SvcControlDTCSetting_2013](#) on page 675 (**class-method:** [SvcControlDTCSetting_2013](#) on page 305).

3.5.16 uds_svc_param_roe

Represents the subfunction parameter for UDS service ResponseOnEvent. According to the programming language, this type can be a group of defined values or an enumeration.

Syntax

C/C++

```
#define PUDS_SVC_PARAM_ROE_STPROE      0x00
#define PUDS_SVC_PARAM_ROE_ONDTCS     0x01
#define PUDS_SVC_PARAM_ROE_OTI       0x02
#define PUDS_SVC_PARAM_ROE_OCODID    0x03
#define PUDS_SVC_PARAM_ROE_RAE       0x04
#define PUDS_SVC_PARAM_ROE_STRTROE   0x05
#define PUDS_SVC_PARAM_ROE_CLRROE    0x06
#define PUDS_SVC_PARAM_ROE_OCOV      0x07
#define PUDS_SVC_PARAM_ROE_RMRDOSC   0x08
#define PUDS_SVC_PARAM_ROE_RDRIODSC  0x09
```

Pascal OO

```
uds_svc_param_roe = (
  PUDS_SVC_PARAM_ROE_STPROE = $0,
  PUDS_SVC_PARAM_ROE_ONDTCS = $1,
  PUDS_SVC_PARAM_ROE_OTI = $2,
  PUDS_SVC_PARAM_ROE_OCODID = $3,
  PUDS_SVC_PARAM_ROE_RAE = $4,
  PUDS_SVC_PARAM_ROE_STRTROE = $5,
  PUDS_SVC_PARAM_ROE_CLRROE = $6,
  PUDS_SVC_PARAM_ROE_OCOV = $7,
  PUDS_SVC_PARAM_ROE_RMRDOSC = $8,
  PUDS_SVC_PARAM_ROE_RDRIODSC = $9
);
```

C#

```
public enum uds_svc_param_roe : Byte
{
  PUDS_SVC_PARAM_ROE_STPROE = 0x00,
  PUDS_SVC_PARAM_ROE_ONDTCS = 0x01,
  PUDS_SVC_PARAM_ROE_OTI = 0x02,
  PUDS_SVC_PARAM_ROE_OCODID = 0x03,
  PUDS_SVC_PARAM_ROE_RAE = 0x04,
  PUDS_SVC_PARAM_ROE_STRTROE = 0x05,
  PUDS_SVC_PARAM_ROE_CLRROE = 0x06,
  PUDS_SVC_PARAM_ROE_OCOV = 0x07,
  PUDS_SVC_PARAM_ROE_RMRDOSC = 0x08,
  PUDS_SVC_PARAM_ROE_RDRIODSC = 0x09
}
```

C++ / CLR

```
enum struct uds_svc_param_roe : Byte
{
  PUDS_SVC_PARAM_ROE_STPROE = 0x00,
  PUDS_SVC_PARAM_ROE_ONDTCS = 0x01,
  PUDS_SVC_PARAM_ROE_OTI = 0x02,
  PUDS_SVC_PARAM_ROE_OCODID = 0x03,
  PUDS_SVC_PARAM_ROE_RAE = 0x04,
  PUDS_SVC_PARAM_ROE_STRTROE = 0x05,
  PUDS_SVC_PARAM_ROE_CLRROE = 0x06,
  PUDS_SVC_PARAM_ROE_OCOV = 0x07,
```

```

PUDS_SVC_PARAM_ROE_RMRDOSC = 0x08,
PUDS_SVC_PARAM_ROE_RDRIODSC = 0x09
};

```

Visual Basic

```

Public Enum uds_svc_param_roe As Byte
    PUDS_SVC_PARAM_ROE_STPROE = &H0
    PUDS_SVC_PARAM_ROE_ONDTCS = &H1
    PUDS_SVC_PARAM_ROE_OTI = &H2
    PUDS_SVC_PARAM_ROE_OCODID = &H3
    PUDS_SVC_PARAM_ROE_RAE = &H4
    PUDS_SVC_PARAM_ROE_STRTROE = &H5
    PUDS_SVC_PARAM_ROE_CLRROE = &H6
    PUDS_SVC_PARAM_ROE_OCOV = &H7
    PUDS_SVC_PARAM_ROE_RMRDOSC = &H8
    PUDS_SVC_PARAM_ROE_RDRIODSC = &H9
End Enum

```

values

| Name | Value | Description |
|-----------------------------|-------|---|
| PUDS_SVC_PARAM_ROE_STPROE | 0 | Stop Response On Event. |
| PUDS_SVC_PARAM_ROE_ONDTCS | 1 | On DTC Status Change. |
| PUDS_SVC_PARAM_ROE_OTI | 2 | On Timer Interrupt. |
| PUDS_SVC_PARAM_ROE_OCODID | 3 | On Change Of Data Identifier. |
| PUDS_SVC_PARAM_ROE_RAE | 4 | Report Activated Events. |
| PUDS_SVC_PARAM_ROE_STRTROE | 5 | Start Response On Event. |
| PUDS_SVC_PARAM_ROE_CLRROE | 6 | Clear Response On Event. |
| PUDS_SVC_PARAM_ROE_OCOV | 7 | On Comparison Of Values. |
| PUDS_SVC_PARAM_ROE_RMRDOSC | 8 | Report Most Recent Dtc On Status Change (ISO 14229-1:2020) |
| PUDS_SVC_PARAM_ROE_RDRIODSC | 9 | Report Dtc Record Information On Dtc Status Change (ISO 14229-1:2020) |

See also: [UDS_SvcResponseOnEvent_2013](#) on page 677 (class-method: [SvcResponseOnEvent_2013](#) on page 314).

3.5.17 uds_svc_param_roe_recommended_service_id

Represents the recommended service to use with the UDS service ResponseOnEvent. According to the programming language, this type can be a group of defined values or an enumeration.

Syntax

C/C++

```

#define PUDS_SVC_PARAM_ROE_STRT_SI_RDBI      PUDS_SI_ReadDataByIdentifier
#define PUDS_SVC_PARAM_ROE_STRT_SI_RDTCI     PUDS_SI_ReadDTCInformation
#define PUDS_SVC_PARAM_ROE_STRT_SI_RC        PUDS_SI_RoutineControl
#define PUDS_SVC_PARAM_ROE_STRT_SI_IOCBI     PUDS_SI_InputOutputControlByIdentifier

```

Pascal OO

```

uds_svc_param_roe_recommended_service_id = (
    PUDS_SVC_PARAM_ROE_STRT_SI_RDBI = Byte(PUDS_SERVICE_SI_ReadDataByIdentifier),
    PUDS_SVC_PARAM_ROE_STRT_SI_RDTCI = Byte(PUDS_SERVICE_SI_ReadDTCInformation),
    PUDS_SVC_PARAM_ROE_STRT_SI_RC = Byte(PUDS_SERVICE_SI_RoutineControl),
    PUDS_SVC_PARAM_ROE_STRT_SI_IOCBI = Byte(PUDS_SERVICE_SI_InputOutputControlByIdentifier)
);

```

C#

```
enum uds_svc_param_roe_recommended_service_id : Byte
{
    PUDS_SVC_PARAM_ROE_STRT_SI_RDBI = uds_service.PUDS_SERVICE_SI_ReadDataByIdentifier,
    PUDS_SVC_PARAM_ROE_STRT_SI_RDTCI = uds_service.PUDS_SERVICE_SI_ReadDTCInformation,
    PUDS_SVC_PARAM_ROE_STRT_SI_RC = uds_service.PUDS_SERVICE_SI_RoutineControl,
    PUDS_SVC_PARAM_ROE_STRT_SI_IOCBI = uds_service.PUDS_SERVICE_SI_InputOutputControlByIdentifier
}
```

C++ / CLR

```
enum struct uds_svc_param_roe_recommended_service_id : Byte
{
    PUDS_SVC_PARAM_ROE_STRT_SI_RDBI = uds_service::PUDS_SERVICE_SI_ReadDataByIdentifier,
    PUDS_SVC_PARAM_ROE_STRT_SI_RDTCI = uds_service::PUDS_SERVICE_SI_ReadDTCInformation,
    PUDS_SVC_PARAM_ROE_STRT_SI_RC = uds_service::PUDS_SERVICE_SI_RoutineControl,
    PUDS_SVC_PARAM_ROE_STRT_SI_IOCBI = uds_service::PUDS_SERVICE_SI_InputOutputControlByIdentifier
};
```

Visual Basic

```
Enum uds_svc_param_roe_recommended_service_id As Byte
    PUDS_SVC_PARAM_ROE_STRT_SI_RDBI = uds_service.PUDS_SERVICE_SI_ReadDataByIdentifier
    PUDS_SVC_PARAM_ROE_STRT_SI_RDTCI = uds_service.PUDS_SERVICE_SI_ReadDTCInformation
    PUDS_SVC_PARAM_ROE_STRT_SI_RC = uds_service.PUDS_SERVICE_SI_RoutineControl
    PUDS_SVC_PARAM_ROE_STRT_SI_IOCBI = uds_service.PUDS_SERVICE_SI_InputOutputControlByIdentifier
End Enum
```

values

| Name | Value | Description |
|----------------------------------|--|---|
| PUDS_SVC_PARAM_ROE_STRT_SI_RDBI | PUDS_SI_ReadDataByIdentifier | UDS service ReadDataByIdentifier. |
| PUDS_SVC_PARAM_ROE_STRT_SI_RDTCI | PUDS_SI_ReadDTCInformation | UDS service ReadDTCInformation. |
| PUDS_SVC_PARAM_ROE_STRT_SI_RC | PUDS_SI_RoutineControl | UDS service RoutineControl. |
| PUDS_SVC_PARAM_ROE_STRT_SI_IOCBI | PUDS_SI_InputOutputControlByIdentifier | UDS service InputOutputControlByIdentifier. |

See also: `UDS_SvcResponseOnEvent_2013` on page 677 (**class-method:** `SvcResponseOnEvent_2013` on page 314), `uds_service` on page 53.

3.5.18 uds_svc_param_lc

Represents the subfunction parameter for UDS service LinkControl. According to the programming language, this type can be a group of defined values or an enumeration.

Syntax**C/C++**

```
#define PUDS_SVC_PARAM_LC_VBTWFBR 0x01
#define PUDS_SVC_PARAM_LC_VBTWSBR 0x02
#define PUDS_SVC_PARAM_LC_TB 0x03
```

Pascal OO

```
uds_svc_param_lc = (
    PUDS_SVC_PARAM_LC_VBTWFBR = $1,
    PUDS_SVC_PARAM_LC_VBTWSBR = $2,
    PUDS_SVC_PARAM_LC_TB = $3
);
```

C#

```
public enum uds_svc_param_lc : Byte
{
    PUDS_SVC_PARAM_LC_VBTWFBR = 0x01,
    PUDS_SVC_PARAM_LC_VBTWSBR = 0x02,
    PUDS_SVC_PARAM_LC_TB = 0x03
}
```

C++ / CLR

```
enum struct uds_svc_param_lc : Byte
{
    PUDS_SVC_PARAM_LC_VBTWFBR = 0x01,
    PUDS_SVC_PARAM_LC_VBTWSBR = 0x02,
    PUDS_SVC_PARAM_LC_TB = 0x03
};
```

Visual Basic

```
Public Enum uds_svc_param_lc As Byte
    PUDS_SVC_PARAM_LC_VBTWFBR = &H1
    PUDS_SVC_PARAM_LC_VBTWSBR = &H2
    PUDS_SVC_PARAM_LC_TB = &H3
End Enum
```

Values

| Name | Value | Description |
|---------------------------|-------|--|
| PUDS_SVC_PARAM_LC_VBTWFBR | 1 | Verify Baud rate Transition With Fixed Baud rate. |
| PUDS_SVC_PARAM_LC_VBTWSBR | 2 | Verify Baud rate Transition With Specific Baud rate. |
| PUDS_SVC_PARAM_LC_TB | 3 | Transition Baud rate. |

See also: [UDS_SvcLinkControl_2013](#) on page 679 (**class-method:** [SvcLinkControl_2013](#) on page 328).

3.5.19 uds_svc_param_lc_baudrate_identifier

Represents the standard baud rate identifiers to use with the UDS service LinkControl. According to the programming language, this type can be a group of defined values or an enumeration.

Syntax**C/C++**

```
#define PUDS_SVC_PARAM_LC_BAUDRATE_PC_9600 0x01
#define PUDS_SVC_PARAM_LC_BAUDRATE_PC_19200 0x02
#define PUDS_SVC_PARAM_LC_BAUDRATE_PC_38400 0x03
#define PUDS_SVC_PARAM_LC_BAUDRATE_PC_57600 0x04
#define PUDS_SVC_PARAM_LC_BAUDRATE_PC_115200 0x05
#define PUDS_SVC_PARAM_LC_BAUDRATE_CAN_125K 0x10
#define PUDS_SVC_PARAM_LC_BAUDRATE_CAN_250K 0x11
#define PUDS_SVC_PARAM_LC_BAUDRATE_CAN_500K 0x12
#define PUDS_SVC_PARAM_LC_BAUDRATE_CAN_1M 0x13
#define PUDS_SVC_PARAM_LC_BAUDRATE_PROGSU 0x20
```

Pascal OO

```
uds_svc_param_lc_baudrate_identifier = (
    PUDS_SVC_PARAM_LC_BAUDRATE_PC_9600 = $1,
    PUDS_SVC_PARAM_LC_BAUDRATE_PC_19200 = $2,
```



```

PUDS_SVC_PARAM_LC_BAUDRATE_PC_38400 = $3,
PUDS_SVC_PARAM_LC_BAUDRATE_PC_57600 = $4,
PUDS_SVC_PARAM_LC_BAUDRATE_PC_115200 = $5,
PUDS_SVC_PARAM_LC_BAUDRATE_CAN_125K = $10,
PUDS_SVC_PARAM_LC_BAUDRATE_CAN_250K = $11,
PUDS_SVC_PARAM_LC_BAUDRATE_CAN_500K = $12,
PUDS_SVC_PARAM_LC_BAUDRATE_CAN_1M = $13,
PUDS_SVC_PARAM_LC_BAUDRATE_PROGSU = $20
);

```

C#

```

public enum uds_svc_param_lc_baudrate_identifier : Byte
{
    PUDS_SVC_PARAM_LC_BAUDRATE_PC_9600 = 0x01,
    PUDS_SVC_PARAM_LC_BAUDRATE_PC_19200 = 0x02,
    PUDS_SVC_PARAM_LC_BAUDRATE_PC_38400 = 0x03,
    PUDS_SVC_PARAM_LC_BAUDRATE_PC_57600 = 0x04,
    PUDS_SVC_PARAM_LC_BAUDRATE_PC_115200 = 0x05,
    PUDS_SVC_PARAM_LC_BAUDRATE_CAN_125K = 0x10,
    PUDS_SVC_PARAM_LC_BAUDRATE_CAN_250K = 0x11,
    PUDS_SVC_PARAM_LC_BAUDRATE_CAN_500K = 0x12,
    PUDS_SVC_PARAM_LC_BAUDRATE_CAN_1M = 0x13,
    PUDS_SVC_PARAM_LC_BAUDRATE_PROGSU = 0x20
}

```

C++ / CLR

```

enum struct uds_svc_param_lc_baudrate_identifier : Byte
{
    PUDS_SVC_PARAM_LC_BAUDRATE_PC_9600 = 0x01,
    PUDS_SVC_PARAM_LC_BAUDRATE_PC_19200 = 0x02,
    PUDS_SVC_PARAM_LC_BAUDRATE_PC_38400 = 0x03,
    PUDS_SVC_PARAM_LC_BAUDRATE_PC_57600 = 0x04,
    PUDS_SVC_PARAM_LC_BAUDRATE_PC_115200 = 0x05,
    PUDS_SVC_PARAM_LC_BAUDRATE_CAN_125K = 0x10,
    PUDS_SVC_PARAM_LC_BAUDRATE_CAN_250K = 0x11,
    PUDS_SVC_PARAM_LC_BAUDRATE_CAN_500K = 0x12,
    PUDS_SVC_PARAM_LC_BAUDRATE_CAN_1M = 0x13,
    PUDS_SVC_PARAM_LC_BAUDRATE_PROGSU = 0x20
};

```

Visual Basic

```

Public Enum uds_svc_param_lc_baudrate_identifier As Byte
    PUDS_SVC_PARAM_LC_BAUDRATE_PC_9600 = &H1
    PUDS_SVC_PARAM_LC_BAUDRATE_PC_19200 = &H2
    PUDS_SVC_PARAM_LC_BAUDRATE_PC_38400 = &H3
    PUDS_SVC_PARAM_LC_BAUDRATE_PC_57600 = &H4
    PUDS_SVC_PARAM_LC_BAUDRATE_PC_115200 = &H5
    PUDS_SVC_PARAM_LC_BAUDRATE_CAN_125K = &H10
    PUDS_SVC_PARAM_LC_BAUDRATE_CAN_250K = &H11
    PUDS_SVC_PARAM_LC_BAUDRATE_CAN_500K = &H12
    PUDS_SVC_PARAM_LC_BAUDRATE_CAN_1M = &H13
    PUDS_SVC_PARAM_LC_BAUDRATE_PROGSU = &H20
End Enum

```

Values

| Name | Value | Description |
|-------------------------------------|---------|--------------------------------------|
| PUDS_SVC_PARAM_LC_BAUDRATE_PC_9600 | 0x1 (1) | Standard PC baud rate of 9.6 Kbaud. |
| PUDS_SVC_PARAM_LC_BAUDRATE_PC_19200 | 0x2 (2) | Standard PC baud rate of 19.2 Kbaud. |

| Name | Value | Description |
|--------------------------------------|-----------|---------------------------------------|
| PUDS_SVC_PARAM_LC_BAUDRATE_PC_38400 | 0x3 (3) | Standard PC baud rate of 38.4 KBaud. |
| PUDS_SVC_PARAM_LC_BAUDRATE_PC_57600 | 0x4 (4) | Standard PC baud rate of 57.6 KBaud. |
| PUDS_SVC_PARAM_LC_BAUDRATE_PC_115200 | 0x5 (5) | Standard PC baud rate of 115.2 KBaud. |
| PUDS_SVC_PARAM_LC_BAUDRATE_CAN_125K | 0x10 (16) | Standard CAN baud rate of 125 KBaud. |
| PUDS_SVC_PARAM_LC_BAUDRATE_CAN_250K | 0x11 (17) | Standard CAN baud rate of 250 KBaud. |
| PUDS_SVC_PARAM_LC_BAUDRATE_CAN_500K | 0x12 (18) | Standard CAN baud rate of 500 KBaud. |
| PUDS_SVC_PARAM_LC_BAUDRATE_CAN_1M | 0x13 (19) | Standard CAN baud rate of 1 MBaud. |
| PUDS_SVC_PARAM_LC_BAUDRATE_PROGSU | 0x20 (32) | Programming setup |

See also: `UDS_SvcLinkControl_2013` on page 679 (class-method: `SvcLinkControl_2013` on page 328).

3.5.20 uds_svc_param_di

Represents the Data Identifiers defined in UDS standard ISO-14229-1. According to the programming language, this type can be a group of defined values or an enumeration.

Syntax

C/C++

```
#define PUDS_SVC_PARAM_DI_BSIDIID 0xF180
#define PUDS_SVC_PARAM_DI_ASIDIID 0xF181
#define PUDS_SVC_PARAM_DI_ADIDIID 0xF182
#define PUDS_SVC_PARAM_DI_BSFPDIID 0xF183
#define PUDS_SVC_PARAM_DI_ASFPDIID 0xF184
#define PUDS_SVC_PARAM_DI_ADFPDIID 0xF185
#define PUDS_SVC_PARAM_DI_ADSDIID 0xF186
#define PUDS_SVC_PARAM_DI_VMSPNDIID 0xF187
#define PUDS_SVC_PARAM_DI_VMECUSNDIID 0xF188
#define PUDS_SVC_PARAM_DI_VMECUSVNDIID 0xF189
#define PUDS_SVC_PARAM_DI_SSIDIID 0xF18A
#define PUDS_SVC_PARAM_DI_ECUMDDIID 0xF18B
#define PUDS_SVC_PARAM_DI_ECUSNDIID 0xF18C
#define PUDS_SVC_PARAM_DI_SFUDIID 0xF18D
#define PUDS_SVC_PARAM_DI_VMKAPNDIID 0xF18E
#define PUDS_SVC_PARAM_DI_VINDIID 0xF190
#define PUDS_SVC_PARAM_DI_VMECUHNDIID 0xF191
#define PUDS_SVC_PARAM_DI_SSECUHWNDIID 0xF192
#define PUDS_SVC_PARAM_DI_SSECUHWVNDIID 0xF193
#define PUDS_SVC_PARAM_DI_SSECUSWNDIID 0xF194
#define PUDS_SVC_PARAM_DI_SSECUSWVNDIID 0xF195
#define PUDS_SVC_PARAM_DI_EROTANDIID 0xF196
#define PUDS_SVC_PARAM_DI_SNOETDIID 0xF197
#define PUDS_SVC_PARAM_DI_RSCOTSNDIID 0xF198
#define PUDS_SVC_PARAM_DI_PDDIID 0xF199
#define PUDS_SVC_PARAM_DI_CRSCOCESNDIID 0xF19A
#define PUDS_SVC_PARAM_DI_CDDIID 0xF19B
#define PUDS_SVC_PARAM_DI_CESWNDIID 0xF19C
#define PUDS_SVC_PARAM_DI_EIDIID 0xF19D
#define PUDS_SVC_PARAM_DI_ODXFDIID 0xF19E
#define PUDS_SVC_PARAM_DI_EDIID 0xF19F
```

Pascal OO

```
uds_svc_param_di = (
    PUDS_SVC_PARAM_DI_BSIDIID = $F180,
    PUDS_SVC_PARAM_DI_ASIDIID = $F181,
    PUDS_SVC_PARAM_DI_ADIDIID = $F182,
    PUDS_SVC_PARAM_DI_BSFPDIID = $F183,
```

```

PUDS_SVC_PARAM_DI_ASFPDID = $F184,
PUDS_SVC_PARAM_DI_ADFPDID = $F185,
PUDS_SVC_PARAM_DI_ADSDID = $F186,
PUDS_SVC_PARAM_DI_VMSPNDID = $F187,
PUDS_SVC_PARAM_DI_VMECUSNDID = $F188,
PUDS_SVC_PARAM_DI_VMECUSVNDID = $F189,
PUDS_SVC_PARAM_DI_SSIDDID = $F18A,
PUDS_SVC_PARAM_DI_ECUMDDID = $F18B,
PUDS_SVC_PARAM_DI_ECUSNDID = $F18C,
PUDS_SVC_PARAM_DI_SFUDID = $F18D,
PUDS_SVC_PARAM_DI_VMKAPNDID = $F18E,
PUDS_SVC_PARAM_DI_VINDID = $F190,
PUDS_SVC_PARAM_DI_VMECUHNDID = $F191,
PUDS_SVC_PARAM_DI_SSECUHWNDID = $F192,
PUDS_SVC_PARAM_DI_SSECUHWVNDID = $F193,
PUDS_SVC_PARAM_DI_SSECUSWNDID = $F194,
PUDS_SVC_PARAM_DI_SSECUSWVNDID = $F195,
PUDS_SVC_PARAM_DI_EROTANDID = $F196,
PUDS_SVC_PARAM_DI_SNOETDID = $F197,
PUDS_SVC_PARAM_DI_RSCOTSNDID = $F198,
PUDS_SVC_PARAM_DI_PDDID = $F199,
PUDS_SVC_PARAM_DI_CRSCOCESNDID = $F19A,
PUDS_SVC_PARAM_DI_CDDID = $F19B,
PUDS_SVC_PARAM_DI_CESWNDID = $F19C,
PUDS_SVC_PARAM_DI_EIDDID = $F19D,
PUDS_SVC_PARAM_DI_ODXFDID = $F19E,
PUDS_SVC_PARAM_DI_EDID = $F19F

```

```
);
```

C#

```

public enum uds_svc_param_di : UInt16
{
    PUDS_SVC_PARAM_DI_BSIDID = 0xF180,
    PUDS_SVC_PARAM_DI_ASIDID = 0xF181,
    PUDS_SVC_PARAM_DI_ADIDID = 0xF182,
    PUDS_SVC_PARAM_DI_BSPFDID = 0xF183,
    PUDS_SVC_PARAM_DI_ASFPDID = 0xF184,
    PUDS_SVC_PARAM_DI_ADFPDID = 0xF185,
    PUDS_SVC_PARAM_DI_ADSDID = 0xF186,
    PUDS_SVC_PARAM_DI_VMSPNDID = 0xF187,
    PUDS_SVC_PARAM_DI_VMECUSNDID = 0xF188,
    PUDS_SVC_PARAM_DI_VMECUSVNDID = 0xF189,
    PUDS_SVC_PARAM_DI_SSIDDID = 0xF18A,
    PUDS_SVC_PARAM_DI_ECUMDDID = 0xF18B,
    PUDS_SVC_PARAM_DI_ECUSNDID = 0xF18C,
    PUDS_SVC_PARAM_DI_SFUDID = 0xF18D,
    PUDS_SVC_PARAM_DI_VMKAPNDID = 0xF18E,
    PUDS_SVC_PARAM_DI_VINDID = 0xF190,
    PUDS_SVC_PARAM_DI_VMECUHNDID = 0xF191,
    PUDS_SVC_PARAM_DI_SSECUHWNDID = 0xF192,
    PUDS_SVC_PARAM_DI_SSECUHWVNDID = 0xF193,
    PUDS_SVC_PARAM_DI_SSECUSWNDID = 0xF194,
    PUDS_SVC_PARAM_DI_SSECUSWVNDID = 0xF195,
    PUDS_SVC_PARAM_DI_EROTANDID = 0xF196,
    PUDS_SVC_PARAM_DI_SNOETDID = 0xF197,
    PUDS_SVC_PARAM_DI_RSCOTSNDID = 0xF198,
    PUDS_SVC_PARAM_DI_PDDID = 0xF199,
    PUDS_SVC_PARAM_DI_CRSCOCESNDID = 0xF19A,
    PUDS_SVC_PARAM_DI_CDDID = 0xF19B,
    PUDS_SVC_PARAM_DI_CESWNDID = 0xF19C,
    PUDS_SVC_PARAM_DI_EIDDID = 0xF19D,

```

```

PUDS_SVC_PARAM_DI_ODXFDID = 0xF19E,
PUDS_SVC_PARAM_DI_EDID = 0xF19F
}

```

C++ / CLR

```

enum struct uds_svc_param_di : Uint16
{
    PUDS_SVC_PARAM_DI_BSIDDID = 0xF180,
    PUDS_SVC_PARAM_DI_ASIDDID = 0xF181,
    PUDS_SVC_PARAM_DI_ADIDDID = 0xF182,
    PUDS_SVC_PARAM_DI_BSFPDID = 0xF183,
    PUDS_SVC_PARAM_DI_ASFPDID = 0xF184,
    PUDS_SVC_PARAM_DI_ADFPDID = 0xF185,
    PUDS_SVC_PARAM_DI_ADSDID = 0xF186,
    PUDS_SVC_PARAM_DI_VMSPNDID = 0xF187,
    PUDS_SVC_PARAM_DI_VMECUSNDID = 0xF188,
    PUDS_SVC_PARAM_DI_VMECUSVNDID = 0xF189,
    PUDS_SVC_PARAM_DI_SSIDDID = 0xF18A,
    PUDS_SVC_PARAM_DI_ECUMDDID = 0xF18B,
    PUDS_SVC_PARAM_DI_ECUSNDID = 0xF18C,
    PUDS_SVC_PARAM_DI_SFUDID = 0xF18D,
    PUDS_SVC_PARAM_DI_VMKAPNDID = 0xF18E,
    PUDS_SVC_PARAM_DI_VINDID = 0xF190,
    PUDS_SVC_PARAM_DI_VMECUHNDID = 0xF191,
    PUDS_SVC_PARAM_DI_SSECUHWNDID = 0xF192,
    PUDS_SVC_PARAM_DI_SSECUHWVNDID = 0xF193,
    PUDS_SVC_PARAM_DI_SSECUSWNDID = 0xF194,
    PUDS_SVC_PARAM_DI_SSECUSWVNDID = 0xF195,
    PUDS_SVC_PARAM_DI_EROTANDID = 0xF196,
    PUDS_SVC_PARAM_DI_SNOETDID = 0xF197,
    PUDS_SVC_PARAM_DI_RSCOTSNDID = 0xF198,
    PUDS_SVC_PARAM_DI_PDDID = 0xF199,
    PUDS_SVC_PARAM_DI_CRSCOCESNDID = 0xF19A,
    PUDS_SVC_PARAM_DI_CDDID = 0xF19B,
    PUDS_SVC_PARAM_DI_CESWNDID = 0xF19C,
    PUDS_SVC_PARAM_DI_EIDDID = 0xF19D,
    PUDS_SVC_PARAM_DI_ODXFDID = 0xF19E,
    PUDS_SVC_PARAM_DI_EDID = 0xF19F
};

```

Visual Basic

```

Public Enum uds_svc_param_di As Uint16
    PUDS_SVC_PARAM_DI_BSIDDID = &HF180
    PUDS_SVC_PARAM_DI_ASIDDID = &HF181
    PUDS_SVC_PARAM_DI_ADIDDID = &HF182
    PUDS_SVC_PARAM_DI_BSFPDID = &HF183
    PUDS_SVC_PARAM_DI_ASFPDID = &HF184
    PUDS_SVC_PARAM_DI_ADFPDID = &HF185
    PUDS_SVC_PARAM_DI_ADSDID = &HF186
    PUDS_SVC_PARAM_DI_VMSPNDID = &HF187
    PUDS_SVC_PARAM_DI_VMECUSNDID = &HF188
    PUDS_SVC_PARAM_DI_VMECUSVNDID = &HF189
    PUDS_SVC_PARAM_DI_SSIDDID = &HF18A
    PUDS_SVC_PARAM_DI_ECUMDDID = &HF18B
    PUDS_SVC_PARAM_DI_ECUSNDID = &HF18C
    PUDS_SVC_PARAM_DI_SFUDID = &HF18D
    PUDS_SVC_PARAM_DI_VMKAPNDID = &HF18E
    PUDS_SVC_PARAM_DI_VINDID = &HF190
    PUDS_SVC_PARAM_DI_VMECUHNDID = &HF191
    PUDS_SVC_PARAM_DI_SSECUHWNDID = &HF192

```

```

PUDS_SVC_PARAM_DI_SSECUHWVNDID = &HF193
PUDS_SVC_PARAM_DI_SSECUSWNDID = &HF194
PUDS_SVC_PARAM_DI_SSECUSWVNDID = &HF195
PUDS_SVC_PARAM_DI_EROTANDID = &HF196
PUDS_SVC_PARAM_DI_SNOETDID = &HF197
PUDS_SVC_PARAM_DI_RSCOTSNDID = &HF198
PUDS_SVC_PARAM_DI_PDDID = &HF199
PUDS_SVC_PARAM_DI_CRSCOCESNDID = &HF19A
PUDS_SVC_PARAM_DI_CDDID = &HF19B
PUDS_SVC_PARAM_DI_CESWNDID = &HF19C
PUDS_SVC_PARAM_DI_EIDDDID = &HF19D
PUDS_SVC_PARAM_DI_ODXFDID = &HF19E
PUDS_SVC_PARAM_DI_EDID = &HF19F

```

End Enum

values

| Name | Value | Description |
|--------------------------------|--------|--|
| PUDS_SVC_PARAM_DI_BSIDID | 0xF180 | Boot Software Identification Data Identifier. |
| PUDS_SVC_PARAM_DI_ASIDID | 0xF181 | Application Software Identification Data Identifier. |
| PUDS_SVC_PARAM_DI_ADIDID | 0xF182 | Application Data Identification Data Identifier. |
| PUDS_SVC_PARAM_DI_BSFPDID | 0xF183 | Boot Software Identification Data Identifier. |
| PUDS_SVC_PARAM_DI_ASFPDID | 0xF184 | Application Software Fingerprint Data Identifier. |
| PUDS_SVC_PARAM_DI_ADFPDID | 0xF185 | Application Data Fingerprint Data Identifier. |
| PUDS_SVC_PARAM_DI_ADSDID | 0xF186 | Active Diagnostic Session Data Identifier. |
| PUDS_SVC_PARAM_DI_VMSPNDID | 0xF187 | Vehicle Manufacturer Spare Part Number Data Identifier. |
| PUDS_SVC_PARAM_DI_VMECUSNDID | 0xF188 | Vehicle Manufacturer ECU Software Number Data Identifier. |
| PUDS_SVC_PARAM_DI_VMECUSVNDID | 0xF189 | Vehicle Manufacturer ECU Software Version Number Data Identifier. |
| PUDS_SVC_PARAM_DI_SSIDDDID | 0xF18A | System Supplier Identifier Data Identifier. |
| PUDS_SVC_PARAM_DI_ECUMDDID | 0xF18B | ECU Manufacturing Date Data Identifier. |
| PUDS_SVC_PARAM_DI_ECUSNDID | 0xF18C | ECU Serial Number Data Identifier. |
| PUDS_SVC_PARAM_DI_SFUDID | 0xF18D | Supported Functional Units Data Identifier. |
| PUDS_SVC_PARAM_DI_VMKAPNDID | 0xF18E | Vehicle Manufacturer Kit Assembly Part Number Data Identifier. |
| PUDS_SVC_PARAM_DI_VINDID | 0xF190 | VIN Data Identifier. |
| PUDS_SVC_PARAM_DI_VMECUHNDID | 0xF191 | Vehicle Manufacturer ECU Hardware Number Data Identifier. |
| PUDS_SVC_PARAM_DI_SSECUHWNDID | 0xF192 | System Supplier ECU Hardware Number Data Identifier. |
| PUDS_SVC_PARAM_DI_SSECUHWVNDID | 0xF193 | System Supplier ECU Hardware Version Number Data Identifier. |
| PUDS_SVC_PARAM_DI_SSECUSWNDID | 0xF194 | System Supplier ECU Software Number Data Identifier. |
| PUDS_SVC_PARAM_DI_SSECUSWVNDID | 0xF195 | System Supplier ECU Software Version Number Data Identifier. |
| PUDS_SVC_PARAM_DI_EROTANDID | 0xF196 | Exhaust Regulation Or Type Approval Number Data Identifier. |
| PUDS_SVC_PARAM_DI_SNOETDID | 0xF197 | System Name Or Engine Type Data Identifier. |
| PUDS_SVC_PARAM_DI_RSCOTSNDID | 0xF198 | Repair Shop Code Or Tester Serial Number Data Identifier. |
| PUDS_SVC_PARAM_DI_PDDID | 0xF199 | Programming Date Data Identifier. |
| PUDS_SVC_PARAM_DI_CRSCOCESNDID | 0xF19A | Calibration Repair Shop Code Or Calibration Equipment Serial Number Data Identifier. |
| PUDS_SVC_PARAM_DI_CDDID | 0xF19B | Calibration Date Data Identifier. |
| PUDS_SVC_PARAM_DI_CESWNDID | 0xF19C | Calibration Equipment Software Number Data Identifier. |
| PUDS_SVC_PARAM_DI_EIDDDID | 0xF19D | ECU Installation Date Data Identifier. |
| PUDS_SVC_PARAM_DI_ODXFDID | 0xF19E | ODX File Data Identifier. |
| PUDS_SVC_PARAM_DI_EDID | 0xF19F | Entity Data Identifier. |

See also: [UDS_SvcReadDataByIdentifier_2013](#) on page 681 (**class-method:** [SvcReadDataByIdentifier_2013](#) on page 337).

3.5.21 uds_svc_param_rdbpi

Represents the subfunction parameter for UDS service ReadDataByPeriodicIdentifier. According to the programming language, this type can be a group of defined values or an enumeration.

Syntax

C/C++

```
#define PUDS_SVC_PARAM_RDBPI_SASR    0x01
#define PUDS_SVC_PARAM_RDBPI_SAMR    0x02
#define PUDS_SVC_PARAM_RDBPI_SAFR    0x03
#define PUDS_SVC_PARAM_RDBPI_SS      0x04
```

Pascal OO

```
uds_svc_param_rdbpi = (
    PUDS_SVC_PARAM_RDBPI_SASR = $1,
    PUDS_SVC_PARAM_RDBPI_SAMR = $2,
    PUDS_SVC_PARAM_RDBPI_SAFR = $3,
    PUDS_SVC_PARAM_RDBPI_SS = $4
);
```

C#

```
public enum uds_svc_param_rdbpi : Byte
{
    PUDS_SVC_PARAM_RDBPI_SASR = 0x01,
    PUDS_SVC_PARAM_RDBPI_SAMR = 0x02,
    PUDS_SVC_PARAM_RDBPI_SAFR = 0x03,
    PUDS_SVC_PARAM_RDBPI_SS = 0x04
}
```

C++ / CLR

```
enum struct uds_svc_param_rdbpi : Byte
{
    PUDS_SVC_PARAM_RDBPI_SASR = 0x01,
    PUDS_SVC_PARAM_RDBPI_SAMR = 0x02,
    PUDS_SVC_PARAM_RDBPI_SAFR = 0x03,
    PUDS_SVC_PARAM_RDBPI_SS = 0x04
};
```

Visual Basic

```
Public Enum uds_svc_param_rdbpi As Byte
    PUDS_SVC_PARAM_RDBPI_SASR = &H1
    PUDS_SVC_PARAM_RDBPI_SAMR = &H2
    PUDS_SVC_PARAM_RDBPI_SAFR = &H3
    PUDS_SVC_PARAM_RDBPI_SS = &H4
End Enum
```

Values

| Name | Value | Description |
|---------------------------|-------|----------------------|
| PUDS_SVC_PARAM_RDBPI_SASR | 1 | Send At Slow Rate. |
| PUDS_SVC_PARAM_RDBPI_SAMR | 2 | Send At Medium Rate. |
| PUDS_SVC_PARAM_RDBPI_SAFR | 3 | Send At Fast Rate. |
| PUDS_SVC_PARAM_RDBPI_SS | 4 | Stop Sending. |

See also: [UDS_SvcReadDataByIdentifier_2013](#) on page 681 (**class-method:** [SvcReadDataByIdentifier_2013](#) on page 337).

3.5.22 uds_svc_param_dddI

Represents the subfunction parameter for UDS service DynamicallyDefineDataIdentifier. According to the programming language, this type can be a group of defined values or an enumeration.

Syntax

C/C++

```
#define PUDS_SVC_PARAM_DDDI_DBID      0x01
#define PUDS_SVC_PARAM_DDDI_DBMA      0x02
#define PUDS_SVC_PARAM_DDDI_CDDDI     0x03
```

Pascal OO

```
uds_svc_param_dddI = (
    PUDS_SVC_PARAM_DDDI_DBID = $1,
    PUDS_SVC_PARAM_DDDI_DBMA = $2,
    PUDS_SVC_PARAM_DDDI_CDDDI = $3
);
```

C#

```
public enum uds_svc_param_dddI : Byte
{
    PUDS_SVC_PARAM_DDDI_DBID = 0x01,
    PUDS_SVC_PARAM_DDDI_DBMA = 0x02,
    PUDS_SVC_PARAM_DDDI_CDDDI = 0x03
}
```

C++ / CLR

```
enum struct uds_svc_param_dddI : Byte
{
    PUDS_SVC_PARAM_DDDI_DBID = 0x01,
    PUDS_SVC_PARAM_DDDI_DBMA = 0x02,
    PUDS_SVC_PARAM_DDDI_CDDDI = 0x03
};
```

Visual Basic

```
Public Enum uds_svc_param_dddI As Byte
    PUDS_SVC_PARAM_DDDI_DBID = &H1
    PUDS_SVC_PARAM_DDDI_DBMA = &H2
    PUDS_SVC_PARAM_DDDI_CDDDI = &H3
End Enum
```

Values

| Name | Value | Description |
|---------------------------|-------|--|
| PUDS_SVC_PARAM_DDDI_DBID | 1 | Define By Identifier. |
| PUDS_SVC_PARAM_DDDI_DBMA | 2 | Define By Memory Address. |
| PUDS_SVC_PARAM_DDDI_CDDDI | 3 | Clear Dynamically Defined Data Identifier. |

See also: [UDS_SvcDynamicallyDefineDataIdentifierDBID_2013](#) on page 688 (**class-method:**

[SvcDynamicallyDefineDataIdentifierDBID_2013](#) on page 356, [UDS_SvcDynamicallyDefineDataIdentifierDBMA_2013](#) on page 690, [SvcDynamicallyDefineDataIdentifierDBMA_2013](#) on page 362, [UDS_SvcDynamicallyDefineDataIdentifierCDDDI_2013](#) on page 693, [SvcDynamicallyDefineDataIdentifierCDDDI_2013](#) on page 368).

3.5.23 uds_svc_param_rdtci

Represents the subfunction parameter for UDS service ReadDTCInformation. According to the programming language, this type can be a group of defined values or an enumeration.

Syntax

C/C++

```
#define PUDS_SVC_PARAM_RDTCI_RNODTCBSM      0x01
#define PUDS_SVC_PARAM_RDTCI_RDTCBSM       0x02
#define PUDS_SVC_PARAM_RDTCI_RDTCSSI       0x03
#define PUDS_SVC_PARAM_RDTCI_RDTCSSBDTC    0x04
#define PUDS_SVC_PARAM_RDTCI_RDTCSSBRN     0x05
#define PUDS_SVC_PARAM_RDTCI_RDTCEDRBDN    0x06
#define PUDS_SVC_PARAM_RDTCI_RNODTCBSMR    0x07
#define PUDS_SVC_PARAM_RDTCI_RDTCBSMR      0x08
#define PUDS_SVC_PARAM_RDTCI_RSIODTC       0x09
#define PUDS_SVC_PARAM_RDTCI_RSUPDTC       0x0A
#define PUDS_SVC_PARAM_RDTCI_RFTFDTC       0x0B
#define PUDS_SVC_PARAM_RDTCI_RFCDTC        0x0C
#define PUDS_SVC_PARAM_RDTCI_RMRTFDTC      0x0D
#define PUDS_SVC_PARAM_RDTCI_RMRCDTC       0x0E
#define PUDS_SVC_PARAM_RDTCI_RMMDTCBSM     0x0F
#define PUDS_SVC_PARAM_RDTCI_RMMDEDRBDN    0x10
#define PUDS_SVC_PARAM_RDTCI_RNOMMDTCBSM   0x11
#define PUDS_SVC_PARAM_RDTCI_RNOOBDDTCBSM  0x12
#define PUDS_SVC_PARAM_RDTCI_ROBDDTCBSM    0x13
#define PUDS_SVC_PARAM_RDTCI_RDTCEDBR      0x16
#define PUDS_SVC_PARAM_RDTCI_RUDMDTCBSM    0x17
#define PUDS_SVC_PARAM_RDTCI_RUDMDTCSSBDTC 0x18
#define PUDS_SVC_PARAM_RDTCI_RUDMDTCEDRBDN 0x19
#define PUDS_SVC_PARAM_RDTCI_RDTCEDI       0x1A
#define PUDS_SVC_PARAM_RDTCI_RWWHOBDDTCBMR 0x42
#define PUDS_SVC_PARAM_RDTCI_RWWHOBDDTCWPS 0x55
#define PUDS_SVC_PARAM_RDTCI_RDTCBRGI      0x56
#define PUDS_SVC_PARAM_RDTCI_RDTCFDC       0x14
#define PUDS_SVC_PARAM_RDTCI_RDTCWPS       0x15
```

Pascal OO

```
uds_svc_param_rdtci = (
    PUDS_SVC_PARAM_RDTCI_RNODTCBSM = $1,
    PUDS_SVC_PARAM_RDTCI_RDTCBSM = $2,
    PUDS_SVC_PARAM_RDTCI_RDTCSSI = $3,
    PUDS_SVC_PARAM_RDTCI_RDTCSSBDTC = $4,
    PUDS_SVC_PARAM_RDTCI_RDTCSSBRN = $5,
    PUDS_SVC_PARAM_RDTCI_RDTCEDRBDN = $6,
    PUDS_SVC_PARAM_RDTCI_RNODTCBSMR = $7,
    PUDS_SVC_PARAM_RDTCI_RDTCBSMR = $8,
    PUDS_SVC_PARAM_RDTCI_RSIODTC = $9,
    PUDS_SVC_PARAM_RDTCI_RSUPDTC = $A,
    PUDS_SVC_PARAM_RDTCI_RFTFDTC = $B,
    PUDS_SVC_PARAM_RDTCI_RFCDTC = $C,
    PUDS_SVC_PARAM_RDTCI_RMRTFDTC = $D,
    PUDS_SVC_PARAM_RDTCI_RMRCDTC = $E,
    PUDS_SVC_PARAM_RDTCI_RMMDTCBSM = $F,
    PUDS_SVC_PARAM_RDTCI_RMMDEDRBDN = $10,
    PUDS_SVC_PARAM_RDTCI_RNOMMDTCBSM = $11,
    PUDS_SVC_PARAM_RDTCI_RNOOBDDTCBSM = $12,
    PUDS_SVC_PARAM_RDTCI_ROBDDTCBSM = $13,
    PUDS_SVC_PARAM_RDTCI_RDTCEDBR = $16,
    PUDS_SVC_PARAM_RDTCI_RUDMDTCBSM = $17,
```



```

PUDS_SVC_PARAM_RDTCI_RUDMDTCSSBDTC = $18,
PUDS_SVC_PARAM_RDTCI_RUDMDTCEDRBDN = $19,
PUDS_SVC_PARAM_RDTCI_RDTCEDI = $1A,
PUDS_SVC_PARAM_RDTCI_RWWHOBDDTCBMR = $42,
PUDS_SVC_PARAM_RDTCI_RWWHOBDDTCWPS = $55,
PUDS_SVC_PARAM_RDTCI_RDTCBRGI = $56,
PUDS_SVC_PARAM_RDTCI_RDTCFDC = $14,
PUDS_SVC_PARAM_RDTCI_RDTCWPS = $15
);

```

C#

```

public enum uds_svc_param_rdtci : Byte
{
    PUDS_SVC_PARAM_RDTCI_RNODTCBSM = 0x01,
    PUDS_SVC_PARAM_RDTCI_RDTCBSM = 0x02,
    PUDS_SVC_PARAM_RDTCI_RDTCSSI = 0x03,
    PUDS_SVC_PARAM_RDTCI_RDTCSSBDTC = 0x04,
    PUDS_SVC_PARAM_RDTCI_RDTCSSBRN = 0x05,
    PUDS_SVC_PARAM_RDTCI_RDTCEDRBDN = 0x06,
    PUDS_SVC_PARAM_RDTCI_RNODTCBSMR = 0x07,
    PUDS_SVC_PARAM_RDTCI_RDTCBSMR = 0x08,
    PUDS_SVC_PARAM_RDTCI_RSIODTC = 0x09,
    PUDS_SVC_PARAM_RDTCI_RSUPDTC = 0x0A,
    PUDS_SVC_PARAM_RDTCI_RFTFDTC = 0x0B,
    PUDS_SVC_PARAM_RDTCI_RFCDDTC = 0x0C,
    PUDS_SVC_PARAM_RDTCI_RMRTFDTC = 0x0D,
    PUDS_SVC_PARAM_RDTCI_RMRCDDTC = 0x0E,
    PUDS_SVC_PARAM_RDTCI_RMMDDTCBSM = 0x0F,
    PUDS_SVC_PARAM_RDTCI_RMMDEDRBDN = 0x10,
    PUDS_SVC_PARAM_RDTCI_RNOMMDTCBSM = 0x11,
    PUDS_SVC_PARAM_RDTCI_RNOOBDDTCBSM = 0x12,
    PUDS_SVC_PARAM_RDTCI_ROBDDTCBSM = 0x13,
    PUDS_SVC_PARAM_RDTCI_RDTCEDBR = 0x16,
    PUDS_SVC_PARAM_RDTCI_RUDMDTCBSM = 0x17,
    PUDS_SVC_PARAM_RDTCI_RUDMDTCSSBDTC = 0x18,
    PUDS_SVC_PARAM_RDTCI_RUDMDTCEDRBDN = 0x19,
    PUDS_SVC_PARAM_RDTCI_RDTCEDI = 0x1A,
    PUDS_SVC_PARAM_RDTCI_RWWHOBDDTCBMR = 0x42,
    PUDS_SVC_PARAM_RDTCI_RWWHOBDDTCWPS = 0x55,
    PUDS_SVC_PARAM_RDTCI_RDTCBRGI = 0x56,
    PUDS_SVC_PARAM_RDTCI_RDTCFDC = 0x14,
    PUDS_SVC_PARAM_RDTCI_RDTCWPS = 0x15
}

```

C++ / CLR

```

enum struct uds_svc_param_rdtci : Byte
{
    PUDS_SVC_PARAM_RDTCI_RNODTCBSM = 0x01,
    PUDS_SVC_PARAM_RDTCI_RDTCBSM = 0x02,
    PUDS_SVC_PARAM_RDTCI_RDTCSSI = 0x03,
    PUDS_SVC_PARAM_RDTCI_RDTCSSBDTC = 0x04,
    PUDS_SVC_PARAM_RDTCI_RDTCSSBRN = 0x05,
    PUDS_SVC_PARAM_RDTCI_RDTCEDRBDN = 0x06,
    PUDS_SVC_PARAM_RDTCI_RNODTCBSMR = 0x07,
    PUDS_SVC_PARAM_RDTCI_RDTCBSMR = 0x08,
    PUDS_SVC_PARAM_RDTCI_RSIODTC = 0x09,
    PUDS_SVC_PARAM_RDTCI_RSUPDTC = 0x0A,
    PUDS_SVC_PARAM_RDTCI_RFTFDTC = 0x0B,
    PUDS_SVC_PARAM_RDTCI_RFCDDTC = 0x0C,
    PUDS_SVC_PARAM_RDTCI_RMRTFDTC = 0x0D,

```

```

PUDS_SVC_PARAM_RDTCI_RMRCBTC = 0x0E,
PUDS_SVC_PARAM_RDTCI_RMMDTCSM = 0x0F,
PUDS_SVC_PARAM_RDTCI_RMMDEDRBDN = 0x10,
PUDS_SVC_PARAM_RDTCI_RNOMMDTCBSM = 0x11,
PUDS_SVC_PARAM_RDTCI_RNOOBDDTCBSM = 0x12,
PUDS_SVC_PARAM_RDTCI_ROBDDTCBSM = 0x13,
PUDS_SVC_PARAM_RDTCI_RDTCEDBR = 0x16,
PUDS_SVC_PARAM_RDTCI_RUDMDTCBSM = 0x17,
PUDS_SVC_PARAM_RDTCI_RUDMDTCSSBDTC = 0x18,
PUDS_SVC_PARAM_RDTCI_RUDMDTCEDRBDN = 0x19,
PUDS_SVC_PARAM_RDTCI_RDTCEDI = 0x1A,
PUDS_SVC_PARAM_RDTCI_RWWHOBDDTCBMR = 0x42,
PUDS_SVC_PARAM_RDTCI_RWWHOBDDTCWPS = 0x55,
PUDS_SVC_PARAM_RDTCI_RDTCBRGI = 0x56,
PUDS_SVC_PARAM_RDTCI_RDTCFDC = 0x14,
PUDS_SVC_PARAM_RDTCI_RDTCWPS = 0x15

```

```
};
```

Visual Basic

```
Public Enum uds_svc_param_rdtci As Byte
```

```

    PUDS_SVC_PARAM_RDTCI_RNODTCBSM = &H1
    PUDS_SVC_PARAM_RDTCI_RDTCBSM = &H2
    PUDS_SVC_PARAM_RDTCI_RDTCSSI = &H3
    PUDS_SVC_PARAM_RDTCI_RDTCSSBDTC = &H4
    PUDS_SVC_PARAM_RDTCI_RDTCSSBRN = &H5
    PUDS_SVC_PARAM_RDTCI_RDTCEDRBDN = &H6
    PUDS_SVC_PARAM_RDTCI_RNODTCBSMR = &H7
    PUDS_SVC_PARAM_RDTCI_RDTCBSMR = &H8
    PUDS_SVC_PARAM_RDTCI_RSIODTC = &H9
    PUDS_SVC_PARAM_RDTCI_RSUPDTC = &HA
    PUDS_SVC_PARAM_RDTCI_RFTFDTC = &HB
    PUDS_SVC_PARAM_RDTCI_RFCDDTC = &HC
    PUDS_SVC_PARAM_RDTCI_RMRTFDTC = &HD
    PUDS_SVC_PARAM_RDTCI_RMRCBTC = &HE
    PUDS_SVC_PARAM_RDTCI_RMMDTCSM = &HF
    PUDS_SVC_PARAM_RDTCI_RMMDEDRBDN = &H10
    PUDS_SVC_PARAM_RDTCI_RNOMMDTCBSM = &H11
    PUDS_SVC_PARAM_RDTCI_RNOOBDDTCBSM = &H12
    PUDS_SVC_PARAM_RDTCI_ROBDDTCBSM = &H13
    PUDS_SVC_PARAM_RDTCI_RDTCEDBR = &H16
    PUDS_SVC_PARAM_RDTCI_RUDMDTCBSM = &H17
    PUDS_SVC_PARAM_RDTCI_RUDMDTCSSBDTC = &H18
    PUDS_SVC_PARAM_RDTCI_RUDMDTCEDRBDN = &H19
    PUDS_SVC_PARAM_RDTCI_RDTCEDI = &H1A
    PUDS_SVC_PARAM_RDTCI_RWWHOBDDTCBMR = &H42
    PUDS_SVC_PARAM_RDTCI_RWWHOBDDTCWPS = &H55
    PUDS_SVC_PARAM_RDTCI_RDTCBRGI = &H56
    PUDS_SVC_PARAM_RDTCI_RDTCFDC = &H14
    PUDS_SVC_PARAM_RDTCI_RDTCWPS = &H15

```

```
End Enum
```

Values

| Name | Value | Description |
|---------------------------------|----------|--|
| PUDS_SVC_PARAM_RDTCI_RNODTCBSM | 0x01 (1) | Report Number Of DTC By Status Mask. |
| PUDS_SVC_PARAM_RDTCI_RDTCBSM | 0x02 (2) | Report DTC By Status Mask. |
| PUDS_SVC_PARAM_RDTCI_RDTCSSI | 0x03 (3) | Report DTC Snapshot Identification. |
| PUDS_SVC_PARAM_RDTCI_RDTCSSBDTC | 0x04 (4) | Report DTC Snapshot Record By DTC Number. |
| PUDS_SVC_PARAM_RDTCI_RDTCSSBRN | 0x05 (5) | Report DTC Snapshot Record By Record Number. |
| PUDS_SVC_PARAM_RDTCI_RDTCEDRBDN | 0x06 (6) | Report DTC Extended Data Record By DTC Number. |

| Name | Value | Description |
|------------------------------------|-----------|---|
| PUDS_SVC_PARAM_RDTCI_RNODTCBSMR | 0x07 (7) | Report Number Of DTC By Severity Mask Record. |
| PUDS_SVC_PARAM_RDTCI_RDTCSMR | 0x08 (8) | Report DTC By Severity Mask Record. |
| PUDS_SVC_PARAM_RDTCI_RSIODTC | 0x09 (9) | Report Severity Information Of DTC. |
| PUDS_SVC_PARAM_RDTCI_RSUPDTC | 0x0A (10) | Report Supported DTC. |
| PUDS_SVC_PARAM_RDTCI_RFTFDTC | 0x0B (11) | Report First Test Failed DTC. |
| PUDS_SVC_PARAM_RDTCI_RFCDDTC | 0x0C (12) | Report First Confirmed DTC. |
| PUDS_SVC_PARAM_RDTCI_RMRTFDTC | 0x0D (13) | Report Most Recent Test Failed DTC. |
| PUDS_SVC_PARAM_RDTCI_RMRCDDTC | 0x0E (14) | Report Most Recent Confirmed DTC. |
| PUDS_SVC_PARAM_RDTCI_RMMDTCBSM | 0x0F (15) | Report Mirror Memory DTC By Status Mask. |
| PUDS_SVC_PARAM_RDTCI_RMMDEDRBDN | 0x10 (16) | Report Mirror Memory DTC Extended Data Record By DTC Number. |
| PUDS_SVC_PARAM_RDTCI_RNOMMDTCBSM, | 0x11 (16) | Report Number Of Mirror MemoryDTC By Status Mask. |
| PUDS_SVC_PARAM_RDTCI_RNOOBDDTCBSM | 0x12 (17) | Report Number Of Emissions Related OBD DTC By Status Mask. |
| PUDS_SVC_PARAM_RDTCI_ROBDDTCBSM | 0x13 (18) | Report Emissions Related OBD DTC By Status Mask. |
| PUDS_SVC_PARAM_RDTCI_RDTCEDBR | 0x16 (22) | Report DTC Ext Data Record By Record Number. |
| PUDS_SVC_PARAM_RDTCI_RUDMDTCBSM | 0x17 (23) | Report User Def Memory DTC By Status Mask. |
| PUDS_SVC_PARAM_RDTCI_RUDMDTCSSBDTC | 0x18 (24) | Report User Def Memory DTC Snapshot Record By DTC Number. |
| PUDS_SVC_PARAM_RDTCI_RUDMDTCEDRBDN | 0x19 (25) | Report User Def Memory DTC Ext Data Record By DTC Number. |
| PUDS_SVC_PARAM_RDTCI_RDTCEDI | 0x1A (26) | Report DTC Extended Data Record Identification (ISO_14229-1 2020) |
| PUDS_SVC_PARAM_RDTCI_RWWHOBDDTCBMR | 0x42 (66) | Report WWHOBDDTC By Mask Record. |
| PUDS_SVC_PARAM_RDTCI_RWWHOBDDTCWPS | 0x55 (85) | Report WWHOBDDTC With Permanent Status. |
| PUDS_SVC_PARAM_RDTCI_RDTCBRGI | 0x56 (86) | Report DTC Information By DTC Readiness Group Identifier (ISO_14229-1 2020) |
| PUDS_SVC_PARAM_RDTCI_RDTCFDC | 0x14 (20) | Not in ISO-15765-3: report DTC Fault Detection Counter. |
| PUDS_SVC_PARAM_RDTCI_RDTCWPS | 0x15 (21) | Not in ISO-15765-3: report DTC With Permanent Status. |

See also: [UDS_SvcReadDTCInformation_2013](#) on page 704 (**class-method:** [SvcReadDTCInformation_2013](#) on page 396).

3.5.24 uds_svc_param_rdtci_dtcsvm

Represents the DTC severity mask's flags (DTCSVM) used with the UDS service SvcReadDTCInformation. According to the programming language, this type can be a group of defined values or an enumeration.

Syntax

C/C++

```
#define PUDS_SVC_PARAM_RDTCI_DTCSVM_NSA 0x00
#define PUDS_SVC_PARAM_RDTCI_DTCSVM_MO 0x20
#define PUDS_SVC_PARAM_RDTCI_DTCSVM_CHKANH 0x40
#define PUDS_SVC_PARAM_RDTCI_DTCSVM_CHK1 0x80
```

Pascal OO

```
uds_svc_param_rdtci_dtcsvm = (
    PUDS_SVC_PARAM_RDTCI_DTCSVM_NSA = $0,
    PUDS_SVC_PARAM_RDTCI_DTCSVM_MO = $20,
    PUDS_SVC_PARAM_RDTCI_DTCSVM_CHKANH = $40,
    PUDS_SVC_PARAM_RDTCI_DTCSVM_CHK1 = $80
);
```

C#

```
[Flags]
public enum uds_svc_param_rdtci_dtcsvm : Byte
{
    PUDS_SVC_PARAM_RDTCI_DTCSVM_NSA = 0x00,
```

```

    PUDS_SVC_PARAM_RDTCI_DTCSVM_MO = 0x20,
    PUDS_SVC_PARAM_RDTCI_DTCSVM_CHKANH = 0x40,
    PUDS_SVC_PARAM_RDTCI_DTCSVM_CHK_I = 0x80
}

```

C++ / CLR

```

enum struct uds_svc_param_rdtci_dtcsvm : Byte
{
    PUDS_SVC_PARAM_RDTCI_DTCSVM_NSA = 0x00,
    PUDS_SVC_PARAM_RDTCI_DTCSVM_MO = 0x20,
    PUDS_SVC_PARAM_RDTCI_DTCSVM_CHKANH = 0x40,
    PUDS_SVC_PARAM_RDTCI_DTCSVM_CHK_I = 0x80
};

```

Visual Basic

```

<Flags(>
Public Enum uds_svc_param_rdtci_dtcsvm As Byte
    PUDS_SVC_PARAM_RDTCI_DTCSVM_NSA = &H0
    PUDS_SVC_PARAM_RDTCI_DTCSVM_MO = &H20
    PUDS_SVC_PARAM_RDTCI_DTCSVM_CHKANH = &H40
    PUDS_SVC_PARAM_RDTCI_DTCSVM_CHK_I = &H80
End Enum

```

Values

| Name | Value | Description |
|------------------------------------|------------|--|
| PUDS_SVC_PARAM_RDTCI_DTCSVM_NSA | 0x00 (0) | DTC severity bit definitions: No Severity Available. |
| PUDS_SVC_PARAM_RDTCI_DTCSVM_MO | 0x20 (32) | DTC severity bit definitions: Maintenance Only. |
| PUDS_SVC_PARAM_RDTCI_DTCSVM_CHKANH | 0x40 (64) | DTC severity bit definitions: CHecK At Next Halt. |
| PUDS_SVC_PARAM_RDTCI_DTCSVM_CHK_I | 0x80 (128) | DTC severity bit definitions: Check Immediately. |

See also: [UDS_SvcReadDTCInformation_2013](#) on page 704 (**class-method:** [SvcReadDTCInformation_2013](#) on page 396).

3.5.25 uds_svc_param_iocbi

Represents the InputOutputControl parameter for UDS service InputOutputControlByIdentifier. According to the programming language, this type can be a group of defined values or an enumeration.

Syntax

C/C++

```

#define PUDS_SVC_PARAM_IOCBI_RTECU 0x00
#define PUDS_SVC_PARAM_IOCBI_RTD 0x01
#define PUDS_SVC_PARAM_IOCBI_FCS 0x02
#define PUDS_SVC_PARAM_IOCBI_STA 0x03

```

Pascal OO

```

uds_svc_param_iocbi = (
    PUDS_SVC_PARAM_IOCBI_RTECU = $0,
    PUDS_SVC_PARAM_IOCBI_RTD = $1,
    PUDS_SVC_PARAM_IOCBI_FCS = $2,
    PUDS_SVC_PARAM_IOCBI_STA = $3
);

```

C#

```
public enum uds_svc_param_iocbi : Byte
{
    PUDS_SVC_PARAM_IOCBI_RCTECU = 0x00,
    PUDS_SVC_PARAM_IOCBI_RTD = 0x01,
    PUDS_SVC_PARAM_IOCBI_FCS = 0x02,
    PUDS_SVC_PARAM_IOCBI_STA = 0x03
}
```

C++ / CLR

```
enum struct uds_svc_param_iocbi : Byte
{
    PUDS_SVC_PARAM_IOCBI_RCTECU = 0x00,
    PUDS_SVC_PARAM_IOCBI_RTD = 0x01,
    PUDS_SVC_PARAM_IOCBI_FCS = 0x02,
    PUDS_SVC_PARAM_IOCBI_STA = 0x03
};
```

Visual Basic

```
Public Enum uds_svc_param_iocbi As Byte
    PUDS_SVC_PARAM_IOCBI_RCTECU = &H0
    PUDS_SVC_PARAM_IOCBI_RTD = &H1
    PUDS_SVC_PARAM_IOCBI_FCS = &H2
    PUDS_SVC_PARAM_IOCBI_STA = &H3
End Enum
```

Values

| Name | Value | Description |
|-----------------------------|-------|------------------------|
| PUDS_SVC_PARAM_IOCBI_RCTECU | 0 | Return Control To ECU. |
| PUDS_SVC_PARAM_IOCBI_RTD | 1 | Reset To Default. |
| PUDS_SVC_PARAM_IOCBI_FCS | 2 | Freeze Current State. |
| PUDS_SVC_PARAM_IOCBI_STA | 3 | Short Term Adjustment. |

See also: [UDS_SvcInputOutputControlByIdentifier_2013](#) on page 731
(**class-method:** [SvcInputOutputControlByIdentifier_2013](#) on page 462).

3.5.26 uds_svc_param_rc

Represents the subfunction parameter for UDS service RoutineControl. According to the programming language, this type can be a group of defined values or an enumeration.

Syntax

C/C++

```
#define PUDS_SVC_PARAM_RC_STR          0x01
#define PUDS_SVC_PARAM_RC_STPR        0x02
#define PUDS_SVC_PARAM_RC_RRR        0x03
```

Pascal OO

```
uds_svc_param_rc = (
    PUDS_SVC_PARAM_RC_STR = $1,
    PUDS_SVC_PARAM_RC_STPR = $2,
    PUDS_SVC_PARAM_RC_RRR = $3
);
```

C#

```
public enum uds_svc_param_rc : Byte
{
    PUDS_SVC_PARAM_RC_STR = 0x01,
    PUDS_SVC_PARAM_RC_STPR = 0x02,
    PUDS_SVC_PARAM_RC_RRR = 0x03
}
```

C++ / CLR

```
enum struct uds_svc_param_rc : Byte
{
    PUDS_SVC_PARAM_RC_STR = 0x01,
    PUDS_SVC_PARAM_RC_STPR = 0x02,
    PUDS_SVC_PARAM_RC_RRR = 0x03
};
```

Visual Basic

```
Public Enum uds_svc_param_rc As Byte
    PUDS_SVC_PARAM_RC_STR = &H1
    PUDS_SVC_PARAM_RC_STPR = &H2
    PUDS_SVC_PARAM_RC_RRR = &H3
End Enum
```

Values

| Name | Value | Description |
|------------------------|-------|--------------------------|
| PUDS_SVC_PARAM_RC_STR | 1 | Start Routine. |
| PUDS_SVC_PARAM_RC_STPR | 2 | Stop Routine. |
| PUDS_SVC_PARAM_RC_RRR | 3 | Request Routine Results. |

See also: [UDS_SvcRoutineControl_2013](#) on page 733 (**class-method:** [SvcRoutineControl_2013](#) on page 473).

3.5.27 uds_svc_param_rc_rid

Represents the routine identifier used with the UDS service RoutineControl. According to the programming language, this type can be a group of defined values or an enumeration.

Syntax

C/C++

```
#define PUDS_SVC_PARAM_RC_RID_DLRI_ 0xE200
#define PUDS_SVC_PARAM_RC_RID_EM_ 0xFF00
#define PUDS_SVC_PARAM_RC_RID_CPD_ 0xFF01
#define PUDS_SVC_PARAM_RC_RID_EEMDTC_ 0xFF02
```

Pascal OO

```
uds_svc_param_rc_rid = (
    PUDS_SVC_PARAM_RC_RID_DLRI_ = $E200,
    PUDS_SVC_PARAM_RC_RID_EM_ = $FF00,
    PUDS_SVC_PARAM_RC_RID_CPD_ = $FF01,
    PUDS_SVC_PARAM_RC_RID_EEMDTC_ = $FF02
);
```

C#

```
public enum uds_svc_param_rc_rid : UInt16
{
```

```

PUDS_SVC_PARAM_RC_RID_DLRI_ = 0xE200,
PUDS_SVC_PARAM_RC_RID_EM_   = 0xFF00,
PUDS_SVC_PARAM_RC_RID_CPD_  = 0xFF01,
PUDS_SVC_PARAM_RC_RID_EMDTC_ = 0xFF02
}

```

C++ / CLR

```

enum struct uds_svc_param_rc_rid : UInt16
{
    PUDS_SVC_PARAM_RC_RID_DLRI_ = 0xE200,
    PUDS_SVC_PARAM_RC_RID_EM_   = 0xFF00,
    PUDS_SVC_PARAM_RC_RID_CPD_  = 0xFF01,
    PUDS_SVC_PARAM_RC_RID_EMDTC_ = 0xFF02
};

```

Visual Basic

```

Public Enum uds_svc_param_rc_rid As UInt16
    PUDS_SVC_PARAM_RC_RID_DLRI_ = &HE200
    PUDS_SVC_PARAM_RC_RID_EM_   = &HFF00
    PUDS_SVC_PARAM_RC_RID_CPD_  = &HFF01
    PUDS_SVC_PARAM_RC_RID_EMDTC_ = &HFF02
End Enum

```

Values

| Name | Value | Description |
|------------------------------|----------------|---------------------------------|
| PUDS_SVC_PARAM_RC_RID_DLRI_ | 0xE200 (57856) | Deploy Loop Routine ID. |
| PUDS_SVC_PARAM_RC_RID_EM_ | 0xFF00 (65280) | Erase Memory. |
| PUDS_SVC_PARAM_RC_RID_CPD_ | 0xFF01 (65281) | Check Programming Dependencies. |
| PUDS_SVC_PARAM_RC_RID_EMDTC_ | 0xFF02 (65282) | Erase Mirror Memory DTCs. |

See also: [UDS_SvcRoutineControl_2013](#) on page 733 (**class-method:** [SvcRoutineControl_2013](#) on page 473).

3.5.28 uds_svc_param_atp

Represents the subfunction parameter for the UDS service AccessTimingParameter. It defines the access type. According to the programming language, this type can be a group of defined values or an enumeration.

Syntax

C/C++

```

#define PUDS_SVC_PARAM_ATP_TRETPS      0x01
#define PUDS_SVC_PARAM_ATP_STPTDV      0x02
#define PUDS_SVC_PARAM_ATP_RCATP       0x03
#define PUDS_SVC_PARAM_ATP_STPTGV      0x04

```

Pascal OO

```

uds_svc_param_atp = (
    PUDS_SVC_PARAM_ATP_TRETPS = $1,
    PUDS_SVC_PARAM_ATP_STPTDV = $2,
    PUDS_SVC_PARAM_ATP_RCATP  = $3,
    PUDS_SVC_PARAM_ATP_STPTGV = $4
);

```

C#

```
public enum uds_svc_param_atp : Byte
{
    PUDS_SVC_PARAM_ATP_TRETPS = 0x01,
    PUDS_SVC_PARAM_ATP_STPTDV = 0x02,
    PUDS_SVC_PARAM_ATP_RCATP = 0x03,
    PUDS_SVC_PARAM_ATP_STPTGV = 0x04
}
```

C++ / CLR

```
enum struct uds_svc_param_atp : Byte
{
    PUDS_SVC_PARAM_ATP_TRETPS = 0x01,
    PUDS_SVC_PARAM_ATP_STPTDV = 0x02,
    PUDS_SVC_PARAM_ATP_RCATP = 0x03,
    PUDS_SVC_PARAM_ATP_STPTGV = 0x04
};
```

Visual Basic

```
Public Enum uds_svc_param_atp As Byte
    PUDS_SVC_PARAM_ATP_TRETPS = &H1
    PUDS_SVC_PARAM_ATP_STPTDV = &H2
    PUDS_SVC_PARAM_ATP_RCATP = &H3
    PUDS_SVC_PARAM_ATP_STPTGV = &H4
End Enum
```

Values

| Name | Value | Description |
|---------------------------|-------|--|
| PUDS_SVC_PARAM_ATP_TRETPS | 0x01 | Read Extended Timing Parameter Set. |
| PUDS_SVC_PARAM_ATP_STPTDV | 0x02 | Set Timing Parameters to Default Values. |
| PUDS_SVC_PARAM_ATP_RCATP | 0x03 | Read Currently Active Timing Parameters. |
| PUDS_SVC_PARAM_ATP_STPTGV | 0x04 | Set Timing Parameters to Given Values. |

See also: [UDS_SvcAccessTimingParameter_2013](#) on page 744 (**class-method:** [SvcAccessTimingParameter_2013](#) on page 512).

3.5.29 uds_svc_param_rft_moop

Represents the mode of operation parameter for the UDS service RequestFileTransfer. According to the programming language, this type can be a group of defined values or an enumeration.

Syntax

C/C++

```
#define PUDS_SVC_PARAM_RFT_MOOP_ADDFILE 0x1
#define PUDS_SVC_PARAM_RFT_MOOP_DELFIL 0x2
#define PUDS_SVC_PARAM_RFT_MOOP_REPLFILE 0x3
#define PUDS_SVC_PARAM_RFT_MOOP_RDFIL 0x4
#define PUDS_SVC_PARAM_RFT_MOOP_RDDIR 0x5
#define PUDS_SVC_PARAM_RFT_MOOP_RSFILE 0x6
```

Pascal OO

```
uds_svc_param_rft_moop = (
    PUDS_SVC_PARAM_RFT_MOOP_ADDFILE = $1,
    PUDS_SVC_PARAM_RFT_MOOP_DELFIL = $2,
    PUDS_SVC_PARAM_RFT_MOOP_REPLFILE = $3,
```



```

PUDS_SVC_PARAM_RFT_MOOP_RDFILE = $4,
PUDS_SVC_PARAM_RFT_MOOP_RDDIR = $5,
PUDS_SVC_PARAM_RFT_MOOP_RSFILE = $6
);

```

C#

```

public enum uds_svc_param_rft_moop : Byte
{
    PUDS_SVC_PARAM_RFT_MOOP_ADDFILE = 0x1,
    PUDS_SVC_PARAM_RFT_MOOP_DELFILE = 0x2,
    PUDS_SVC_PARAM_RFT_MOOP_REPLFILE = 0x3,
    PUDS_SVC_PARAM_RFT_MOOP_RDFILE = 0x4,
    PUDS_SVC_PARAM_RFT_MOOP_RDDIR = 0x5,
    PUDS_SVC_PARAM_RFT_MOOP_RSFILE = 0x6
}

```

C++ / CLR

```

enum struct uds_svc_param_rft_moop : Byte
{
    PUDS_SVC_PARAM_RFT_MOOP_ADDFILE = 0x1,
    PUDS_SVC_PARAM_RFT_MOOP_DELFILE = 0x2,
    PUDS_SVC_PARAM_RFT_MOOP_REPLFILE = 0x3,
    PUDS_SVC_PARAM_RFT_MOOP_RDFILE = 0x4,
    PUDS_SVC_PARAM_RFT_MOOP_RDDIR = 0x5,
    PUDS_SVC_PARAM_RFT_MOOP_RSFILE = 0x6
};

```

Visual Basic

```

Public Enum uds_svc_param_rft_moop As Byte
    PUDS_SVC_PARAM_RFT_MOOP_ADDFILE = &H1
    PUDS_SVC_PARAM_RFT_MOOP_DELFILE = &H2
    PUDS_SVC_PARAM_RFT_MOOP_REPLFILE = &H3
    PUDS_SVC_PARAM_RFT_MOOP_RDFILE = &H4
    PUDS_SVC_PARAM_RFT_MOOP_RDDIR = &H5
    PUDS_SVC_PARAM_RFT_MOOP_RSFILE = &H6
End Enum

```

Values

| Name | Value | Description |
|----------------------------------|-------|---------------------------------|
| PUDS_SVC_PARAM_RFT_MOOP_ADDFILE | 0x1 | Add file. |
| PUDS_SVC_PARAM_RFT_MOOP_DELFILE | 0x2 | Delete file. |
| PUDS_SVC_PARAM_RFT_MOOP_REPLFILE | 0x3 | Replace file. |
| PUDS_SVC_PARAM_RFT_MOOP_RDFILE | 0x4 | Read file. |
| PUDS_SVC_PARAM_RFT_MOOP_RDDIR | 0x5 | Read directory. |
| PUDS_SVC_PARAM_RFT_MOOP_RSFILE | 0x6 | Resume file (ISO-14229-1:2020). |

See also: [UDS_SvcRequestFileTransfer_2013](#) on page 745 (**class-method:** [SvcRequestFileTransfer_2013](#) on page 517).

3.5.30 uds_svc_authentication_subfunction

Represents the subfunction parameter for UDS service Authentication (see ISO 14229-1:2020 §10.6.5.2 Table 74 Request message SubFunction parameter definition p.76).

Syntax

C/C++

```
typedef enum _uds_svc_authentication_subfunction
{
    PUDS_SVC_PARAM_AT_DA = 0x00,
    PUDS_SVC_PARAM_AT_VCU = 0x01,
    PUDS_SVC_PARAM_AT_VCB = 0x02,
    PUDS_SVC_PARAM_AT_POWN = 0x03,
    PUDS_SVC_PARAM_AT_TC = 0x04,
    PUDS_SVC_PARAM_AT_RCFA = 0x05,
    PUDS_SVC_PARAM_AT_VPOWNU = 0x06,
    PUDS_SVC_PARAM_AT_VPOWNB = 0x07,
    PUDS_SVC_PARAM_AT_AC = 0x08
} uds_svc_authentication_subfunction;
```

Pascal OO

```
uds_svc_authentication_subfunction = (
    PUDS_SVC_PARAM_AT_DA = $00,
    PUDS_SVC_PARAM_AT_VCU = $01,
    PUDS_SVC_PARAM_AT_VCB = $02,
    PUDS_SVC_PARAM_AT_POWN = $03,
    PUDS_SVC_PARAM_AT_TC = $04,
    PUDS_SVC_PARAM_AT_RCFA = $05,
    PUDS_SVC_PARAM_AT_VPOWNU = $06,
    PUDS_SVC_PARAM_AT_VPOWNB = $07,
    PUDS_SVC_PARAM_AT_AC = $08);
```

C#

```
public enum uds_svc_authentication_subfunction : Byte
{
    PUDS_SVC_PARAM_AT_DA = 0x00,
    PUDS_SVC_PARAM_AT_VCU = 0x01,
    PUDS_SVC_PARAM_AT_VCB = 0x02,
    PUDS_SVC_PARAM_AT_POWN = 0x03,
    PUDS_SVC_PARAM_AT_TC = 0x04,
    PUDS_SVC_PARAM_AT_RCFA = 0x05,
    PUDS_SVC_PARAM_AT_VPOWNU = 0x06,
    PUDS_SVC_PARAM_AT_VPOWNB = 0x07,
    PUDS_SVC_PARAM_AT_AC = 0x08
}
```

C++ / CLR

```
enum struct uds_svc_authentication_subfunction : Byte
{
    PUDS_SVC_PARAM_AT_DA = 0x00,
    PUDS_SVC_PARAM_AT_VCU = 0x01,
    PUDS_SVC_PARAM_AT_VCB = 0x02,
    PUDS_SVC_PARAM_AT_POWN = 0x03,
    PUDS_SVC_PARAM_AT_TC = 0x04,
    PUDS_SVC_PARAM_AT_RCFA = 0x05,
    PUDS_SVC_PARAM_AT_VPOWNU = 0x06,
    PUDS_SVC_PARAM_AT_VPOWNB = 0x07,
    PUDS_SVC_PARAM_AT_AC = 0x08
}
```

};

Visual Basic

```
Public Enum uds_svc_authentication_subfunction As Byte
    PUDS_SVC_PARAM_AT_DA = &H0
    PUDS_SVC_PARAM_AT_VCU = &H1
    PUDS_SVC_PARAM_AT_VCB = &H2
    PUDS_SVC_PARAM_AT_POWN = &H3
    PUDS_SVC_PARAM_AT_TC = &H4
    PUDS_SVC_PARAM_AT_RCFA = &H5
    PUDS_SVC_PARAM_AT_VPOWNU = &H6
    PUDS_SVC_PARAM_AT_VPOWNB = &H7
    PUDS_SVC_PARAM_AT_AC = &H8
End Enum
```

Values

| Name | Value | Description |
|--------------------------|-------|---|
| PUDS_SVC_PARAM_AT_DA | 0x00 | DeAuthenticate subfunction. |
| PUDS_SVC_PARAM_AT_VCU | 0x01 | VerifyCertificateUnidirectional subfunction. |
| PUDS_SVC_PARAM_AT_VCB | 0x02 | VerifyCertificateBidirectional subfunction. |
| PUDS_SVC_PARAM_AT_POWN | 0x03 | ProofOfOwnership subfunction. |
| PUDS_SVC_PARAM_AT_TC | 0x04 | TransmitCertificate subfunction. |
| PUDS_SVC_PARAM_AT_RCFA | 0x05 | RequestChallengeForAuthentication subfunction. |
| PUDS_SVC_PARAM_AT_VPOWNU | 0x06 | VerifyProofOfOwnershipUnidirectional subfunction. |
| PUDS_SVC_PARAM_AT_VPOWNB | 0x07 | VerifyProofOfOwnershipBidirectional subfunction. |
| PUDS_SVC_PARAM_AT_AC | 0x08 | AuthenticationConfiguration subfunction. |

3.5.31 uds_svc_authentication_return_parameter

Represents the return parameter for UDS service Authentication (see ISO 14229-1:2020 §B.5 AuthenticationReturnParameter definitions p.403).

Syntax

C/C++

```
typedef enum _uds_svc_authentication_return_parameter
{
    PUDS_SVC_PARAM_AT_RV_RA = 0x00,
    PUDS_SVC_PARAM_AT_RV_GR = 0x01,
    PUDS_SVC_PARAM_AT_RV_ACAPCE = 0x02,
    PUDS_SVC_PARAM_AT_RV_ACACRAC = 0x03,
    PUDS_SVC_PARAM_AT_RV_ACACRSC = 0x04,
    PUDS_SVC_PARAM_AT_RV_DAS = 0x10,
    PUDS_SVC_PARAM_AT_RV_CVOVN = 0x11,
    PUDS_SVC_PARAM_AT_RV_OVAC = 0x12,
    PUDS_SVC_PARAM_AT_RV_CV = 0x13
} uds_svc_authentication_return_parameter;
```

Pascal OO

```
uds_svc_authentication_return_parameter = (
    PUDS_SVC_PARAM_AT_RV_RA = $00,
    PUDS_SVC_PARAM_AT_RV_GR = $01,
    PUDS_SVC_PARAM_AT_RV_ACAPCE = $02,
    PUDS_SVC_PARAM_AT_RV_ACACRAC = $03,
    PUDS_SVC_PARAM_AT_RV_ACACRSC = $04,
    PUDS_SVC_PARAM_AT_RV_DAS = $10,
```

```
PUDS_SVC_PARAM_AT_RV_CVOVN = $11,
PUDS_SVC_PARAM_AT_RV_OVAC = $12,
PUDS_SVC_PARAM_AT_RV_CV = $13);
```

C#

```
public enum uds_svc_authentication_return_parameter : Byte
{
    PUDS_SVC_PARAM_AT_RV_RA = 0x00,
    PUDS_SVC_PARAM_AT_RV_GR = 0x01,
    PUDS_SVC_PARAM_AT_RV_ACAPCE = 0x02,
    PUDS_SVC_PARAM_AT_RV_ACACRAC = 0x03,
    PUDS_SVC_PARAM_AT_RV_ACACRSC = 0x04,
    PUDS_SVC_PARAM_AT_RV_DAS = 0x10,
    PUDS_SVC_PARAM_AT_RV_CVOVN = 0x11,
    PUDS_SVC_PARAM_AT_RV_OVAC = 0x12,
    PUDS_SVC_PARAM_AT_RV_CV = 0x13
}
```

C++ / CLR

```
enum struct uds_svc_authentication_return_parameter : Byte
{
    PUDS_SVC_PARAM_AT_RV_RA = 0x00,
    PUDS_SVC_PARAM_AT_RV_GR = 0x01,
    PUDS_SVC_PARAM_AT_RV_ACAPCE = 0x02,
    PUDS_SVC_PARAM_AT_RV_ACACRAC = 0x03,
    PUDS_SVC_PARAM_AT_RV_ACACRSC = 0x04,
    PUDS_SVC_PARAM_AT_RV_DAS = 0x10,
    PUDS_SVC_PARAM_AT_RV_CVOVN = 0x11,
    PUDS_SVC_PARAM_AT_RV_OVAC = 0x12,
    PUDS_SVC_PARAM_AT_RV_CV = 0x13
};
```

Visual Basic

```
Public Enum uds_svc_authentication_return_parameter As Byte
    PUDS_SVC_PARAM_AT_RV_RA = &H00
    PUDS_SVC_PARAM_AT_RV_GR = &H01
    PUDS_SVC_PARAM_AT_RV_ACAPCE = &H02
    PUDS_SVC_PARAM_AT_RV_ACACRAC = &H03
    PUDS_SVC_PARAM_AT_RV_ACACRSC = &H04
    PUDS_SVC_PARAM_AT_RV_DAS = &H10
    PUDS_SVC_PARAM_AT_RV_CVOVN = &H11
    PUDS_SVC_PARAM_AT_RV_OVAC = &H12
    PUDS_SVC_PARAM_AT_RV_CV = &H13
End Enum
```



Values

| Name | Value | Description |
|------------------------------|-------|--|
| PUDS_SVC_PARAM_AT_RV_RA | 0x00 | Request Accepted. |
| PUDS_SVC_PARAM_AT_RV_GR | 0x01 | General Reject. |
| PUDS_SVC_PARAM_AT_RV_ACAPCE | 0x02 | Authentication Configuration APCE. |
| PUDS_SVC_PARAM_AT_RV_ACACRAC | 0x03 | Authentication Configuration ACR with Asymmetric Cryptography. |
| PUDS_SVC_PARAM_AT_RV_ACACRSC | 0x04 | Authentication Configuration ACR with Symmetric Cryptography. |
| PUDS_SVC_PARAM_AT_RV_DAS | 0x10 | DeAuthentication Successful. |
| PUDS_SVC_PARAM_AT_RV_CVOVN | 0x11 | Certificate Verified, Ownership Verification Necessary. |
| PUDS_SVC_PARAM_AT_RV_OVAC | 0x12 | Ownership Verified, Authentication Complete. |
| PUDS_SVC_PARAM_AT_RV_CV | 0x13 | Certificate Verified. |


















3.6 PCAN-ISO-TP 3.x Dependencies

PCAN-UDS 2.x API is built on the PCAN-ISO-TP 3.x API, so it has some dependencies. These dependencies are described in this chapter.

Aliases

| | Alias | Description |
|---|-----------------|---------------------------------------|
|  | cantp_bitrate | Represents a PCAN FD bit rate string. |
|  | cantp_timestamp | Defines the timestamp of a message. |

Enumerations

| | Name | Description |
|---|------------------------|--|
|  | cantp_handle | Represents a PCANTP channel or a PUDS channel. |
|  | cantp_hwtype | Represents the type of PCAN hardware to be initialized. |
|  | cantp_isotp_addressing | Represents the type of message addressing of a PCANTP message. |
|  | cantp_baudrate | Represents a PCAN Baud rate register value for the PCANTP channel. |
|  | cantp_msg | Defines a PCANTP message. |
|  | cantp_can_msgtype | Represents the flags of a CAN or CAN FD message. |
|  | cantp_msgtype | Represents the type of a PCANTP message. |
|  | cantp_can_info | Represents common CAN information. |
|  | cantp_msgdata | Represents the content of a generic PCANTP message. |
|  | cantp_msgdata_can | Represents the content of a standard CAN message. |
|  | cantp_msgdata_canfd | Represents the content of a CAN FD message. |
|  | cantp_msgdata_isotp | Represents the content of an ISO TP message. |
|  | cantp_msgflag | Represents the flags common to all types of cantp_msg. |
|  | cantp_netstatus | Represents the network result of the communication of an PCANTP message. |
|  | cantp_netaddrinfo | Represents the network address information of an PCANTP message. |
|  | cantp_isotp_msgtype | Represents the addressing format of an PCANTP message. |
|  | cantp_isotp_format | Represents the addressing format of an PCANTP message. |

3.6.1 cantp_bitrate

Represents a bit rate string with flexible data rate (FD) for PCANTP or PUDS channels.

Syntax

C/C++

```
#define cantp_bitrate char*
```

Pascal OO

```
cantp_bitrate = PAnsiChar;
```

C#

```
using cantp_bitrate = String;
```

C++ / CLR

```
using cantp_bitrate = String^;
```

Visual Basic

```
Imports cantp_bitrate = System.String
```

Remarks

.NET Framework programming languages:

An alias is used to represent a flexible data rate under Microsoft .NET to originate a homogeneity between all programming languages listed above.

Aliases are defined in the `Peak.Can.IsoTp` Namespace for C# and VB .NET. However, including a namespace does not include the defined aliases.

If it is wished to work with aliases, those must be copied to the working file, right after the inclusion of the `Peak.Can.IsoTp` Namespace. Otherwise, just use the native type, which in this case is a string.

C#

```
using System;
using Peak.Can.IsoTp;
using cantp_bitrate = System.String; // Alias' declaration for System.String
```

Visual Basic

```
Imports System
Imports Peak.Can.IsoTp
Imports cantp_bitrate = System.String ' Alias declaration for System.String
```

FD Bit Rate Parameter Definitions

Defines the different configuration parameters used to create a flexible data rate string for FD capable PCAN channel initialization. These values are used as parameter with `UDS_InitializeFD_2013` (class method version: `InitializeFD_2013`).

Clock frequency parameters:

| Type | Constant | Value | Description |
|--------|---------------------|---------------|---|
| String | PCANTP_BR_CLOCK | "f_clock" | Clock frequency in Hertz (80000000, 60000000, 40000000, 30000000, 24000000, 20000000) |
| String | PCANTP_BR_CLOCK_MHZ | "f_clock_mhz" | Clock frequency in Megahertz (80, 60, 40, 30, 24, 20) |

Nominal bit rate parameters:

| Type | Constant | Value | Description |
|--------|---------------------|-------------|--|
| String | PCANTP_BR_NOM_BRP | "nom_brp" | Clock prescaler for nominal time quantum (1..1024). |
| String | PCANTP_BR_NOM_TSEG1 | "nom_tseg1" | TSEG1 segment for nominal bit rate in time quanta (1..256). |
| String | PCANTP_BR_NOM_TSEG2 | "nom_tseg2" | TSEG2 segment for nominal bit rate in time quanta (1..128). |
| String | PCANTP_BR_NOM_SJW | "nom_sjw" | Synchronization Jump Width for nominal bit rate in time quanta (1..128). |

Data bit rate parameters:

| Type | Constant | Value | Description |
|--------|----------------------|--------------|---|
| String | PCANTP_BR_DATA_BRP | "data_brp" | Clock prescaler for fast data time quantum (1..1024). |
| String | PCANTP_BR_DATA_TSEG1 | "data_tseg1" | TSEG1 segment for fast data bit rate in time quanta (1..32). |
| String | PCANTP_BR_DATA_TSEG2 | "data_tseg2" | TSEG2 segment for fast data bit rate in time quanta (1..16). |
| String | PCANTP_BR_DATA_SJW | "data_sjw" | Synchronization Jump Width for fast data bit rate in time quanta (1..16). |

Remarks

These definitions are constant values in an object-oriented environment (Delphi, .NET Framework) and declared as defines in C/C++ (plain API, see Functions on page 621).

Following points are to be respected to construct a valid FD bit rate string:

- └ The string must contain only one of the two possible clock frequency parameters, depending on the unit used (Hz, or MHz).
- └ The frequency to use must be one of the 6 listed within the clock frequency parameters.
- └ The value for each parameter must be separated with a '='. **Example:** "data_brp=1"
- └ Each pair of parameter/value must be separated with a ','. Blank spaces are allowed but are not necessary. Example: "f_clock_mhz=24, nom_brp=1,"
- └ Both bit rates, or only the nominal one, must be defined within the string (`PCANTP_BR_DATA_*` and `PCANTP_BR_NOM_*`, or only `PCANTP_BR_NOM_*`).

Example with nominal bit rate only:

A valid string representing 1 Mbit/sec for both, nominal and data bit rates:

"f_clock_mhz=20, nom_brp=5, nom_tseg1=2, nom_tseg2=1, nom_sjw=1"

Example with nominal and data bit rate:

A valid string representing 1 Mbit/sec for nominal bit rate, and 2 Mbit/sec for data bit rate:

"f_clock_mhz=20, nom_brp=5, nom_tseg1=2, nom_tseg2=1, nom_sjw=1, data_brp=2, data_tseg1=3, data_tseg2=1, data_sjw=1"

Parameter value ranges:

| Parameter | Value Range |
|-------------|--|
| f_clock | [80000000, 60000000, 40000000, 30000000, 24000000, 20000000] |
| f_clock_mhz | [80, 60, 40, 30, 24, 20] |
| nom_brp | 1 .. 1024 |
| nom_tseg1 | 1 .. 256 |
| nom_tseg2 | 1 .. 128 |
| nom_sjw | 1 .. 128 |
| data_brp | 1 .. 1024 |
| data_tseg1 | 1 .. 32 |
| data_tseg2 | 1 .. 16 |
| data_sjw | 1 .. 16 |

See Also: `UDS_InitializeFD_2013` on page 626, **class method version:** `InitializeFD_2013` on page 148.

3.6.2 cantp_timestamp

Represents a timestamp in PCAN-ISO-TP 3.x API and PCAN-UDS 2.x API.

Syntax

C/C++

```
#define cantp_timestamp uint64_t
```

Pascal OO

```
cantp_timestamp = uint64;
```

C#

```
using cantp_timestamp = UInt64;
```

C++ / CLR

```
using cantp_timestamp = UInt64;
```

Visual Basic

```
Imports cantp_timestamp = System.UInt64
```

Remarks

.NET Framework programming languages:

An alias is used to represent a flexible data rate under Microsoft .NET to originate a homogeneity between all programming languages listed above.

Aliases are defined in the `Peak.Can.IsoTp` Namespace for C# and VB .NET. However, including a namespace does not include the defined aliases.

If it is wished to work with aliases, those must be copied to the working file, right after the inclusion of the `Peak.Can.IsoTp` Namespace. Otherwise, just use the native type.

C#

```
using System;  
using Peak.Can.IsoTp;  
using cantp_timestamp = System.UInt64;
```

Visual Basic

```
Imports System  
Imports Peak.Can.IsoTp  
Imports cantp_timestamp = System.UInt64
```


3.6.3 cantp_handle

Represents currently defined and supported PCANTP handle (a.k.a. channels). These identifiers are also used to identify PUDS channels.

Syntax

C/C++

```
#define PCAN_NONEBUS          0x00U
#define PCAN_ISABUS1          0x21U
#define PCAN_ISABUS2          0x22U
#define PCAN_ISABUS3          0x23U
#define PCAN_ISABUS4          0x24U
#define PCAN_ISABUS5          0x25U
#define PCAN_ISABUS6          0x26U
#define PCAN_ISABUS7          0x27U
#define PCAN_ISABUS8          0x28U
#define PCAN_DNGBUS1          0x31U
#define PCAN_PCIBUS1          0x41U
#define PCAN_PCIBUS2          0x42U
#define PCAN_PCIBUS3          0x43U
#define PCAN_PCIBUS4          0x44U
#define PCAN_PCIBUS5          0x45U
#define PCAN_PCIBUS6          0x46U
#define PCAN_PCIBUS7          0x47U
#define PCAN_PCIBUS8          0x48U
#define PCAN_PCIBUS9          0x409U
#define PCAN_PCIBUS10         0x40AU
#define PCAN_PCIBUS11         0x40BU
#define PCAN_PCIBUS12         0x40CU
#define PCAN_PCIBUS13         0x40DU
#define PCAN_PCIBUS14         0x40EU
#define PCAN_PCIBUS15         0x40FU
#define PCAN_PCIBUS16         0x410U
#define PCAN_USBBUS1          0x51U
#define PCAN_USBBUS2          0x52U
#define PCAN_USBBUS3          0x53U
#define PCAN_USBBUS4          0x54U
#define PCAN_USBBUS5          0x55U
#define PCAN_USBBUS6          0x56U
#define PCAN_USBBUS7          0x57U
#define PCAN_USBBUS8          0x58U
#define PCAN_USBBUS9          0x509U
#define PCAN_USBBUS10         0x50AU
#define PCAN_USBBUS11         0x50BU
#define PCAN_USBBUS12         0x50CU
#define PCAN_USBBUS13         0x50DU
#define PCAN_USBBUS14         0x50EU
#define PCAN_USBBUS15         0x50FU
#define PCAN_USBBUS16         0x510U
#define PCAN_PCCBUS1          0x61U
#define PCAN_PCCBUS2          0x62U
#define PCAN_LANBUS1          0x801U
#define PCAN_LANBUS2          0x802U
#define PCAN_LANBUS3          0x803U
#define PCAN_LANBUS4          0x804U
#define PCAN_LANBUS5          0x805U
#define PCAN_LANBUS6          0x806U
#define PCAN_LANBUS7          0x807U
#define PCAN_LANBUS8          0x808U
#define PCAN_LANBUS9          0x809U
#define PCAN_LANBUS10         0x80AU
```

```

#define PCAN_LANBUS1          0x80BU
#define PCAN_LANBUS2          0x80CU
#define PCAN_LANBUS3          0x80DU
#define PCAN_LANBUS4          0x80EU
#define PCAN_LANBUS5          0x80FU
#define PCAN_LANBUS16         0x810U

typedef enum _cantp_handle {
    PCANTP_HANDLE_NONEBUS = PCAN_NONEBUS,

    PCANTP_HANDLE_ISABUS1 = PCAN_ISABUS1,
    PCANTP_HANDLE_ISABUS2 = PCAN_ISABUS2,
    PCANTP_HANDLE_ISABUS3 = PCAN_ISABUS3,
    PCANTP_HANDLE_ISABUS4 = PCAN_ISABUS4,
    PCANTP_HANDLE_ISABUS5 = PCAN_ISABUS5,
    PCANTP_HANDLE_ISABUS6 = PCAN_ISABUS6,
    PCANTP_HANDLE_ISABUS7 = PCAN_ISABUS7,
    PCANTP_HANDLE_ISABUS8 = PCAN_ISABUS8,

    PCANTP_HANDLE_DNGBUS1 = PCAN_DNGBUS1,

    PCANTP_HANDLE_PCIBUS1 = PCAN_PCIBUS1,
    PCANTP_HANDLE_PCIBUS2 = PCAN_PCIBUS2,
    PCANTP_HANDLE_PCIBUS3 = PCAN_PCIBUS3,
    PCANTP_HANDLE_PCIBUS4 = PCAN_PCIBUS4,
    PCANTP_HANDLE_PCIBUS5 = PCAN_PCIBUS5,
    PCANTP_HANDLE_PCIBUS6 = PCAN_PCIBUS6,
    PCANTP_HANDLE_PCIBUS7 = PCAN_PCIBUS7,
    PCANTP_HANDLE_PCIBUS8 = PCAN_PCIBUS8,
    PCANTP_HANDLE_PCIBUS9 = PCAN_PCIBUS9,
    PCANTP_HANDLE_PCIBUS10 = PCAN_PCIBUS10,
    PCANTP_HANDLE_PCIBUS11 = PCAN_PCIBUS11,
    PCANTP_HANDLE_PCIBUS12 = PCAN_PCIBUS12,
    PCANTP_HANDLE_PCIBUS13 = PCAN_PCIBUS13,
    PCANTP_HANDLE_PCIBUS14 = PCAN_PCIBUS14,
    PCANTP_HANDLE_PCIBUS15 = PCAN_PCIBUS15,
    PCANTP_HANDLE_PCIBUS16 = PCAN_PCIBUS16,

    PCANTP_HANDLE_USBBUS1 = PCAN_USBBUS1,
    PCANTP_HANDLE_USBBUS2 = PCAN_USBBUS2,
    PCANTP_HANDLE_USBBUS3 = PCAN_USBBUS3,
    PCANTP_HANDLE_USBBUS4 = PCAN_USBBUS4,
    PCANTP_HANDLE_USBBUS5 = PCAN_USBBUS5,
    PCANTP_HANDLE_USBBUS6 = PCAN_USBBUS6,
    PCANTP_HANDLE_USBBUS7 = PCAN_USBBUS7,
    PCANTP_HANDLE_USBBUS8 = PCAN_USBBUS8,
    PCANTP_HANDLE_USBBUS9 = PCAN_USBBUS9,
    PCANTP_HANDLE_USBBUS10 = PCAN_USBBUS10,
    PCANTP_HANDLE_USBBUS11 = PCAN_USBBUS11,
    PCANTP_HANDLE_USBBUS12 = PCAN_USBBUS12,
    PCANTP_HANDLE_USBBUS13 = PCAN_USBBUS13,
    PCANTP_HANDLE_USBBUS14 = PCAN_USBBUS14,
    PCANTP_HANDLE_USBBUS15 = PCAN_USBBUS15,
    PCANTP_HANDLE_USBBUS16 = PCAN_USBBUS16,

    PCANTP_HANDLE_PCCBUS1 = PCAN_PCCBUS1,
    PCANTP_HANDLE_PCCBUS2 = PCAN_PCCBUS2,

    PCANTP_HANDLE_LANBUS1 = PCAN_LANBUS1,
    PCANTP_HANDLE_LANBUS2 = PCAN_LANBUS2,
    PCANTP_HANDLE_LANBUS3 = PCAN_LANBUS3,
    PCANTP_HANDLE_LANBUS4 = PCAN_LANBUS4,

```

```

PCANTP_HANDLE_LANBUS5 = PCAN_LANBUS5,
PCANTP_HANDLE_LANBUS6 = PCAN_LANBUS6,
PCANTP_HANDLE_LANBUS7 = PCAN_LANBUS7,
PCANTP_HANDLE_LANBUS8 = PCAN_LANBUS8,
PCANTP_HANDLE_LANBUS9 = PCAN_LANBUS9,
PCANTP_HANDLE_LANBUS10 = PCAN_LANBUS10,
PCANTP_HANDLE_LANBUS11 = PCAN_LANBUS11,
PCANTP_HANDLE_LANBUS12 = PCAN_LANBUS12,
PCANTP_HANDLE_LANBUS13 = PCAN_LANBUS13,
PCANTP_HANDLE_LANBUS14 = PCAN_LANBUS14,
PCANTP_HANDLE_LANBUS15 = PCAN_LANBUS15,
PCANTP_HANDLE_LANBUS16 = PCAN_LANBUS16,
} cantp_handle;

```

C++ / CLR

```

public enum cantp_handle : UInt32
{
    PCANTP_HANDLE_NONEBUS = PCANBasic::PCAN_NONEBUS,
    PCANTP_HANDLE_ISABUS1 = PCANBasic::PCAN_ISABUS1,
    PCANTP_HANDLE_ISABUS2 = PCANBasic::PCAN_ISABUS2,
    PCANTP_HANDLE_ISABUS3 = PCANBasic::PCAN_ISABUS3,
    PCANTP_HANDLE_ISABUS4 = PCANBasic::PCAN_ISABUS4,
    PCANTP_HANDLE_ISABUS5 = PCANBasic::PCAN_ISABUS5,
    PCANTP_HANDLE_ISABUS6 = PCANBasic::PCAN_ISABUS6,
    PCANTP_HANDLE_ISABUS7 = PCANBasic::PCAN_ISABUS7,
    PCANTP_HANDLE_ISABUS8 = PCANBasic::PCAN_ISABUS8,
    PCANTP_HANDLE_DNGBUS1 = PCANBasic::PCAN_DNGBUS1,
    PCANTP_HANDLE_PCIBUS1 = PCANBasic::PCAN_PCIBUS1,
    PCANTP_HANDLE_PCIBUS2 = PCANBasic::PCAN_PCIBUS2,
    PCANTP_HANDLE_PCIBUS3 = PCANBasic::PCAN_PCIBUS3,
    PCANTP_HANDLE_PCIBUS4 = PCANBasic::PCAN_PCIBUS4,
    PCANTP_HANDLE_PCIBUS5 = PCANBasic::PCAN_PCIBUS5,
    PCANTP_HANDLE_PCIBUS6 = PCANBasic::PCAN_PCIBUS6,
    PCANTP_HANDLE_PCIBUS7 = PCANBasic::PCAN_PCIBUS7,
    PCANTP_HANDLE_PCIBUS8 = PCANBasic::PCAN_PCIBUS8,
    PCANTP_HANDLE_PCIBUS9 = PCANBasic::PCAN_PCIBUS9,
    PCANTP_HANDLE_PCIBUS10 = PCANBasic::PCAN_PCIBUS10,
    PCANTP_HANDLE_PCIBUS11 = PCANBasic::PCAN_PCIBUS11,
    PCANTP_HANDLE_PCIBUS12 = PCANBasic::PCAN_PCIBUS12,
    PCANTP_HANDLE_PCIBUS13 = PCANBasic::PCAN_PCIBUS13,
    PCANTP_HANDLE_PCIBUS14 = PCANBasic::PCAN_PCIBUS14,
    PCANTP_HANDLE_PCIBUS15 = PCANBasic::PCAN_PCIBUS15,
    PCANTP_HANDLE_PCIBUS16 = PCANBasic::PCAN_PCIBUS16,
    PCANTP_HANDLE_USBBUS1 = PCANBasic::PCAN_USBBUS1,
    PCANTP_HANDLE_USBBUS2 = PCANBasic::PCAN_USBBUS2,
    PCANTP_HANDLE_USBBUS3 = PCANBasic::PCAN_USBBUS3,
    PCANTP_HANDLE_USBBUS4 = PCANBasic::PCAN_USBBUS4,
    PCANTP_HANDLE_USBBUS5 = PCANBasic::PCAN_USBBUS5,
    PCANTP_HANDLE_USBBUS6 = PCANBasic::PCAN_USBBUS6,
    PCANTP_HANDLE_USBBUS7 = PCANBasic::PCAN_USBBUS7,
    PCANTP_HANDLE_USBBUS8 = PCANBasic::PCAN_USBBUS8,
    PCANTP_HANDLE_USBBUS9 = PCANBasic::PCAN_USBBUS9,
    PCANTP_HANDLE_USBBUS10 = PCANBasic::PCAN_USBBUS10,
    PCANTP_HANDLE_USBBUS11 = PCANBasic::PCAN_USBBUS11,
    PCANTP_HANDLE_USBBUS12 = PCANBasic::PCAN_USBBUS12,
    PCANTP_HANDLE_USBBUS13 = PCANBasic::PCAN_USBBUS13,
    PCANTP_HANDLE_USBBUS14 = PCANBasic::PCAN_USBBUS14,
    PCANTP_HANDLE_USBBUS15 = PCANBasic::PCAN_USBBUS15,
    PCANTP_HANDLE_USBBUS16 = PCANBasic::PCAN_USBBUS16,
    PCANTP_HANDLE_PCCBUS1 = PCANBasic::PCAN_PCCBUS1,
    PCANTP_HANDLE_PCCBUS2 = PCANBasic::PCAN_PCCBUS2,
}

```

```

PCANTP_HANDLE_LANBUS1 = PCANBasic::PCAN_LANBUS1,
PCANTP_HANDLE_LANBUS2 = PCANBasic::PCAN_LANBUS2,
PCANTP_HANDLE_LANBUS3 = PCANBasic::PCAN_LANBUS3,
PCANTP_HANDLE_LANBUS4 = PCANBasic::PCAN_LANBUS4,
PCANTP_HANDLE_LANBUS5 = PCANBasic::PCAN_LANBUS5,
PCANTP_HANDLE_LANBUS6 = PCANBasic::PCAN_LANBUS6,
PCANTP_HANDLE_LANBUS7 = PCANBasic::PCAN_LANBUS7,
PCANTP_HANDLE_LANBUS8 = PCANBasic::PCAN_LANBUS8,
PCANTP_HANDLE_LANBUS9 = PCANBasic::PCAN_LANBUS9,
PCANTP_HANDLE_LANBUS10 = PCANBasic::PCAN_LANBUS10,
PCANTP_HANDLE_LANBUS11 = PCANBasic::PCAN_LANBUS11,
PCANTP_HANDLE_LANBUS12 = PCANBasic::PCAN_LANBUS12,
PCANTP_HANDLE_LANBUS13 = PCANBasic::PCAN_LANBUS13,
PCANTP_HANDLE_LANBUS14 = PCANBasic::PCAN_LANBUS14,
PCANTP_HANDLE_LANBUS15 = PCANBasic::PCAN_LANBUS15,
PCANTP_HANDLE_LANBUS16 = PCANBasic::PCAN_LANBUS16,

```

```
};
```

C#

```

public enum cantp_handle : UInt32
{
    PCANTP_HANDLE_NONEBUS = PCANBasic.PCAN_NONEBUS,
    PCANTP_HANDLE_ISABUS1 = PCANBasic.PCAN_ISABUS1,
    PCANTP_HANDLE_ISABUS2 = PCANBasic.PCAN_ISABUS2,
    PCANTP_HANDLE_ISABUS3 = PCANBasic.PCAN_ISABUS3,
    PCANTP_HANDLE_ISABUS4 = PCANBasic.PCAN_ISABUS4,
    PCANTP_HANDLE_ISABUS5 = PCANBasic.PCAN_ISABUS5,
    PCANTP_HANDLE_ISABUS6 = PCANBasic.PCAN_ISABUS6,
    PCANTP_HANDLE_ISABUS7 = PCANBasic.PCAN_ISABUS7,
    PCANTP_HANDLE_ISABUS8 = PCANBasic.PCAN_ISABUS8,
    PCANTP_HANDLE_DNGBUS1 = PCANBasic.PCAN_DNGBUS1,
    PCANTP_HANDLE_PCIBUS1 = PCANBasic.PCAN_PCIBUS1,
    PCANTP_HANDLE_PCIBUS2 = PCANBasic.PCAN_PCIBUS2,
    PCANTP_HANDLE_PCIBUS3 = PCANBasic.PCAN_PCIBUS3,
    PCANTP_HANDLE_PCIBUS4 = PCANBasic.PCAN_PCIBUS4,
    PCANTP_HANDLE_PCIBUS5 = PCANBasic.PCAN_PCIBUS5,
    PCANTP_HANDLE_PCIBUS6 = PCANBasic.PCAN_PCIBUS6,
    PCANTP_HANDLE_PCIBUS7 = PCANBasic.PCAN_PCIBUS7,
    PCANTP_HANDLE_PCIBUS8 = PCANBasic.PCAN_PCIBUS8,
    PCANTP_HANDLE_PCIBUS9 = PCANBasic.PCAN_PCIBUS9,
    PCANTP_HANDLE_PCIBUS10 = PCANBasic.PCAN_PCIBUS10,
    PCANTP_HANDLE_PCIBUS11 = PCANBasic.PCAN_PCIBUS11,
    PCANTP_HANDLE_PCIBUS12 = PCANBasic.PCAN_PCIBUS12,
    PCANTP_HANDLE_PCIBUS13 = PCANBasic.PCAN_PCIBUS13,
    PCANTP_HANDLE_PCIBUS14 = PCANBasic.PCAN_PCIBUS14,
    PCANTP_HANDLE_PCIBUS15 = PCANBasic.PCAN_PCIBUS15,
    PCANTP_HANDLE_PCIBUS16 = PCANBasic.PCAN_PCIBUS16,
    PCANTP_HANDLE_USBBUS1 = PCANBasic.PCAN_USBBUS1,
    PCANTP_HANDLE_USBBUS2 = PCANBasic.PCAN_USBBUS2,
    PCANTP_HANDLE_USBBUS3 = PCANBasic.PCAN_USBBUS3,
    PCANTP_HANDLE_USBBUS4 = PCANBasic.PCAN_USBBUS4,
    PCANTP_HANDLE_USBBUS5 = PCANBasic.PCAN_USBBUS5,
    PCANTP_HANDLE_USBBUS6 = PCANBasic.PCAN_USBBUS6,
    PCANTP_HANDLE_USBBUS7 = PCANBasic.PCAN_USBBUS7,
    PCANTP_HANDLE_USBBUS8 = PCANBasic.PCAN_USBBUS8,
    PCANTP_HANDLE_USBBUS9 = PCANBasic.PCAN_USBBUS9,
    PCANTP_HANDLE_USBBUS10 = PCANBasic.PCAN_USBBUS10,
    PCANTP_HANDLE_USBBUS11 = PCANBasic.PCAN_USBBUS11,
    PCANTP_HANDLE_USBBUS12 = PCANBasic.PCAN_USBBUS12,
    PCANTP_HANDLE_USBBUS13 = PCANBasic.PCAN_USBBUS13,
    PCANTP_HANDLE_USBBUS14 = PCANBasic.PCAN_USBBUS14,

```

```

PCANTP_HANDLE_USBBUS15 = PCANBasic.PCAN_USBBUS15,
PCANTP_HANDLE_USBBUS16 = PCANBasic.PCAN_USBBUS16,
PCANTP_HANDLE_PCCBUS1 = PCANBasic.PCAN_PCCBUS1,
PCANTP_HANDLE_PCCBUS2 = PCANBasic.PCAN_PCCBUS2,
PCANTP_HANDLE_LANBUS1 = PCANBasic.PCAN_LANBUS1,
PCANTP_HANDLE_LANBUS2 = PCANBasic.PCAN_LANBUS2,
PCANTP_HANDLE_LANBUS3 = PCANBasic.PCAN_LANBUS3,
PCANTP_HANDLE_LANBUS4 = PCANBasic.PCAN_LANBUS4,
PCANTP_HANDLE_LANBUS5 = PCANBasic.PCAN_LANBUS5,
PCANTP_HANDLE_LANBUS6 = PCANBasic.PCAN_LANBUS6,
PCANTP_HANDLE_LANBUS7 = PCANBasic.PCAN_LANBUS7,
PCANTP_HANDLE_LANBUS8 = PCANBasic.PCAN_LANBUS8,
PCANTP_HANDLE_LANBUS9 = PCANBasic.PCAN_LANBUS9,
PCANTP_HANDLE_LANBUS10 = PCANBasic.PCAN_LANBUS10,
PCANTP_HANDLE_LANBUS11 = PCANBasic.PCAN_LANBUS11,
PCANTP_HANDLE_LANBUS12 = PCANBasic.PCAN_LANBUS12,
PCANTP_HANDLE_LANBUS13 = PCANBasic.PCAN_LANBUS13,
PCANTP_HANDLE_LANBUS14 = PCANBasic.PCAN_LANBUS14,
PCANTP_HANDLE_LANBUS15 = PCANBasic.PCAN_LANBUS15,
PCANTP_HANDLE_LANBUS16 = PCANBasic.PCAN_LANBUS16,
}

```

Pascal OO

```

cantp_handle = (
    PCANTP_HANDLE_NONEBUS = PCAN_NONEBUS,
    PCANTP_HANDLE_ISABUS1 = PCAN_ISABUS1,
    PCANTP_HANDLE_ISABUS2 = PCAN_ISABUS2,
    PCANTP_HANDLE_ISABUS3 = PCAN_ISABUS3,
    PCANTP_HANDLE_ISABUS4 = PCAN_ISABUS4,
    PCANTP_HANDLE_ISABUS5 = PCAN_ISABUS5,
    PCANTP_HANDLE_ISABUS6 = PCAN_ISABUS6,
    PCANTP_HANDLE_ISABUS7 = PCAN_ISABUS7,
    PCANTP_HANDLE_ISABUS8 = PCAN_ISABUS8,
    PCANTP_HANDLE_DNGBUS1 = PCAN_DNGBUS1,
    PCANTP_HANDLE_PCIBUS1 = PCAN_PCIBUS1,
    PCANTP_HANDLE_PCIBUS2 = PCAN_PCIBUS2,
    PCANTP_HANDLE_PCIBUS3 = PCAN_PCIBUS3,
    PCANTP_HANDLE_PCIBUS4 = PCAN_PCIBUS4,
    PCANTP_HANDLE_PCIBUS5 = PCAN_PCIBUS5,
    PCANTP_HANDLE_PCIBUS6 = PCAN_PCIBUS6,
    PCANTP_HANDLE_PCIBUS7 = PCAN_PCIBUS7,
    PCANTP_HANDLE_PCIBUS8 = PCAN_PCIBUS8,
    PCANTP_HANDLE_PCIBUS9 = PCAN_PCIBUS9,
    PCANTP_HANDLE_PCIBUS10 = PCAN_PCIBUS10,
    PCANTP_HANDLE_PCIBUS11 = PCAN_PCIBUS11,
    PCANTP_HANDLE_PCIBUS12 = PCAN_PCIBUS12,
    PCANTP_HANDLE_PCIBUS13 = PCAN_PCIBUS13,
    PCANTP_HANDLE_PCIBUS14 = PCAN_PCIBUS14,
    PCANTP_HANDLE_PCIBUS15 = PCAN_PCIBUS15,
    PCANTP_HANDLE_PCIBUS16 = PCAN_PCIBUS16,
    PCANTP_HANDLE_USBBUS1 = PCAN_USBBUS1,
    PCANTP_HANDLE_USBBUS2 = PCAN_USBBUS2,
    PCANTP_HANDLE_USBBUS3 = PCAN_USBBUS3,
    PCANTP_HANDLE_USBBUS4 = PCAN_USBBUS4,
    PCANTP_HANDLE_USBBUS5 = PCAN_USBBUS5,
    PCANTP_HANDLE_USBBUS6 = PCAN_USBBUS6,
    PCANTP_HANDLE_USBBUS7 = PCAN_USBBUS7,
    PCANTP_HANDLE_USBBUS8 = PCAN_USBBUS8,
    PCANTP_HANDLE_USBBUS9 = PCAN_USBBUS9,
    PCANTP_HANDLE_USBBUS10 = PCAN_USBBUS10,
    PCANTP_HANDLE_USBBUS11 = PCAN_USBBUS11,

```

```

PCANTP_HANDLE_USBBUS12 = PCAN_USBBUS12,
PCANTP_HANDLE_USBBUS13 = PCAN_USBBUS13,
PCANTP_HANDLE_USBBUS14 = PCAN_USBBUS14,
PCANTP_HANDLE_USBBUS15 = PCAN_USBBUS15,
PCANTP_HANDLE_USBBUS16 = PCAN_USBBUS16,
PCANTP_HANDLE_PCCBUS1 = PCAN_PCCBUS1,
PCANTP_HANDLE_PCCBUS2 = PCAN_PCCBUS2,
PCANTP_HANDLE_LANBUS1 = PCAN_LANBUS1,
PCANTP_HANDLE_LANBUS2 = PCAN_LANBUS2,
PCANTP_HANDLE_LANBUS3 = PCAN_LANBUS3,
PCANTP_HANDLE_LANBUS4 = PCAN_LANBUS4,
PCANTP_HANDLE_LANBUS5 = PCAN_LANBUS5,
PCANTP_HANDLE_LANBUS6 = PCAN_LANBUS6,
PCANTP_HANDLE_LANBUS7 = PCAN_LANBUS7,
PCANTP_HANDLE_LANBUS8 = PCAN_LANBUS8,
PCANTP_HANDLE_LANBUS9 = PCAN_LANBUS9,
PCANTP_HANDLE_LANBUS10 = PCAN_LANBUS10,
PCANTP_HANDLE_LANBUS11 = PCAN_LANBUS11,
PCANTP_HANDLE_LANBUS12 = PCAN_LANBUS12,
PCANTP_HANDLE_LANBUS13 = PCAN_LANBUS13,
PCANTP_HANDLE_LANBUS14 = PCAN_LANBUS14,
PCANTP_HANDLE_LANBUS15 = PCAN_LANBUS15,
PCANTP_HANDLE_LANBUS16 = PCAN_LANBUS16

```

```
);
```

Visual Basic

```

Public Enum cantp_handle As UInt32
    PCANTP_HANDLE_NONEBUS = PCANBasic.PCAN_NONEBUS
    PCANTP_HANDLE_ISABUS1 = PCANBasic.PCAN_ISABUS1
    PCANTP_HANDLE_ISABUS2 = PCANBasic.PCAN_ISABUS2
    PCANTP_HANDLE_ISABUS3 = PCANBasic.PCAN_ISABUS3
    PCANTP_HANDLE_ISABUS4 = PCANBasic.PCAN_ISABUS4
    PCANTP_HANDLE_ISABUS5 = PCANBasic.PCAN_ISABUS5
    PCANTP_HANDLE_ISABUS6 = PCANBasic.PCAN_ISABUS6
    PCANTP_HANDLE_ISABUS7 = PCANBasic.PCAN_ISABUS7
    PCANTP_HANDLE_ISABUS8 = PCANBasic.PCAN_ISABUS8
    PCANTP_HANDLE_DNGBUS1 = PCANBasic.PCAN_DNGBUS1
    PCANTP_HANDLE_PCIBUS1 = PCANBasic.PCAN_PCIBUS1
    PCANTP_HANDLE_PCIBUS2 = PCANBasic.PCAN_PCIBUS2
    PCANTP_HANDLE_PCIBUS3 = PCANBasic.PCAN_PCIBUS3
    PCANTP_HANDLE_PCIBUS4 = PCANBasic.PCAN_PCIBUS4
    PCANTP_HANDLE_PCIBUS5 = PCANBasic.PCAN_PCIBUS5
    PCANTP_HANDLE_PCIBUS6 = PCANBasic.PCAN_PCIBUS6
    PCANTP_HANDLE_PCIBUS7 = PCANBasic.PCAN_PCIBUS7
    PCANTP_HANDLE_PCIBUS8 = PCANBasic.PCAN_PCIBUS8
    PCANTP_HANDLE_PCIBUS9 = PCANBasic.PCAN_PCIBUS9
    PCANTP_HANDLE_PCIBUS10 = PCANBasic.PCAN_PCIBUS10
    PCANTP_HANDLE_PCIBUS11 = PCANBasic.PCAN_PCIBUS11
    PCANTP_HANDLE_PCIBUS12 = PCANBasic.PCAN_PCIBUS12
    PCANTP_HANDLE_PCIBUS13 = PCANBasic.PCAN_PCIBUS13
    PCANTP_HANDLE_PCIBUS14 = PCANBasic.PCAN_PCIBUS14
    PCANTP_HANDLE_PCIBUS15 = PCANBasic.PCAN_PCIBUS15
    PCANTP_HANDLE_PCIBUS16 = PCANBasic.PCAN_PCIBUS16
    PCANTP_HANDLE_USBBUS1 = PCANBasic.PCAN_USBBUS1
    PCANTP_HANDLE_USBBUS2 = PCANBasic.PCAN_USBBUS2
    PCANTP_HANDLE_USBBUS3 = PCANBasic.PCAN_USBBUS3
    PCANTP_HANDLE_USBBUS4 = PCANBasic.PCAN_USBBUS4
    PCANTP_HANDLE_USBBUS5 = PCANBasic.PCAN_USBBUS5
    PCANTP_HANDLE_USBBUS6 = PCANBasic.PCAN_USBBUS6
    PCANTP_HANDLE_USBBUS7 = PCANBasic.PCAN_USBBUS7
    PCANTP_HANDLE_USBBUS8 = PCANBasic.PCAN_USBBUS8

```



```

PCANTP_HANDLE_USBBUS9 = PCANBasic.PCAN_USBBUS9
PCANTP_HANDLE_USBBUS10 = PCANBasic.PCAN_USBBUS10
PCANTP_HANDLE_USBBUS11 = PCANBasic.PCAN_USBBUS11
PCANTP_HANDLE_USBBUS12 = PCANBasic.PCAN_USBBUS12
PCANTP_HANDLE_USBBUS13 = PCANBasic.PCAN_USBBUS13
PCANTP_HANDLE_USBBUS14 = PCANBasic.PCAN_USBBUS14
PCANTP_HANDLE_USBBUS15 = PCANBasic.PCAN_USBBUS15
PCANTP_HANDLE_USBBUS16 = PCANBasic.PCAN_USBBUS16
PCANTP_HANDLE_PCCBUS1 = PCANBasic.PCAN_PCCBUS1
PCANTP_HANDLE_PCCBUS2 = PCANBasic.PCAN_PCCBUS2
PCANTP_HANDLE_LANBUS1 = PCANBasic.PCAN_LANBUS1
PCANTP_HANDLE_LANBUS2 = PCANBasic.PCAN_LANBUS2
PCANTP_HANDLE_LANBUS3 = PCANBasic.PCAN_LANBUS3
PCANTP_HANDLE_LANBUS4 = PCANBasic.PCAN_LANBUS4
PCANTP_HANDLE_LANBUS5 = PCANBasic.PCAN_LANBUS5
PCANTP_HANDLE_LANBUS6 = PCANBasic.PCAN_LANBUS6
PCANTP_HANDLE_LANBUS7 = PCANBasic.PCAN_LANBUS7
PCANTP_HANDLE_LANBUS8 = PCANBasic.PCAN_LANBUS8
PCANTP_HANDLE_LANBUS9 = PCANBasic.PCAN_LANBUS9
PCANTP_HANDLE_LANBUS10 = PCANBasic.PCAN_LANBUS10
PCANTP_HANDLE_LANBUS11 = PCANBasic.PCAN_LANBUS11
PCANTP_HANDLE_LANBUS12 = PCANBasic.PCAN_LANBUS12
PCANTP_HANDLE_LANBUS13 = PCANBasic.PCAN_LANBUS13
PCANTP_HANDLE_LANBUS14 = PCANBasic.PCAN_LANBUS14
PCANTP_HANDLE_LANBUS15 = PCANBasic.PCAN_LANBUS15
PCANTP_HANDLE_LANBUS16 = PCANBasic.PCAN_LANBUS16


```

End Enum









Definitions

Defines the handles for the different PCAN buses (channels) within a class. The values are used as parameters where a `cantp_handle` is needed.


Default/Undefined Handle:

| | Name | Value | Description |
|---|-----------------------|-------|---|
|  | PCANTP_HANDLE_NONEBUS | 0x0 | Undefined/default value for a PCAN-ISO-TP 3.x channel |

















Handles for the ISA bus (Non-Plug and Play):

| | Name | Value | Description |
|---|-----------------------|-------|-------------------------------|
|  | PCANTP_HANDLE_ISABUS1 | 0x21 | PCAN-ISA interface, channel 1 |
|  | PCANTP_HANDLE_ISABUS2 | 0x22 | PCAN-ISA interface, channel 2 |
|  | PCANTP_HANDLE_ISABUS3 | 0x23 | PCAN-ISA interface, channel 3 |
|  | PCANTP_HANDLE_ISABUS4 | 0x24 | PCAN-ISA interface, channel 4 |
|  | PCANTP_HANDLE_ISABUS5 | 0x25 | PCAN-ISA interface, channel 5 |
|  | PCANTP_HANDLE_ISABUS6 | 0x26 | PCAN-ISA interface, channel 6 |
|  | PCANTP_HANDLE_ISABUS7 | 0x27 | PCAN-ISA interface, channel 7 |
|  | PCANTP_HANDLE_ISABUS8 | 0x28 | PCAN-ISA interface, channel 8 |

















Handles for the Dongle Bus (Non-Plug and Play):

| | Name | Value | Description |
|---|-----------------------|-------|--------------------------------------|
|  | PCANTP_HANDLE_DNGBUS1 | 0x31 | PCAN-Dongle/LPT interface, channel 1 |



Handles for the PCI bus:

| | Name | Value | Description |
|---|------------------------|-------|--------------------------------|
|  | PCANTP_HANDLE_PCIBUS1 | 0x41 | PCAN-PCI interface, channel 1 |
|  | PCANTP_HANDLE_PCIBUS2 | 0x42 | PCAN-PCI interface, channel 2 |
|  | PCANTP_HANDLE_PCIBUS3 | 0x43 | PCAN-PCI interface, channel 3 |
|  | PCANTP_HANDLE_PCIBUS4 | 0x44 | PCAN-PCI interface, channel 4 |
|  | PCANTP_HANDLE_PCIBUS5 | 0x45 | PCAN-PCI interface, channel 5 |
|  | PCANTP_HANDLE_PCIBUS6 | 0x46 | PCAN-PCI interface, channel 6 |
|  | PCANTP_HANDLE_PCIBUS7 | 0x47 | PCAN-PCI interface, channel 7 |
|  | PCANTP_HANDLE_PCIBUS8 | 0x48 | PCAN-PCI interface, channel 8 |
|  | PCANTP_HANDLE_PCIBUS9 | 0x409 | PCAN-PCI interface, channel 9 |
|  | PCANTP_HANDLE_PCIBUS10 | 0x40A | PCAN-PCI interface, channel 10 |
|  | PCANTP_HANDLE_PCIBUS11 | 0x40B | PCAN-PCI interface, channel 11 |
|  | PCANTP_HANDLE_PCIBUS12 | 0x40C | PCAN-PCI interface, channel 12 |
|  | PCANTP_HANDLE_PCIBUS13 | 0x40D | PCAN-PCI interface, channel 13 |
|  | PCANTP_HANDLE_PCIBUS14 | 0x40E | PCAN-PCI interface, channel 14 |
|  | PCANTP_HANDLE_PCIBUS15 | 0x40F | PCAN-PCI interface, channel 15 |
|  | PCANTP_HANDLE_PCIBUS16 | 0x410 | PCAN-PCI interface, channel 16 |



Handles for the USB Bus:















| | Name | Value | Description |
|---|------------------------|-------|--------------------------------|
|  | PCANTP_HANDLE_USBBUS1 | 0x51 | PCAN-USB interface, channel 1 |
|  | PCANTP_HANDLE_USBBUS2 | 0x52 | PCAN-USB interface, channel 2 |
|  | PCANTP_HANDLE_USBBUS3 | 0x53 | PCAN-USB interface, channel 3 |
|  | PCANTP_HANDLE_USBBUS4 | 0x54 | PCAN-USB interface, channel 4 |
|  | PCANTP_HANDLE_USBBUS5 | 0x55 | PCAN-USB interface, channel 5 |
|  | PCANTP_HANDLE_USBBUS6 | 0x56 | PCAN-USB interface, channel 6 |
|  | PCANTP_HANDLE_USBBUS7 | 0x57 | PCAN-USB interface, channel 7 |
|  | PCANTP_HANDLE_USBBUS8 | 0x58 | PCAN-USB interface, channel 8 |
|  | PCANTP_HANDLE_USBBUS9 | 0x509 | PCAN-USB interface, channel 9 |
|  | PCANTP_HANDLE_USBBUS10 | 0x50A | PCAN-USB interface, channel 10 |
|  | PCANTP_HANDLE_USBBUS11 | 0x50B | PCAN-USB interface, channel 11 |
|  | PCANTP_HANDLE_USBBUS12 | 0x50C | PCAN-USB interface, channel 12 |
|  | PCANTP_HANDLE_USBBUS13 | 0x50D | PCAN-USB interface, channel 13 |
|  | PCANTP_HANDLE_USBBUS14 | 0x50E | PCAN-USB interface, channel 14 |
|  | PCANTP_HANDLE_USBBUS15 | 0x50F | PCAN-USB interface, channel 15 |
|  | PCANTP_HANDLE_USBBUS16 | 0x510 | PCAN-USB interface, channel 16 |

Handles for the PC Card Bus:

| | Name | Value | Description |
|---|-----------------------|-------|-----------------------------------|
|  | PCANTP_HANDLE_PCCBUS1 | 0x61 | PCAN-PC Card interface, channel 1 |
|  | PCANTP_HANDLE_PCCBUS2 | 0x62 | PCAN-PC Card interface, channel 2 |

Handles for the LAN interface:

| | Name | Value | Description |
|---|-----------------------|-------|-------------------------------|
|  | PCANTP_HANDLE_LANBUS1 | 0x801 | PCAN-LAN interface, channel 1 |
|  | PCANTP_HANDLE_LANBUS2 | 0x802 | PCAN-LAN interface, channel 2 |

| | Name | Value | Description |
|---|------------------------|-------|--------------------------------|
|  | PCANTP_HANDLE_LANBUS3 | 0x803 | PCAN-LAN interface, channel 3 |
|  | PCANTP_HANDLE_LANBUS4 | 0x804 | PCAN-LAN interface, channel 4 |
|  | PCANTP_HANDLE_LANBUS5 | 0x805 | PCAN-LAN interface, channel 5 |
|  | PCANTP_HANDLE_LANBUS6 | 0x806 | PCAN-LAN interface, channel 6 |
|  | PCANTP_HANDLE_LANBUS7 | 0x807 | PCAN-LAN interface, channel 7 |
|  | PCANTP_HANDLE_LANBUS8 | 0x808 | PCAN-LAN interface, channel 8 |
|  | PCANTP_HANDLE_LANBUS9 | 0x809 | PCAN-LAN interface, channel 9 |
|  | PCANTP_HANDLE_LANBUS10 | 0x80A | PCAN-LAN interface, channel 10 |
|  | PCANTP_HANDLE_LANBUS11 | 0x80B | PCAN-LAN interface, channel 11 |
|  | PCANTP_HANDLE_LANBUS12 | 0x80C | PCAN-LAN interface, channel 12 |
|  | PCANTP_HANDLE_LANBUS13 | 0x80D | PCAN-LAN interface, channel 13 |
|  | PCANTP_HANDLE_LANBUS14 | 0x80E | PCAN-LAN interface, channel 14 |
|  | PCANTP_HANDLE_LANBUS15 | 0x80F | PCAN-LAN interface, channel 15 |
|  | PCANTP_HANDLE_LANBUS16 | 0x810 | PCAN-LAN interface, channel 16 |

Remarks

Hardware Type and Channels

Non-Plug and Play: The hardware channels of this kind are used as registered. This means, for example, it can register the `PCANTP_HANDLE_ISABUS3` without having registered `PCANTP_HANDLE_ISA1` and `PCANTP_HANDLE_ISA2`. It is a decision of each user, how to associate a PCAN-channel (logical part) and a port/interrupt pair (physical part).

Plug and Play: For hardware handles of PCI, USB, and PC-Card, the availability of the channels is determined by the count of hardware connected to a computer at a specific moment, in conjunction with their internal handle. This means that having four PCAN-USB connected to a computer will let the user to connect the channels `PCANTP_HANDLE_USBBUS1` to `PCANTP_HANDLE_USBBUS4`. The association of each channel with hardware is managed internally using the handle of hardware.

See also: Detailed Parameters Characteristics on page 45.

3.6.4 cantp_hwtype

Represents the type of PCAN (non-Plug and Play) hardware to be initialized. According to the programming language, this type can be a group of defined values or an enumeration. This type is used in `Initialize_2013` method and `UDS_Initialize_2013` function.

Syntax

C/C++

```
#define PCAN_TYPE_ISA            0x01U
#define PCAN_TYPE_ISA_SJA       0x09U
#define PCAN_TYPE_ISA_PHYTEC    0x04U
#define PCAN_TYPE_DNG           0x02U
#define PCAN_TYPE_DNG_EPP       0x03U
#define PCAN_TYPE_DNG_SJA       0x05U
#define PCAN_TYPE_DNG_SJA_EPP   0x06U

typedef enum _cantp_hwtype {
    PCANTP_HWTYPE_ISA = PCAN_TYPE_ISA,
    PCANTP_HWTYPE_ISA_SJA = PCAN_TYPE_ISA_SJA,
    PCANTP_HWTYPE_ISA_PHYTEC = PCAN_TYPE_ISA_PHYTEC,
    PCANTP_HWTYPE_DNG = PCAN_TYPE_DNG,
    PCANTP_HWTYPE_DNG_EPP = PCAN_TYPE_DNG_EPP,
```

```

PCANTP_HWTYPE_DNG_SJA = PCAN_TYPE_DNG_SJA,
PCANTP_HWTYPE_DNG_SJA_EPP = PCAN_TYPE_DNG_SJA_EPP,
} cantp_hwtype;

```

Pascal OO

```

cantp_hwtype = (
  PCANTP_HWTYPE_ISA = UInt32(PCAN_TYPE_ISA),
  PCANTP_HWTYPE_ISA_SJA = UInt32(PCAN_TYPE_ISA_SJA),
  PCANTP_HWTYPE_ISA_PHYTEC = UInt32(PCAN_TYPE_ISA_PHYTEC),
  PCANTP_HWTYPE_DNG = UInt32(PCAN_TYPE_DNG),
  PCANTP_HWTYPE_DNG_EPP = UInt32(PCAN_TYPE_DNG_EPP),
  PCANTP_HWTYPE_DNG_SJA = UInt32(PCAN_TYPE_DNG_SJA),
  PCANTP_HWTYPE_DNG_SJA_EPP = UInt32(PCAN_TYPE_DNG_SJA_EPP)
);

```

C#

```

public enum cantp_hwtype : UInt32
{
  PCANTP_HWTYPE_ISA = TPCANType.PCAN_TYPE_ISA,
  PCANTP_HWTYPE_ISA_SJA = TPCANType.PCAN_TYPE_ISA_SJA,
  PCANTP_HWTYPE_ISA_PHYTEC = TPCANType.PCAN_TYPE_ISA_PHYTEC,
  PCANTP_HWTYPE_DNG = TPCANType.PCAN_TYPE_DNG,
  PCANTP_HWTYPE_DNG_EPP = TPCANType.PCAN_TYPE_DNG_EPP,
  PCANTP_HWTYPE_DNG_SJA = TPCANType.PCAN_TYPE_DNG_SJA,
  PCANTP_HWTYPE_DNG_SJA_EPP = TPCANType.PCAN_TYPE_DNG_SJA_EPP,
}

```

C++ / CLR

```

public enum cantp_hwtype : UInt32
{
  PCANTP_HWTYPE_ISA = (UInt32)TPCANType::PCAN_TYPE_ISA,
  PCANTP_HWTYPE_ISA_SJA = (UInt32)TPCANType::PCAN_TYPE_ISA_SJA,
  PCANTP_HWTYPE_ISA_PHYTEC = (UInt32)TPCANType::PCAN_TYPE_ISA_PHYTEC,
  PCANTP_HWTYPE_DNG = (UInt32)TPCANType::PCAN_TYPE_DNG,
  PCANTP_HWTYPE_DNG_EPP = (UInt32)TPCANType::PCAN_TYPE_DNG_EPP,
  PCANTP_HWTYPE_DNG_SJA = (UInt32)TPCANType::PCAN_TYPE_DNG_SJA,
  PCANTP_HWTYPE_DNG_SJA_EPP = (UInt32)TPCANType::PCAN_TYPE_DNG_SJA_EPP,
};

```

Visual Basic

```

Public Enum cantp_hwtype : UInt32
  PCANTP_HWTYPE_ISA = TPCANType.PCAN_TYPE_ISA
  PCANTP_HWTYPE_ISA_SJA = TPCANType.PCAN_TYPE_ISA_SJA
  PCANTP_HWTYPE_ISA_PHYTEC = TPCANType.PCAN_TYPE_ISA_PHYTEC
  PCANTP_HWTYPE_DNG = TPCANType.PCAN_TYPE_DNG
  PCANTP_HWTYPE_DNG_EPP = TPCANType.PCAN_TYPE_DNG_EPP
  PCANTP_HWTYPE_DNG_SJA = TPCANType.PCAN_TYPE_DNG_SJA
  PCANTP_HWTYPE_DNG_SJA_EPP = TPCANType.PCAN_TYPE_DNG_SJA_EPP
End Enum

```

Values

| Name | Value | Description |
|--------------------------|-------|------------------------|
| PCANTP_HWTYPE_ISA | 1 | PCAN-ISA 82C200 |
| PCANTP_HWTYPE_ISA_SJA | 9 | PCAN-ISA SJA1000 |
| PCANTP_HWTYPE_ISA_PHYTEC | 4 | PHYTEC ISA |
| PCANTP_HWTYPE_DNG | 2 | PCAN-Dongle 82C200 |
| PCANTP_HWTYPE_DNG_EPP | 3 | PCAN-Dongle EPP 82C200 |

| Name | Value | Description |
|---------------------------|-------|-------------------------|
| PCANTP_HWTYPE_DNG_SJA | 5 | PCAN-Dongle SJA1000 |
| PCANTP_HWTYPE_DNG_SJA_EPP | 6 | PCAN-Dongle EPP SJA1000 |

See also: [UDS_Initialize_2013](#) on page 624, **class method:** [Initialize_2013](#) on page 142.

3.6.5 cantp_isotp_addressing

Represents the type of target of a PCANTP or PUDS message.

Syntax

C/C++

```
typedef enum _cantp_isotp_addressing {
    PCANTP_ISOTP_ADDRESSING_UNKNOWN = 0x00,
    PCANTP_ISOTP_ADDRESSING_PHYSICAL = 0x01,
    PCANTP_ISOTP_ADDRESSING_FUNCTIONAL = 0x02,
} cantp_isotp_addressing;
```

Pascal OO

```
cantp_isotp_addressing = (
    PCANTP_ISOTP_ADDRESSING_UNKNOWN = $00,
    PCANTP_ISOTP_ADDRESSING_PHYSICAL = $01,
    PCANTP_ISOTP_ADDRESSING_FUNCTIONAL = $02
);
```

C#

```
public enum cantp_isotp_addressing : UInt32
{
    PCANTP_ISOTP_ADDRESSING_UNKNOWN = 0x00,
    PCANTP_ISOTP_ADDRESSING_PHYSICAL = 0x01,
    PCANTP_ISOTP_ADDRESSING_FUNCTIONAL = 0x02,
}
```

C++ / CLR

```
public enum cantp_isotp_addressing : UInt32
{
    PCANTP_ISOTP_ADDRESSING_UNKNOWN = 0x00,
    PCANTP_ISOTP_ADDRESSING_PHYSICAL = 0x01,
    PCANTP_ISOTP_ADDRESSING_FUNCTIONAL = 0x02,
};
```

Visual Basic

```
Public Enum cantp_isotp_addressing As UInt32
    PCANTP_ISOTP_ADDRESSING_UNKNOWN = &H0
    PCANTP_ISOTP_ADDRESSING_PHYSICAL = &H1
    PCANTP_ISOTP_ADDRESSING_FUNCTIONAL = &H2
End Enum
```

Values

| Name | Value | Description |
|------------------------------------|-------|--|
| PCANTP_ISOTP_ADDRESSING_UNKNOWN | 0x00 | Unknown addressing format. |
| PCANTP_ISOTP_ADDRESSING_PHYSICAL | 0x01 | Physical addressing ("peer to peer"). |
| PCANTP_ISOTP_ADDRESSING_FUNCTIONAL | 0x02 | Functional addressing ("peer to any"). |

3.6.6 cantp_baudrate

Represents a PCAN Baud rate register value for PCANTP channel or PUDS channel.

Syntax

C

```
#define PCAN_BAUD_1M          0x0014U
#define PCAN_BAUD_800K       0x0016U
#define PCAN_BAUD_500K       0x001CU
#define PCAN_BAUD_250K       0x011CU
#define PCAN_BAUD_125K       0x031CU
#define PCAN_BAUD_100K       0x432FU
#define PCAN_BAUD_95K        0xC34EU
#define PCAN_BAUD_83K        0x852BU
#define PCAN_BAUD_50K        0x472FU
#define PCAN_BAUD_47K        0x1414U
#define PCAN_BAUD_33K        0x8B2FU
#define PCAN_BAUD_20K        0x532FU
#define PCAN_BAUD_10K        0x672FU
#define PCAN_BAUD_5K         0x7F7FU

typedef enum _cantp_baudrate {
    PCANTP_BAUDRATE_1M = PCAN_BAUD_1M,
    PCANTP_BAUDRATE_800K = PCAN_BAUD_800K,
    PCANTP_BAUDRATE_500K = PCAN_BAUD_500K,
    PCANTP_BAUDRATE_250K = PCAN_BAUD_250K,
    PCANTP_BAUDRATE_125K = PCAN_BAUD_125K,
    PCANTP_BAUDRATE_100K = PCAN_BAUD_100K,
    PCANTP_BAUDRATE_95K = PCAN_BAUD_95K,
    PCANTP_BAUDRATE_83K = PCAN_BAUD_83K,
    PCANTP_BAUDRATE_50K = PCAN_BAUD_50K,
    PCANTP_BAUDRATE_47K = PCAN_BAUD_47K,
    PCANTP_BAUDRATE_33K = PCAN_BAUD_33K,
    PCANTP_BAUDRATE_20K = PCAN_BAUD_20K,
    PCANTP_BAUDRATE_10K = PCAN_BAUD_10K,
    PCANTP_BAUDRATE_5K = PCAN_BAUD_5K,
} cantp_baudrate;
```

Pascal OO

```
cantp_baudrate = (
    PCANTP_BAUDRATE_1M = UInt32(PCAN_BAUD_1M),
    PCANTP_BAUDRATE_800K = UInt32(PCAN_BAUD_800K),
    PCANTP_BAUDRATE_500K = UInt32(PCAN_BAUD_500K),
    PCANTP_BAUDRATE_250K = UInt32(PCAN_BAUD_250K),
    PCANTP_BAUDRATE_125K = UInt32(PCAN_BAUD_125K),
    PCANTP_BAUDRATE_100K = UInt32(PCAN_BAUD_100K),
    PCANTP_BAUDRATE_95K = UInt32(PCAN_BAUD_95K),
    PCANTP_BAUDRATE_83K = UInt32(PCAN_BAUD_83K),
    PCANTP_BAUDRATE_50K = UInt32(PCAN_BAUD_50K),
    PCANTP_BAUDRATE_47K = UInt32(PCAN_BAUD_47K),
    PCANTP_BAUDRATE_33K = UInt32(PCAN_BAUD_33K),
    PCANTP_BAUDRATE_20K = UInt32(PCAN_BAUD_20K),
    PCANTP_BAUDRATE_10K = UInt32(PCAN_BAUD_10K),
    PCANTP_BAUDRATE_5K = UInt32(PCAN_BAUD_5K)
);
```

C#

```
public enum cantp_baudrate : UInt32
{
    PCANTP_BAUDRATE_1M = TPCANBaudrate.PCAN_BAUD_1M,
    PCANTP_BAUDRATE_800K = TPCANBaudrate.PCAN_BAUD_800K,
    PCANTP_BAUDRATE_500K = TPCANBaudrate.PCAN_BAUD_500K,
    PCANTP_BAUDRATE_250K = TPCANBaudrate.PCAN_BAUD_250K,
    PCANTP_BAUDRATE_125K = TPCANBaudrate.PCAN_BAUD_125K,
    PCANTP_BAUDRATE_100K = TPCANBaudrate.PCAN_BAUD_100K,
    PCANTP_BAUDRATE_95K = TPCANBaudrate.PCAN_BAUD_95K,
    PCANTP_BAUDRATE_83K = TPCANBaudrate.PCAN_BAUD_83K,
    PCANTP_BAUDRATE_50K = TPCANBaudrate.PCAN_BAUD_50K,
    PCANTP_BAUDRATE_47K = TPCANBaudrate.PCAN_BAUD_47K,
    PCANTP_BAUDRATE_33K = TPCANBaudrate.PCAN_BAUD_33K,
    PCANTP_BAUDRATE_20K = TPCANBaudrate.PCAN_BAUD_20K,
    PCANTP_BAUDRATE_10K = TPCANBaudrate.PCAN_BAUD_10K,
    PCANTP_BAUDRATE_5K = TPCANBaudrate.PCAN_BAUD_5K,
}
```

C++ / CLR

```
public enum cantp_baudrate : UInt32
{
    PCANTP_BAUDRATE_1M = (UInt32)TPCANBaudrate::PCAN_BAUD_1M,
    PCANTP_BAUDRATE_800K = (UInt32)TPCANBaudrate::PCAN_BAUD_800K,
    PCANTP_BAUDRATE_500K = (UInt32)TPCANBaudrate::PCAN_BAUD_500K,
    PCANTP_BAUDRATE_250K = (UInt32)TPCANBaudrate::PCAN_BAUD_250K,
    PCANTP_BAUDRATE_125K = (UInt32)TPCANBaudrate::PCAN_BAUD_125K,
    PCANTP_BAUDRATE_100K = (UInt32)TPCANBaudrate::PCAN_BAUD_100K,
    PCANTP_BAUDRATE_95K = (UInt32)TPCANBaudrate::PCAN_BAUD_95K,
    PCANTP_BAUDRATE_83K = (UInt32)TPCANBaudrate::PCAN_BAUD_83K,
    PCANTP_BAUDRATE_50K = (UInt32)TPCANBaudrate::PCAN_BAUD_50K,
    PCANTP_BAUDRATE_47K = (UInt32)TPCANBaudrate::PCAN_BAUD_47K,
    PCANTP_BAUDRATE_33K = (UInt32)TPCANBaudrate::PCAN_BAUD_33K,
    PCANTP_BAUDRATE_20K = (UInt32)TPCANBaudrate::PCAN_BAUD_20K,
    PCANTP_BAUDRATE_10K = (UInt32)TPCANBaudrate::PCAN_BAUD_10K,
    PCANTP_BAUDRATE_5K = (UInt32)TPCANBaudrate::PCAN_BAUD_5K,
};
```

Visual Basic

```
Public Enum cantp_baudrate : UInt32
    PCANTP_BAUDRATE_1M = TPCANBaudrate.PCAN_BAUD_1M
    PCANTP_BAUDRATE_800K = TPCANBaudrate.PCAN_BAUD_800K
    PCANTP_BAUDRATE_500K = TPCANBaudrate.PCAN_BAUD_500K
    PCANTP_BAUDRATE_250K = TPCANBaudrate.PCAN_BAUD_250K
    PCANTP_BAUDRATE_125K = TPCANBaudrate.PCAN_BAUD_125K
    PCANTP_BAUDRATE_100K = TPCANBaudrate.PCAN_BAUD_100K
    PCANTP_BAUDRATE_95K = TPCANBaudrate.PCAN_BAUD_95K
    PCANTP_BAUDRATE_83K = TPCANBaudrate.PCAN_BAUD_83K
    PCANTP_BAUDRATE_50K = TPCANBaudrate.PCAN_BAUD_50K
    PCANTP_BAUDRATE_47K = TPCANBaudrate.PCAN_BAUD_47K
    PCANTP_BAUDRATE_33K = TPCANBaudrate.PCAN_BAUD_33K
    PCANTP_BAUDRATE_20K = TPCANBaudrate.PCAN_BAUD_20K
    PCANTP_BAUDRATE_10K = TPCANBaudrate.PCAN_BAUD_10K
    PCANTP_BAUDRATE_5K = TPCANBaudrate.PCAN_BAUD_5K
End Enum
```

values

| Name | Value | Description |
|----------------------|-------|---------------|
| PCANTP_BAUDRATE_1M | 20 | 1 Mbit/s |
| PCANTP_BAUDRATE_800K | 22 | 800 kBit/s |
| PCANTP_BAUDRATE_500K | 28 | 500 kBit/s |
| PCANTP_BAUDRATE_250K | 284 | 250 kBit/s |
| PCANTP_BAUDRATE_125K | 796 | 125 kBit/s |
| PCANTP_BAUDRATE_100K | 17199 | 100 kBit/s |
| PCANTP_BAUDRATE_95K | 49998 | 95,238 kBit/s |
| PCANTP_BAUDRATE_83K | 34091 | 83,333 kBit/s |
| PCANTP_BAUDRATE_50K | 18223 | 50 kBit/s |
| PCANTP_BAUDRATE_47K | 5140 | 47,619 kBit/s |
| PCANTP_BAUDRATE_33K | 35631 | 33,333 kBit/s |
| PCANTP_BAUDRATE_20K | 21295 | 20 kBit/s |
| PCANTP_BAUDRATE_10K | 26415 | 10 kBit/s |
| PCANTP_BAUDRATE_5K | 32639 | 5 kBit/s |

See also: `UDS_Initialize_2013` on page 624, **class method:** `Initialize_2013` on page 142.

3.6.7 cantp_msg

A `cantp_msg` message is a generic CAN related message that can be either a standard CAN message, a CAN FD message, or an ISO-TP message. The structure `uds_msg` includes a `cantp_msg` (see `uds_msg` on page 21).

Syntax

C/C++

```
typedef struct _cantp_msg {
    cantp_msgtype type;
    cantp_msginfo reserved;
    cantp_can_info can_info;
    union {
        cantp_msgdata* any;
        cantp_msgdata_can* can;
        cantp_msgdata_canfd* canfd;
        cantp_msgdata_isotp* isotp;
    } msgdata;
} cantp_msg;
```

Pascal OO

```
cantp_msg = record
    typem: cantp_msgtype;
    reserved: cantp_msginfo;
    can_info: cantp_can_info;
    case Integer of
        0:
            (msgdata_any: ^cantp_msgdata;);
        1:
            (msgdata_can: ^cantp_msgdata_can;);
        2:
            (msgdata_canfd: ^cantp_msgdata_canfd;);
        3:
            (msgdata_isotp: ^cantp_msgdata_isotp;);
    end;
```

C#

```
[StructLayout(LayoutKind.Sequential, Pack = 8)]
public struct cantp_msg
{
    [MarshalAs(UnmanagedType.U4)]
    public cantp_msgtype type;
    public cantp_msginfo reserved;
    public cantp_can_info can_info;
    private IntPtr msgdata;
    public IntPtr Msgdata
    {
        get { return msgdata; }
    }
    public cantp_msgdata Msgdata_any_Copy
    {
        get
        {
            return (cantp_msgdata)Marshal.PtrToStructure(msgdata, typeof(cantp_msgdata));
        }
    }
    public cantp_msgdata_can Msgdata_can_Copy
    {
        get
        {
            return (cantp_msgdata_can)Marshal.PtrToStructure(msgdata, typeof(cantp_msgdata_can));
        }
    }
    public cantp_msgdata_canfd Msgdata_canfd_Copy
    {
        get
        {
            return (cantp_msgdata_canfd)Marshal.PtrToStructure(msgdata, typeof(cantp_msgdata_canfd));
        }
    }
    public cantp_msgdata_isotp Msgdata_isotp_Copy
    {
        get
        {
            return (cantp_msgdata_isotp)Marshal.PtrToStructure(msgdata, typeof(cantp_msgdata_isotp));
        }
    }
}
```

C++/CLR

```
[StructLayout(LayoutKind::Explicit)]
public value struct cantp_msg_union_msgdata
{
    [FieldOffset(0)]
    cantp_msgdata *any;
    [FieldOffset(0)]
    cantp_msgdata_can *can;
    [FieldOffset(0)]
    cantp_msgdata_canfd *canfd;
    [FieldOffset(0)]
    cantp_msgdata_isotp *isotp;
};

[StructLayout(LayoutKind::Sequential, Pack = 8)]
public value struct cantp_msg
{

```

```
public:
    [MarshalAs(UnmanagedType::U4)]
    cantp_msgtype type;
    cantp_msginfo reserved;
    cantp_can_info can_info;
    cantp_msg_union_msgdata msgdata;
};
```

Visual Basic

```
<StructLayout(LayoutKind.Sequential, Pack:=8)>
Public Structure cantp_msg

    <MarshalAs(UnmanagedType.U4)>
    Public type As cantp_msgtype
    Public reserved As cantp_msginfo
    Public can_info As cantp_can_info
    Private _msgdata As IntPtr
    Public ReadOnly Property Msgdata() As IntPtr
        Get
            Return _msgdata
        End Get
    End Property

    Public ReadOnly Property Msgdata_any_Copy() As cantp_msgdata
        Get
            Return CType(Marshal.PtrToStructure(_msgdata, GetType(cantp_msgdata)), cantp_msgdata)
        End Get
    End Property

    Public ReadOnly Property Msgdata_can_Copy() As cantp_msgdata_can
        Get
            Return CType(Marshal.PtrToStructure(_msgdata, GetType(cantp_msgdata_can)),
                cantp_msgdata_can)
        End Get
    End Property

    Public ReadOnly Property Msgdata_canfd_Copy() As cantp_msgdata_canfd
        Get
            Return CType(Marshal.PtrToStructure(_msgdata, GetType(cantp_msgdata_canfd)),
                cantp_msgdata_canfd)
        End Get
    End Property

    Public ReadOnly Property Msgdata_isotp_Copy() As cantp_msgdata_isotp
        Get
            Return CType(Marshal.PtrToStructure(_msgdata, GetType(cantp_msgdata_isotp)),
                cantp_msgdata_isotp)
        End Get
    End Property
End Structure
```

Fields

| Name | Description |
|----------|--|
| type | Type of the message (see cantp_msgtype on page 123). |
| reserved | PCAN-ISO-TP 3.x API reserved miscellaneous read-only information. |
| can_info | Common CAN information (see cantp_can_info on page 124). |
| any | Shortcut to access message data as Generic content (see cantp_msgdata on page 125). |
| can | Shortcut to access message data as CAN content (see cantp_msgdata_can on page 126). |
| canfd | Shortcut to access message data as CAN FD content (see cantp_msgdata_canfd on page 128). |

| Name | Description |
|-------|---|
| isotp | Shortcut to access message data as ISO-TP content (see <code>cantp_msgdata_isotp</code> on page 129). |

See also: `uds_msg` on page 21.

3.6.8 `cantp_can_msgtype`

Represents the flags of a CAN or CAN FD message (must be used as flags for ex. `EXTENDED|FD|BRS`.) (see field `cantp_msg.can_info.can_msgtype` in `cantp_msg` on page 118).

Syntax

C/C++

```
#define PCAN_MESSAGE_STANDARD      0x00U
#define PCAN_MESSAGE_RTR          0x01U
#define PCAN_MESSAGE_EXTENDED     0x02U
#define PCAN_MESSAGE_FD           0x04U
#define PCAN_MESSAGE_BRS          0x08U
#define PCAN_MESSAGE_ESI          0x10U
#define PCAN_MESSAGE_ERRFRAME     0x40U
#define PCAN_MESSAGE_STATUS       0x80U
typedef enum _cantp_can_msgtype {
    PCANTP_CAN_MSGTYPE_STANDARD = PCAN_MESSAGE_STANDARD,
    PCANTP_CAN_MSGTYPE_RTR      = PCAN_MESSAGE_RTR,
    PCANTP_CAN_MSGTYPE_EXTENDED = PCAN_MESSAGE_EXTENDED,
    PCANTP_CAN_MSGTYPE_FD       = PCAN_MESSAGE_FD,
    PCANTP_CAN_MSGTYPE_BRS      = PCAN_MESSAGE_BRS,
    PCANTP_CAN_MSGTYPE_ESI      = PCAN_MESSAGE_ESI,
    PCANTP_CAN_MSGTYPE_ERRFRAME = PCAN_MESSAGE_ERRFRAME,
    PCANTP_CAN_MSGTYPE_STATUS   = PCAN_MESSAGE_STATUS,
    PCANTP_CAN_MSGTYPE_SELFRECEIVE = 0xC0U,
    PCANTP_CAN_MSGTYPE_FLAG_INFO = (PCAN_MESSAGE_ERRFRAME | PCAN_MESSAGE_STATUS)
} cantp_can_msgtype;
```

Pascal OO

```
cantp_can_msgtype = (
    PCANTP_CAN_MSGTYPE_STANDARD = UInt32(PCAN_MESSAGE_STANDARD),
    PCANTP_CAN_MSGTYPE_RTR      = UInt32(PCAN_MESSAGE_RTR),
    PCANTP_CAN_MSGTYPE_EXTENDED = UInt32(PCAN_MESSAGE_EXTENDED),
    PCANTP_CAN_MSGTYPE_FD       = UInt32(PCAN_MESSAGE_FD),
    PCANTP_CAN_MSGTYPE_BRS      = UInt32(PCAN_MESSAGE_BRS),
    PCANTP_CAN_MSGTYPE_ESI      = UInt32(PCAN_MESSAGE_ESI),
    PCANTP_CAN_MSGTYPE_ERRFRAME = UInt32(PCAN_MESSAGE_ERRFRAME),
    PCANTP_CAN_MSGTYPE_STATUS   = UInt32(PCAN_MESSAGE_STATUS),
    PCANTP_CAN_MSGTYPE_SELFRECEIVE = UInt32($C0),
    PCANTP_CAN_MSGTYPE_FLAG_INFO = (UInt32(PCAN_MESSAGE_ERRFRAME)
    Or UInt32(PCAN_MESSAGE_STATUS))
);
```

C#

```
public enum cantp_can_msgtype : UInt32
{
    PCANTP_CAN_MSGTYPE_STANDARD = TPCANMessageType.PCAN_MESSAGE_STANDARD,
    PCANTP_CAN_MSGTYPE_RTR      = TPCANMessageType.PCAN_MESSAGE_RTR,
    PCANTP_CAN_MSGTYPE_EXTENDED = TPCANMessageType.PCAN_MESSAGE_EXTENDED,
    PCANTP_CAN_MSGTYPE_FD       = TPCANMessageType.PCAN_MESSAGE_FD,
    PCANTP_CAN_MSGTYPE_BRS      = TPCANMessageType.PCAN_MESSAGE_BRS,
    PCANTP_CAN_MSGTYPE_ESI      = TPCANMessageType.PCAN_MESSAGE_ESI,
```

```

PCANTP_CAN_MSGTYPE_ERRFRAME = TPCANMessageType.PCAN_MESSAGE_ERRFRAME,
PCANTP_CAN_MSGTYPE_STATUS = TPCANMessageType.PCAN_MESSAGE_STATUS,
PCANTP_CAN_MSGTYPE_SELFRECEIVE = 0xC0,
PCANTP_CAN_MSGTYPE_FLAG_INFO = (TPCANMessageType.PCAN_MESSAGE_ERRFRAME
    | TPCANMessageType.PCAN_MESSAGE_STATUS)
}

```

C++ / CLR

```

public enum cantp_can_msgtype : UInt32
{
    PCANTP_CAN_MSGTYPE_STANDARD = (UInt32)TPCANMessageType::PCAN_MESSAGE_STANDARD,
    PCANTP_CAN_MSGTYPE_RTR = (UInt32)TPCANMessageType::PCAN_MESSAGE_RTR,
    PCANTP_CAN_MSGTYPE_EXTENDED = (UInt32)TPCANMessageType::PCAN_MESSAGE_EXTENDED,
    PCANTP_CAN_MSGTYPE_FD = (UInt32)TPCANMessageType::PCAN_MESSAGE_FD,
    PCANTP_CAN_MSGTYPE_BRS = (UInt32)TPCANMessageType::PCAN_MESSAGE_BRS,
    PCANTP_CAN_MSGTYPE_ESI = (UInt32)TPCANMessageType::PCAN_MESSAGE_ESI,
    PCANTP_CAN_MSGTYPE_ERRFRAME = (UInt32)TPCANMessageType::PCAN_MESSAGE_ERRFRAME,
    PCANTP_CAN_MSGTYPE_STATUS = (UInt32)TPCANMessageType::PCAN_MESSAGE_STATUS,
    PCANTP_CAN_MSGTYPE_SELFRECEIVE = 0xC0,
    PCANTP_CAN_MSGTYPE_FLAG_INFO = ((UInt32)TPCANMessageType::PCAN_MESSAGE_ERRFRAME
        | (UInt32)TPCANMessageType::PCAN_MESSAGE_STATUS)
};

```

Visual Basic

```

Public Enum cantp_can_msgtype As UInt32
    PCANTP_CAN_MSGTYPE_STANDARD = TPCANMessageType.PCAN_MESSAGE_STANDARD
    PCANTP_CAN_MSGTYPE_RTR = TPCANMessageType.PCAN_MESSAGE_RTR
    PCANTP_CAN_MSGTYPE_EXTENDED = TPCANMessageType.PCAN_MESSAGE_EXTENDED
    PCANTP_CAN_MSGTYPE_FD = TPCANMessageType.PCAN_MESSAGE_FD
    PCANTP_CAN_MSGTYPE_BRS = TPCANMessageType.PCAN_MESSAGE_BRS
    PCANTP_CAN_MSGTYPE_ESI = TPCANMessageType.PCAN_MESSAGE_ESI
    PCANTP_CAN_MSGTYPE_ERRFRAME = TPCANMessageType.PCAN_MESSAGE_ERRFRAME
    PCANTP_CAN_MSGTYPE_STATUS = TPCANMessageType.PCAN_MESSAGE_STATUS
    PCANTP_CAN_MSGTYPE_SELFRECEIVE = &HC0
    PCANTP_CAN_MSGTYPE_FLAG_INFO = (TPCANMessageType.PCAN_MESSAGE_ERRFRAME
        Or TPCANMessageType.PCAN_MESSAGE_STATUS)
End Enum

```

Values

| Name | Value | Description |
|--------------------------------|-------|---|
| PCANTP_CAN_MSGTYPE_STANDARD | 0x00 | The PCAN message is a CAN Standard Frame (11-bit identifier). |
| PCANTP_CAN_MSGTYPE_RTR | 0x01 | The PCAN message is a CAN Remote-Transfer-Request Frame. |
| PCANTP_CAN_MSGTYPE_EXTENDED | 0x02 | The PCAN message is a CAN Extended Frame (29-bit identifier). |
| PCANTP_CAN_MSGTYPE_FD | 0x04 | The PCAN message represents a FD frame in terms of CiA Specs. |
| PCANTP_CAN_MSGTYPE_BRS | 0x08 | The PCAN message represents a FD bit rate switch (CAN data at a higher bit rate). |
| PCANTP_CAN_MSGTYPE_ESI | 0x10 | The PCAN message represents a FD error state indicator (CAN FD transmitter was error active). |
| PCANTP_CAN_MSGTYPE_ERRFRAME | 0x40 | The PCAN message represents an error frame. |
| PCANTP_CAN_MSGTYPE_STATUS | 0x80 | The PCAN message represents a PCAN status message. |
| PCANTP_CAN_MSGTYPE_SELFRECEIVE | 0xC0 | The PCAN message represents a self-received (Tx-loopback) message. |
| PCANTP_CAN_MSGTYPE_FLAG_INFO | 0xC0 | Mask filtering error frame messages and status messages. |

See also: [cantp_msg](#) on page 118, [cantp_can_info](#) on page 124.

3.6.9 cantp_msgtype

Represents the type of a PCANTP message (see field `type` in `cantp_msg` on page 118).

Syntax

C/C++

```
typedef enum _cantp_msgtype {
    PCANTP_MSGTYPE_NONE = 0,
    PCANTP_MSGTYPE_CAN = 1,
    PCANTP_MSGTYPE_CANFD = 2,
    PCANTP_MSGTYPE_ISOTP = 4,
    PCANTP_MSGTYPE_FRAME = PCANTP_MSGTYPE_CAN | PCANTP_MSGTYPE_CANFD,
    PCANTP_MSGTYPE_ANY = PCANTP_MSGTYPE_FRAME | PCANTP_MSGTYPE_ISOTP | 0xFFFFFFFF
} cantp_msgtype;
```

Pascal OO

```
cantp_msgtype = (
    PCANTP_MSGTYPE_NONE = 0,
    PCANTP_MSGTYPE_CAN = 1,
    PCANTP_MSGTYPE_CANFD = 2,
    PCANTP_MSGTYPE_ISOTP = 4,
    PCANTP_MSGTYPE_FRAME = UInt32(PCANTP_MSGTYPE_CAN) Or UInt32(PCANTP_MSGTYPE_CANFD),
    PCANTP_MSGTYPE_ANY = UInt32(PCANTP_MSGTYPE_FRAME) Or UInt32(PCANTP_MSGTYPE_ISOTP)
    Or UInt32($FFFFFFFF)
);
```

C#

```
[Flags]
public enum cantp_msgtype : UInt32
{
    PCANTP_MSGTYPE_NONE = 0,
    PCANTP_MSGTYPE_CAN = 1,
    PCANTP_MSGTYPE_CANFD = 2,
    PCANTP_MSGTYPE_ISOTP = 4,
    PCANTP_MSGTYPE_FRAME = PCANTP_MSGTYPE_CAN | PCANTP_MSGTYPE_CANFD,
    PCANTP_MSGTYPE_ANY = PCANTP_MSGTYPE_FRAME | PCANTP_MSGTYPE_ISOTP | 0xFFFFFFFF
}
```

C++ / CLR

```
public enum cantp_msgtype : UInt32
{
    PCANTP_MSGTYPE_NONE = 0,
    PCANTP_MSGTYPE_CAN = 1,
    PCANTP_MSGTYPE_CANFD = 2,
    PCANTP_MSGTYPE_ISOTP = 4,
    PCANTP_MSGTYPE_FRAME = PCANTP_MSGTYPE_CAN | PCANTP_MSGTYPE_CANFD,
    PCANTP_MSGTYPE_ANY = PCANTP_MSGTYPE_FRAME | PCANTP_MSGTYPE_ISOTP | 0xFFFFFFFF
};
```

Visual Basic

```
<Flags()>
Public Enum cantp_msgtype As UInt32
    PCANTP_MSGTYPE_NONE = 0
    PCANTP_MSGTYPE_CAN = 1
    PCANTP_MSGTYPE_CANFD = 2
    PCANTP_MSGTYPE_ISOTP = 4
```

```
PCANTP_MSGTYPE_FRAME = PCANTP_MSGTYPE_CAN Or PCANTP_MSGTYPE_CANFD
PCANTP_MSGTYPE_ANY = PCANTP_MSGTYPE_FRAME Or PCANTP_MSGTYPE_ISOTP Or &HFFFFFFFUI
```

End Enum

Values

| Name | Value | Description |
|----------------------|------------|---------------------------------------|
| PCANTP_MSGTYPE_NONE | 0x00000000 | Uninitialized message (data is NULL). |
| PCANTP_MSGTYPE_CAN | 0x00000001 | Standard CAN message. |
| PCANTP_MSGTYPE_CANFD | 0x00000002 | CAN message with FD support. |
| PCANTP_MSGTYPE_ISOTP | 0x00000004 | PCAN-ISO-TP message (ISO:15765). |
| PCANTP_MSGTYPE_FRAME | 0x00000003 | Frame only: unsegmented messages. |
| PCANTP_MSGTYPE_ANY | 0xFFFFFFFF | Any supported message types. |

See also: `cantp_msg` on page 118.

3.6.10 cantp_can_info

Represents common CAN information.

Syntax

C/C++

```
typedef struct _cantp_can_info {
    uint32_t can_id;
    cantp_can_msgtype can_msgtype;
    uint8_t dlc;
} cantp_can_info;
```

Pascal OO

```
cantp_can_info = record
    can_id: UInt32;
    can_msgtype: cantp_can_msgtype;
    dlc: Byte;
end;
```

C#

```
[StructLayout(LayoutKind.Sequential, Pack = 8)]
public struct cantp_can_info
{
    public UInt32 can_id;
    [MarshalAs(UnmanagedType.U4)]
    public cantp_can_msgtype can_msgtype;
    public byte dlc;
}
```

C++/CLR

```
[StructLayout(LayoutKind::Sequential, Pack = 8)]
public value struct cantp_can_info
{
    UInt32 can_id;
    [MarshalAs(UnmanagedType::U4)]
    cantp_can_msgtype can_msgtype;
    Byte dlc;
};
```

Visual Basic

```
<StructLayout(LayoutKind.Sequential, Pack:=8)>
Public Structure cantp_can_info
    Public can_id As UInt32
    <MarshalAs(UnmanagedType.U4)>
    Public can_msgtype As cantp_can_msgtype
    Public dlc As Byte
End Structure
```

Fields

| Name | Description |
|-------------|---|
| can_id | CAN identifier. |
| can_msgtype | Type and flags of the CAN/CAN FD frame (see cantp_can_msgtype on page 121). |
| dlc | Data Length Code of the frame. |

Remarks

Specifying a non-zero `dlc` value when writing an ISO-TP message will override the value `can_tx_dlc` of its corresponding mapping (if it exists) and the value of parameter `PUDS_PARAMETER_CAN_TX_DL`.

See also: `cantp_msg` on page 118, `uds_mapping` on page 25.

3.6.11 cantp_msgdata

Represents the content of a generic PCANTP message.

Syntax

C/C++

```
typedef struct _cantp_msgdata {
    cantp_msgflag flags;
    uint32_t length;
    uint8_t* data;
    cantp_netstatus netstatus;
    cantp_msgoption_list* options;
} cantp_msgdata;
```

Pascal OO

```
cantp_msgdata = record
    flags: cantp_msgflag;
    length: UInt32;
    data: ^Byte;
    netstatus: cantp_netstatus;
    options: ^cantp_msgoption_list;
end;
```

C#

```
[StructLayout(LayoutKind.Sequential, Pack = 8)]
public struct cantp_msgdata
{
    [MarshalAs(UnmanagedType.U4)]
    public cantp_msgflag flags;
    public UInt32 length;
    public IntPtr data;
    [MarshalAs(UnmanagedType.U4)]
```

```

    public cantp_netstatus netstatus;
    public IntPtr options;
}

```

C++/CLR

```

[StructLayout(LayoutKind::Sequential, Pack = 8)]
public value struct cantp_msgdata
{
    [MarshalAs(UnmanagedType::U4)]
    cantp_msgflag flags;
    UInt32 length;
    Byte *data;
    [MarshalAs(UnmanagedType::U4)]
    cantp_netstatus netstatus;
    cantp_msgoption_list *options;
};

```

Visual Basic

```

<StructLayout(LayoutKind.Sequential, Pack:=8)>
Public Structure cantp_msgdata
    <MarshalAs(UnmanagedType.U4)>
    Public flags As cantp_msgflag
    Public length As UInt32
    Public data As IntPtr
    <MarshalAs(UnmanagedType.U4)>
    Public netstatus As cantp_netstatus
    Public options As IntPtr
End Structure

```

Fields

| Name | Description |
|-----------|---|
| flags | Structure specific flags (see cantp_msgflag on page 131). |
| Length | Length of the message. |
| Data | Data of the message. |
| Netstatus | Network status (see cantp_netstatus on page 132). |
| Options | Defines specific options to override global message configuration (not used with UDS_Scv*). |

See also: `cantp_msg` on page 118.

3.6.12 cantp_msgdata_can

Represents the content of a standard CAN message.

Syntax

C/C++

```

typedef struct _cantp_msgdata_can {
    cantp_msgflag flags;
    uint32_t length;
    uint8_t* data;
    cantp_netstatus netstatus;
    cantp_msgoption_list* options;
    uint8_t data_max[PCANTP_MAX_LENGTH_CAN_STANDARD];
} cantp_msgdata_can;

```

Pascal OO

```
cantp_msgdata_can = record
  flags: cantp_msgflag;
  length: UInt32;
  data: ^Byte;
  netstatus: cantp_netstatus;
  options: ^cantp_msgoption_list;
  data_max: array [0 .. PCANTP_MAX_LENGTH_CAN_STANDARD - 1] of Byte;
end;
```

C#

```
[StructLayout(LayoutKind.Sequential, Pack = 8)]
public struct cantp_msgdata_can
{
    [MarshalAs(UnmanagedType.U4)]
    public cantp_msgflag flags;
    public UInt32 length;
    public IntPtr data;
    [MarshalAs(UnmanagedType.U4)]
    public cantp_netstatus netstatus;
    public IntPtr options;
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = CanTpApi.PCANTP_MAX_LENGTH_CAN_STANDARD)]
    public byte[] data_max;
}
```

C++/CLR

```
public struct cantp_msgdata_can
{
    [MarshalAs(UnmanagedType::U4)]
    cantp_msgflag flags;
    UInt32 length;
    Byte *data;
    [MarshalAs(UnmanagedType::U4)]
    cantp_netstatus netstatus;
    cantp_msgoption_list *options;
    [MarshalAs(UnmanagedType::ByValArray, SizeConst = 8)]
    Byte data_max[];
};
```

Visual Basic

```
<StructLayout(LayoutKind.Sequential, Pack:=8)>
Public Structure cantp_msgdata_can
    <MarshalAs(UnmanagedType.U4)>
    Public flags As cantp_msgflag
    Public length As UInt32
    Public data As IntPtr
    <MarshalAs(UnmanagedType.U4)>
    Public netstatus As cantp_netstatus
    Public options As IntPtr
    <MarshalAs(UnmanagedType.ByValArray, SizeConst:=CanTpApi.PCANTP_MAX_LENGTH_CAN_STANDARD)>
    Public data_max As Byte()
End Structure
```

Fields

| Name | Description |
|-----------|---|
| flags | Structure specific flags (see <code>cantp_msgflag</code> on page 131). |
| length | Length of the message (0..8). |
| data | Data of the message. |
| netstatus | Network status. (see <code>cantp_netstatus</code> on page 132). |
| options | Defines specific options to override global CAN configuration (not used yet). |
| data_max | Data of the message (<code>data[0]..data[7]</code>). |

See also: `cantp_msg` on page 118, `cantp_msgdata` on page 125.

3.6.13 `cantp_msgdata_canfd`

Represents the content of a CAN FD message.

Syntax

C/C++

```
typedef struct _cantp_msgdata_canfd {
    cantp_msgflag flags;
    uint32_t length;
    uint8_t* data;
    cantp_netstatus netstatus;
    cantp_msgoption_list* options;
    uint8_t data_max[PCANTP_MAX_LENGTH_CAN_FD];
} cantp_msgdata_canfd;
```

Pascal OO

```
cantp_msgdata_canfd = record
    flags: cantp_msgflag;
    length: UInt32;
    data: ^Byte;
    netstatus: cantp_netstatus;
    options: ^cantp_msgoption_list;
    data_max: array [0 .. PCANTP_MAX_LENGTH_CAN_FD - 1] of Byte;
end;
```

C#

```
[StructLayout(LayoutKind.Sequential, Pack = 8)]
public struct cantp_msgdata_canfd
{
    [MarshalAs(UnmanagedType.U4)]
    public cantp_msgflag flags;
    public UInt32 length;
    public IntPtr data;
    [MarshalAs(UnmanagedType.U4)]
    public cantp_netstatus netstatus;
    public IntPtr options;
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = CanTpApi.PCANTP_MAX_LENGTH_CAN_FD)]
    public byte[] data_max;
}
```


C++/CLR

```
public struct cantp_msgdata_canfd
{
    [MarshalAs(UnmanagedType::U4)]
    cantp_msgflag flags;
    UInt32 length;
    Byte *data;
    [MarshalAs(UnmanagedType::U4)]
    cantp_netstatus netstatus;
    cantp_msgoption_list *options;
    [MarshalAs(UnmanagedType::ByValArray, SizeConst = 64)]
    Byte data_max[];
};
```

Visual Basic

```
<StructLayout(LayoutKind.Sequential, Pack:=8)>
Public Structure cantp_msgdata_canfd
    <MarshalAs(UnmanagedType.U4)>
    Public flags As cantp_msgflag
    Public length As UInt32
    Public data As IntPtr
    <MarshalAs(UnmanagedType.U4)>
    Public netstatus As cantp_netstatus
    Public options As IntPtr
    <MarshalAs(UnmanagedType.ByValArray, SizeConst:=CanTpApi.PCANTP_MAX_LENGTH_CAN_FD)>
    Public data_max As Byte()
End Structure
```

Fields

| Name | Description |
|-----------|---|
| flags | Structure specific flags (see cantp_msgflag on page 131). |
| length | Length of the message (0..64). |
| data | Data of the message. |
| netstatus | Network status (see cantp_netstatus on page 132). |
| options | Defines specific options to override global CAN configuration (not used yet). |
| data_max | Data of the message (data[0]..data[63]). |

See also: `cantp_msg` on page 118, `cantp_msgdata` on page 125.

3.6.14 cantp_msgdata_isotp

Represents the content of an ISO-TP message.

Syntax**C/C++**

```
typedef struct _cantp_msgdata_isotp {
    cantp_msgflag flags;
    uint32_t length;
    uint8_t* data;
    cantp_netstatus netstatus;
    cantp_msgoption_list* options;
    cantp_netaddrinfo netaddrinfo;
    cantp_isotp_info reserved;
} cantp_msgdata_isotp;
```

Pascal OO

```
cantp_msgdata_isotp = record
  flags: cantp_msgflag;
  length: UInt32;
  data: ^Byte;
  netstatus: cantp_netstatus;
  options: ^cantp_msgoption_list;
  netaddrinfo: cantp_netaddrinfo;
  reserved: cantp_isotp_info;
end;
```

C#

```
[StructLayout(LayoutKind.Sequential, Pack = 8)]
public struct cantp_msgdata_isotp
{
    [MarshalAs(UnmanagedType.U4)]
    public cantp_msgflag flags;
    public UInt32 length;
    public IntPtr data;
    [MarshalAs(UnmanagedType.U4)]
    public cantp_netstatus netstatus;
    public IntPtr options;
    public cantp_netaddrinfo netaddrinfo;
    public cantp_isotp_info reserved;
}
```

C++/CLR

```
[StructLayout(LayoutKind::Sequential, Pack = 8)]
public value struct cantp_msgdata_isotp
{
    [MarshalAs(UnmanagedType::U4)]
    cantp_msgflag flags;
    UInt32 length;
    Byte *data;
    [MarshalAs(UnmanagedType::U4)]
    cantp_netstatus netstatus;
    cantp_msgoption_list *options;
    cantp_netaddrinfo netaddrinfo;
    cantp_isotp_info reserved;
};
```

Visual Basic

```
<StructLayout(LayoutKind.Sequential, Pack:=8)>
Public Structure cantp_msgdata_isotp
    <MarshalAs(UnmanagedType.U4)>
    Public flags As cantp_msgflag
    Public length As UInt32
    Public data As IntPtr
    <MarshalAs(UnmanagedType.U4)>
    Public netstatus As cantp_netstatus
    Public options As IntPtr
    Public netaddrinfo As cantp_netaddrinfo
    Public reserved As cantp_isotp_info
End Structure
```

Fields

| Name | Description |
|-------------|---|
| flags | Structure specific flags (see <code>cantp_msgflag</code> on page 131). |
| length | Length of the data. |
| data | Data of the message. |
| netstatus | Network status (see <code>cantp_netstatus</code> on page 132). |
| options | Defines specific options to override global CAN configuration (not used in UDS_Scv* functions). |
| netaddrinfo | PCAN-ISO-TP 3.x network address information (see <code>cantp_netaddrinfo</code> on page 134). |
| reserved | Reserved PCAN-ISO-TP 3.x information. |

See also: `cantp_msg` on page 118, `cantp_msgdata` on page 125.

3.6.15 `cantp_msgflag`

Represents the flags common to all types of `cantp_msg` (see field `cantp_msg.msgdata.flags` on page 118).

Syntax

C/C++

```
typedef enum _cantp_msgflag {
    PCANTP_MSGFLAG_NONE = 0,
    PCANTP_MSGFLAG_LOOPBACK = 1,
    PCANTP_MSGFLAG_ISOTP_FRAME = 2,
} cantp_msgflag;
```

Pascal OO

```
cantp_msgflag = (
    PCANTP_MSGFLAG_NONE = 0,
    PCANTP_MSGFLAG_LOOPBACK = 1,
    PCANTP_MSGFLAG_ISOTP_FRAME = 2
);
```

C#

```
public enum cantp_msgflag : UInt32
{
    PCANTP_MSGFLAG_NONE = 0,
    PCANTP_MSGFLAG_LOOPBACK = 1,
    PCANTP_MSGFLAG_ISOTP_FRAME = 2,
}
```

C++ / CLR

```
public enum cantp_msgflag : UInt32
{
    PCANTP_MSGFLAG_NONE = 0,
    PCANTP_MSGFLAG_LOOPBACK = 1,
    PCANTP_MSGFLAG_ISOTP_FRAME = 2,
};
```

Visual Basic

```
Public Enum cantp_msgflag As UInt32
    PCANTP_MSGFLAG_NONE = 0
    PCANTP_MSGFLAG_LOOPBACK = 1
    PCANTP_MSGFLAG_ISOTP_FRAME = 2
End Enum
```

Values

| Name | Value | Description |
|----------------------------|-------|---|
| PCANTP_MSGFLAG_NONE | 0 | No flag. |
| PCANTP_MSGFLAG_LOOPBACK | 1 | Message is the confirmation of a transmitted message. |
| PCANTP_MSGFLAG_ISOTP_FRAME | 2 | Message is a frame of a segmented ISO-TP message. |

See also: `cantp_msg` on page 118, `cantp_msgdata` on page 125.

3.6.16 cantp_netstatus

Represents the network result of the communication of a PCANTP message.

Syntax

C/C++

```
typedef enum _cantp_netstatus {
    PCANTP_NETSTATUS_OK = 0x00,
    PCANTP_NETSTATUS_TIMEOUT_A = 0x01,
    PCANTP_NETSTATUS_TIMEOUT_Bs = 0x02,
    PCANTP_NETSTATUS_TIMEOUT_Cr = 0x03,
    PCANTP_NETSTATUS_WRONG_SN = 0x04,
    PCANTP_NETSTATUS_INVALID_FS = 0x05,
    PCANTP_NETSTATUS_UNEXP_PDU = 0x06,
    PCANTP_NETSTATUS_WFT_OVRN = 0x07,
    PCANTP_NETSTATUS_BUFFER_OVFLW = 0x08,
    PCANTP_NETSTATUS_ERROR = 0x09,
    PCANTP_NETSTATUS_IGNORED = 0x0A,
    PCANTP_NETSTATUS_TIMEOUT_As = 0x10,
    PCANTP_NETSTATUS_TIMEOUT_Ar = 0x11,
    PCANTP_NETSTATUS_XMT_FULL = 0x11,
    PCANTP_NETSTATUS_BUS_ERROR = 0x12,
    PCANTP_NETSTATUS_NO_MEMORY = 0x13,
} cantp_netstatus;
```

Pascal OO

```
cantp_netstatus = (
    PCANTP_NETSTATUS_OK = $00,
    PCANTP_NETSTATUS_TIMEOUT_A = $01,
    PCANTP_NETSTATUS_TIMEOUT_Bs = $02,
    PCANTP_NETSTATUS_TIMEOUT_Cr = $03,
    PCANTP_NETSTATUS_WRONG_SN = $04,
    PCANTP_NETSTATUS_INVALID_FS = $05,
    PCANTP_NETSTATUS_UNEXP_PDU = $06,
    PCANTP_NETSTATUS_WFT_OVRN = $07,
    PCANTP_NETSTATUS_BUFFER_OVFLW = $08,
    PCANTP_NETSTATUS_ERROR = $09,
    PCANTP_NETSTATUS_IGNORED = $0A,
    PCANTP_NETSTATUS_TIMEOUT_As = $10,
    PCANTP_NETSTATUS_TIMEOUT_Ar = $11,
    PCANTP_NETSTATUS_XMT_FULL = $11,
```

```

PCANTP_NETSTATUS_BUS_ERROR = $12,
PCANTP_NETSTATUS_NO_MEMORY = $13
);

```

C#

```

public enum cantp_netstatus : UInt32
{
    PCANTP_NETSTATUS_OK = 0x00,
    PCANTP_NETSTATUS_TIMEOUT_A = 0x01,
    PCANTP_NETSTATUS_TIMEOUT_Bs = 0x02,
    PCANTP_NETSTATUS_TIMEOUT_Cr = 0x03,
    PCANTP_NETSTATUS_WRONG_SN = 0x04,
    PCANTP_NETSTATUS_INVALID_FS = 0x05,
    PCANTP_NETSTATUS_UNEXP_PDU = 0x06,
    PCANTP_NETSTATUS_WFT_OVRN = 0x07,
    PCANTP_NETSTATUS_BUFFER_OVFLW = 0x08,
    PCANTP_NETSTATUS_ERROR = 0x09,
    PCANTP_NETSTATUS_IGNORED = 0x0A,
    PCANTP_NETSTATUS_TIMEOUT_As = 0x10,
    PCANTP_NETSTATUS_TIMEOUT_Ar = 0x11,
    PCANTP_NETSTATUS_XMT_FULL = 0x11,
    PCANTP_NETSTATUS_BUS_ERROR = 0x12,
    PCANTP_NETSTATUS_NO_MEMORY = 0x13,
}

```

C++ / CLR

```

public enum cantp_netstatus : UInt32
{
    PCANTP_NETSTATUS_OK = 0x00,
    PCANTP_NETSTATUS_TIMEOUT_A = 0x01,
    PCANTP_NETSTATUS_TIMEOUT_Bs = 0x02,
    PCANTP_NETSTATUS_TIMEOUT_Cr = 0x03,
    PCANTP_NETSTATUS_WRONG_SN = 0x04,
    PCANTP_NETSTATUS_INVALID_FS = 0x05,
    PCANTP_NETSTATUS_UNEXP_PDU = 0x06,
    PCANTP_NETSTATUS_WFT_OVRN = 0x07,
    PCANTP_NETSTATUS_BUFFER_OVFLW = 0x08,
    PCANTP_NETSTATUS_ERROR = 0x09,
    PCANTP_NETSTATUS_IGNORED = 0x0A,
    PCANTP_NETSTATUS_TIMEOUT_As = 0x10,
    PCANTP_NETSTATUS_TIMEOUT_Ar = 0x11,
    PCANTP_NETSTATUS_XMT_FULL = 0x11,
    PCANTP_NETSTATUS_BUS_ERROR = 0x12,
    PCANTP_NETSTATUS_NO_MEMORY = 0x13,
};

```

Visual Basic

```

Public Enum cantp_netstatus As UInt32
    PCANTP_NETSTATUS_OK = &H0
    PCANTP_NETSTATUS_TIMEOUT_A = &H1
    PCANTP_NETSTATUS_TIMEOUT_Bs = &H2
    PCANTP_NETSTATUS_TIMEOUT_Cr = &H3
    PCANTP_NETSTATUS_WRONG_SN = &H4
    PCANTP_NETSTATUS_INVALID_FS = &H5
    PCANTP_NETSTATUS_UNEXP_PDU = &H6
    PCANTP_NETSTATUS_WFT_OVRN = &H7
    PCANTP_NETSTATUS_BUFFER_OVFLW = &H8
    PCANTP_NETSTATUS_ERROR = &H9
    PCANTP_NETSTATUS_IGNORED = &HA
    PCANTP_NETSTATUS_TIMEOUT_As = &H10
    PCANTP_NETSTATUS_TIMEOUT_Ar = &H11

```

```
PCANTP_NETSTATUS_XMT_FULL = &H11
PCANTP_NETSTATUS_BUS_ERROR = &H12
PCANTP_NETSTATUS_NO_MEMORY = &H13
```

End Enum

Values

| Name | Value | Description |
|-------------------------------|-------|--|
| PCANTP_NETSTATUS_OK | 0x00 | No network errors. |
| PCANTP_NETSTATUS_TIMEOUT_A | 0x01 | Timeout occurred between 2 frames transmission (sender and receiver side). |
| PCANTP_NETSTATUS_TIMEOUT_Bs | 0x02 | Sender side timeout while waiting for flow control frame. |
| PCANTP_NETSTATUS_TIMEOUT_Cr | 0x03 | Receiver side timeout while waiting for consecutive frame. |
| PCANTP_NETSTATUS_WRONG_SN | 0x04 | Unexpected sequence number. |
| PCANTP_NETSTATUS_INVALID_FS | 0x05 | Invalid or unknown FlowStatus. |
| PCANTP_NETSTATUS_UNEXP_PDU | 0x06 | Unexpected protocol data unit. |
| PCANTP_NETSTATUS_WFT_OVRN | 0x07 | Reception of flow control WAIT frame that exceeds the maximum counter defined by PCANTP_PARAMETER_WFT_MAX. |
| PCANTP_NETSTATUS_BUFFER_OVFLW | 0x08 | Buffer on the receiver side cannot store the data length (server side only). |
| PCANTP_NETSTATUS_ERROR | 0x09 | General error. |
| PCANTP_NETSTATUS_IGNORED | 0x0A | Message was invalid and ignored. |
| PCANTP_NETSTATUS_TIMEOUT_As | 0x10 | Sender side timeout while transmitting. |
| PCANTP_NETSTATUS_TIMEOUT_Ar | 0x11 | Receiver side timeout while transmitting. |
| PCANTP_NETSTATUS_XMT_FULL | 0x11 | Transmit queue is full (failed too many times; NON PCANTP related network results). |
| PCANTP_NETSTATUS_BUS_ERROR | 0x12 | CAN bus error (NON PCANTP related network results). |
| PCANTP_NETSTATUS_NO_MEMORY | 0x13 | Memory allocation error (NON PCANTP related network results). |

See also: `cantp_msg` on page 118, `cantp_msgdata` on page 125.

3.6.17 cantp_netaddrinfo

Represents the network address information of a PCANTP message.

Syntax

C/C++

```
typedef struct _cantp_netaddrinfo {
    cantp_isotp_msgtype msgtype;
    cantp_isotp_format format;
    cantp_isotp_addressing target_type;
    uint16_t source_addr;
    uint16_t target_addr;
    uint8_t extension_addr;
} cantp_netaddrinfo;
```

Pascal OO

```
cantp_netaddrinfo = record
    msgtype: cantp_isotp_msgtype;
    format: cantp_isotp_format;
    target_type: cantp_isotp_addressing;
    source_addr: UInt16;
    target_addr: UInt16;
    extension_addr: Byte;
end;
```

C#

```
[StructLayout(LayoutKind.Sequential, Pack = 8)]
public struct cantp_netaddrinfo
{
    [MarshalAs(UnmanagedType.U4)]
    public cantp_isotp_msgtype msgtype;
    [MarshalAs(UnmanagedType.U4)]
    public cantp_isotp_format format;
    [MarshalAs(UnmanagedType.U4)]
    public cantp_isotp_addressing target_type;
    public UInt16 source_addr;
    public UInt16 target_addr;
    public byte extension_addr;
}
```

C++/CLR

```
[StructLayout(LayoutKind::Sequential, Pack = 8)]
public value struct cantp_netaddrinfo
{
    [MarshalAs(UnmanagedType::U4)]
    cantp_isotp_msgtype msgtype;
    [MarshalAs(UnmanagedType::U4)]
    cantp_isotp_format format;
    [MarshalAs(UnmanagedType::U4)]
    cantp_isotp_addressing target_type;
    UInt16 source_addr;
    UInt16 target_addr;
    Byte extension_addr;
};
```

Visual Basic

```
<StructLayout(LayoutKind.Sequential, Pack:=8)>
Public Structure cantp_netaddrinfo
    <MarshalAs(UnmanagedType.U4)>
    Public msgtype As cantp_isotp_msgtype
    <MarshalAs(UnmanagedType.U4)>
    Public format As cantp_isotp_format
    <MarshalAs(UnmanagedType.U4)>
    Public target_type As cantp_isotp_addressing
    Public source_addr As UInt16
    Public target_addr As UInt16
    Public extension_addr As Byte
End Structure
```

Fields

| Name | Description |
|----------------|---|
| msgtype | PCANTP message type (see cantp_isotp_msgtype on page 136). |
| format | PCANTP format addressing (see cantp_isotp_format on page 137). |
| target_type | PCANTP addressing/target type (see cantp_isotp_addressing on page 115). |
| source_addr | Source address. |
| target_addr | Target address. |
| extension_addr | Extension address. |

See also: `cantp_msg` on page 118, `cantp_msgdata` on page 125.

3.6.18 cantp_isotp_msgtype

Represents the addressing format of a PCANTP message (see field `cantp_msg.msgdata.isotp.netaddrinfo.msgtype` on page 118).

Syntax

C/C++

```
typedef enum _cantp_isotp_msgtype {
    PCANTP_ISOTP_MSGTYPE_UNKNOWN = 0x00,
    PCANTP_ISOTP_MSGTYPE_DIAGNOSTIC = 0x01,
    PCANTP_ISOTP_MSGTYPE_REMOTE_DIAGNOSTIC = 0x02,
    PCANTP_ISOTP_MSGTYPE_FLAG_INDICATION_RX = 0x10,
    PCANTP_ISOTP_MSGTYPE_FLAG_INDICATION_TX = 0x20,
    PCANTP_ISOTP_MSGTYPE_FLAG_INDICATION = (0x10 | 0x20),
    PCANTP_ISOTP_MSGTYPE_MASK_INDICATION = 0x0F
} cantp_isotp_msgtype;
```

Pascal OO

```
cantp_isotp_msgtype = (
    PCANTP_ISOTP_MSGTYPE_UNKNOWN = $00,
    PCANTP_ISOTP_MSGTYPE_DIAGNOSTIC = $01,
    PCANTP_ISOTP_MSGTYPE_REMOTE_DIAGNOSTIC = $02,
    PCANTP_ISOTP_MSGTYPE_FLAG_INDICATION_RX = $10,
    PCANTP_ISOTP_MSGTYPE_FLAG_INDICATION_TX = $20,
    PCANTP_ISOTP_MSGTYPE_FLAG_INDICATION = ($10 Or $20),
    PCANTP_ISOTP_MSGTYPE_MASK_INDICATION = $0F
);
```

C#

```
public enum cantp_isotp_msgtype : UInt32
{
    PCANTP_ISOTP_MSGTYPE_UNKNOWN = 0x00,
    PCANTP_ISOTP_MSGTYPE_DIAGNOSTIC = 0x01,
    PCANTP_ISOTP_MSGTYPE_REMOTE_DIAGNOSTIC = 0x02,
    PCANTP_ISOTP_MSGTYPE_FLAG_INDICATION_RX = 0x10,
    PCANTP_ISOTP_MSGTYPE_FLAG_INDICATION_TX = 0x20,
    PCANTP_ISOTP_MSGTYPE_FLAG_INDICATION = (0x10 | 0x20),
    PCANTP_ISOTP_MSGTYPE_MASK_INDICATION = 0x0F
}
```

C++ / CLR

```
public enum cantp_isotp_msgtype : UInt32
{
    PCANTP_ISOTP_MSGTYPE_UNKNOWN = 0x00,
    PCANTP_ISOTP_MSGTYPE_DIAGNOSTIC = 0x01,
    PCANTP_ISOTP_MSGTYPE_REMOTE_DIAGNOSTIC = 0x02,
    PCANTP_ISOTP_MSGTYPE_FLAG_INDICATION_RX = 0x10,
    PCANTP_ISOTP_MSGTYPE_FLAG_INDICATION_TX = 0x20,
    PCANTP_ISOTP_MSGTYPE_FLAG_INDICATION = (0x10 | 0x20),
    PCANTP_ISOTP_MSGTYPE_MASK_INDICATION = 0x0F
};
```

Visual Basic

```
Public Enum cantp_isotp_msgtype As UInt32
    PCANTP_ISOTP_MSGTYPE_UNKNOWN = &H0
    PCANTP_ISOTP_MSGTYPE_DIAGNOSTIC = &H1
```



```

PCANTP_ISOTP_MSGTYPE_REMOTE_DIAGNOSTIC = &H2
PCANTP_ISOTP_MSGTYPE_FLAG_INDICATION_RX = &H10
PCANTP_ISOTP_MSGTYPE_FLAG_INDICATION_TX = &H20
PCANTP_ISOTP_MSGTYPE_FLAG_INDICATION = (&H10 Or &H20)
PCANTP_ISOTP_MSGTYPE_MASK_INDICATION = &HF

```

End Enum

Values

| Name | Value | Description |
|---|-------|---|
| PCANTP_ISOTP_MSGTYPE_UNKNOWN | 0x00 | Unknown (non-PCAN-ISO-TP) message. |
| PCANTP_ISOTP_MSGTYPE_DIAGNOSTIC | 0x01 | Diagnostic message (request or confirmation). |
| PCANTP_ISOTP_MSGTYPE_REMOTE_DIAGNOSTIC | 0x02 | Remote Diagnostic message (request or confirmation). |
| PCANTP_ISOTP_MSGTYPE_FLAG_INDICATION_RX | 0x10 | Multi-Frame message is being received. |
| PCANTP_ISOTP_MSGTYPE_FLAG_INDICATION_TX | 0x20 | Multi-Frame message is being transmitted. |
| PCANTP_ISOTP_MSGTYPE_FLAG_INDICATION | 0x30 | Multi-Frame message is being communicated (Tx or Rx). |
| PCANTP_ISOTP_MSGTYPE_MASK_INDICATION | 0x0F | Mask to remove Indication flags. |

See also: `cantp_msg` on page 118, `cantp_msgdata` on page 125, `cantp_msgdata_isotp` on page 129, `cantp_netaddrinfo` on page 134.

3.6.19 cantp_isotp_format

Represents the addressing format of a PCANTP message (see field `cantp_msg.msgdata.isotp.netaddrinfo.format` on page 118).

Syntax

C/C++

```

typedef enum _cantp_isotp_format {
    PCANTP_ISOTP_FORMAT_UNKNOWN = 0xFF,
    PCANTP_ISOTP_FORMAT_NONE = 0x00,
    PCANTP_ISOTP_FORMAT_NORMAL = 0x01,
    PCANTP_ISOTP_FORMAT_FIXED_NORMAL = 0x02,
    PCANTP_ISOTP_FORMAT_EXTENDED = 0x03,
    PCANTP_ISOTP_FORMAT_MIXED = 0x04,
    PCANTP_ISOTP_FORMAT_ENHANCED = 0x05,
} cantp_isotp_format;

```

Pascal OO

```

cantp_isotp_format = (
    PCANTP_ISOTP_FORMAT_UNKNOWN = $FF,
    PCANTP_ISOTP_FORMAT_NONE = $00,
    PCANTP_ISOTP_FORMAT_NORMAL = $01,
    PCANTP_ISOTP_FORMAT_FIXED_NORMAL = $02,
    PCANTP_ISOTP_FORMAT_EXTENDED = $03,
    PCANTP_ISOTP_FORMAT_MIXED = $04,
    PCANTP_ISOTP_FORMAT_ENHANCED = $05
);

```

C#

```

public enum cantp_isotp_format : UInt32
{
    PCANTP_ISOTP_FORMAT_UNKNOWN = 0xFF,
    PCANTP_ISOTP_FORMAT_NONE = 0x00,
    PCANTP_ISOTP_FORMAT_NORMAL = 0x01,
    PCANTP_ISOTP_FORMAT_FIXED_NORMAL = 0x02,

```

```

PCANTP_ISOTP_FORMAT_EXTENDED = 0x03,
PCANTP_ISOTP_FORMAT_MIXED = 0x04,
PCANTP_ISOTP_FORMAT_ENHANCED = 0x05,
}

```

C++ / CLR

```

public enum cantp_isotp_format : UInt32
{
    PCANTP_ISOTP_FORMAT_UNKNOWN = 0xFF,
    PCANTP_ISOTP_FORMAT_NONE = 0x00,
    PCANTP_ISOTP_FORMAT_NORMAL = 0x01,
    PCANTP_ISOTP_FORMAT_FIXED_NORMAL = 0x02,
    PCANTP_ISOTP_FORMAT_EXTENDED = 0x03,
    PCANTP_ISOTP_FORMAT_MIXED = 0x04,
    PCANTP_ISOTP_FORMAT_ENHANCED = 0x05,
};

```

Visual Basic

```

Public Enum cantp_isotp_format As UInt32
    PCANTP_ISOTP_FORMAT_UNKNOWN = &HFF
    PCANTP_ISOTP_FORMAT_NONE = &H0
    PCANTP_ISOTP_FORMAT_NORMAL = &H1
    PCANTP_ISOTP_FORMAT_FIXED_NORMAL = &H2
    PCANTP_ISOTP_FORMAT_EXTENDED = &H3
    PCANTP_ISOTP_FORMAT_MIXED = &H4
    PCANTP_ISOTP_FORMAT_ENHANCED = &H5
End Enum

```

Values







| Name | Value | Description |
|----------------------------------|-------|--|
| PCANTP_ISOTP_FORMAT_UNKNOWN | 0xFF | Unknown addressing format. |
| PCANTP_ISOTP_FORMAT_NONE | 0x00 | Unsegmented CAN frame. |
| PCANTP_ISOTP_FORMAT_NORMAL | 0x01 | Normal addressing format from ISO 15765-2. |
| PCANTP_ISOTP_FORMAT_FIXED_NORMAL | 0x02 | Fixed normal addressing format from ISO 15765-2. |
| PCANTP_ISOTP_FORMAT_EXTENDED | 0x03 | Extended addressing format from ISO 15765-2. |
| PCANTP_ISOTP_FORMAT_MIXED | 0x04 | Mixed addressing format from ISO 15765-2. |
| PCANTP_ISOTP_FORMAT_ENHANCED | 0x05 | Enhanced addressing format from ISO 15765-3. |

See also: [cantp_msg](#) on page 118, [cantp_msgdata_isotp](#) on page 129, [cantp_netaddrinfo](#) on page 134.













3.7 Methods

The methods defined for the classes `UDSApi` and `TUDSApi` are divided into 6 groups of functionality. Note that these methods are static and can be called in the name of the class, without instantiation.















Connection

| | Method | Description |
|---|--------------------------------|---|
|   | <code>Initialize_2013</code> | Initializes a PUDS channel based on a PCANTP handle (without CAN FD support). |
|   | <code>InitializeFD_2013</code> | Initializes a PUDS channel based on a PCANTP handle (including CAN FD support). |
|   | <code>Uninitialize_2013</code> | Uninitializes a PUDS channel. |









Configuration

| | Method | Description |
|---|--|--|
|   | <code>SetValue_2013</code> | Sets a configuration or information value within a PUDS channel. |
|   | <code>AddMapping_2013</code> | Adds a mapping between a CAN identifier and a network address information. |
|   | <code>RemoveMappingByCanId_2013</code> | Removes all user defined PUDS mappings corresponding to a CAN identifier. |
|   | <code>RemoveMapping_2013</code> | Removes a user defined PUDS mapping. |
|   | <code>AddCanIdFilter_2013</code> | Adds an entry to the CAN identifier white-list filtering. |
|   | <code>RemoveCanIdFilter_2013</code> | Removes an entry from the CAN identifier white-list filtering. |









Information






| | Method | Description |
|---|---|---|
|   | <code>GetValue_2013</code> | Retrieves information from a PUDS channel. |
|   | <code>GetCanBusStatus_2013</code> | Gets information about the internal BUS status of a PUDS channel. |
|   | <code>GetMapping_2013</code> | Retrieves a mapping matching the given CAN identifier and message type (11bits, 29 bits, FD, etc.). |
|   | <code>GetMappings_2013</code> | Retrieves all the mappings defined for a PUDS channel. |
|   | <code>GetSessionInformation_2013</code> | Gets current ECU session information. |
|   | <code>StatusIsOk_2013</code> | Checks if a PUDS status matches an expected result (default is PUDS_STATUS_OK). |
|   | <code>GetErrorText_2013</code> | Gets a descriptive text for an error code. |

Message handling




| | Method | Description |
|---|----------------------------|---|
|   | <code>MsgAlloc_2013</code> | Allocates a PUDS message based using the given configuration. |
|   | <code>MsgFree_2013</code> | Deallocates a PUDS message. |
|   | <code>MsgCopy_2013</code> | Copies a PUDS message to another buffer. |
|   | <code>MsgMove_2013</code> | Moves a PUDS message to another buffer (and cleans the original message structure). |

Communication







| | Method | Description |
|---|--|--|
|   | <code>Read_2013</code> | Reads a CAN message from the receive queue of a PUDS channel. |
|   | <code>Write_2013</code> | Transmits a message using a connected PUDS channel. |
|   | <code>Reset_2013</code> | Resets the receive and transmit queues of a PUDS channel. |
|   | <code>WaitForSingleMessage_2013</code> | Waits for a message (a response or a transmit confirmation) based on a PUDS message request. |

| | Method | Description |
|---|--|---|
|  | WaitForFunctionalResponses_2013 | Waits for multiple messages (multiple responses from a functional request for instance) based on a PUDS message request. |
|  | WaitForService_2013 | Handles the communication workflow for a UDS service expecting a single response. |
|  | WaitForServiceFunctional_2013 | Handles the communication workflow for a UDS service expecting multiple responses. |
|  | SvcDiagnosticSessionControl_2013 | Writes to the transmit queue a request for UDS service DiagnosticSessionControl. |
|  | SvcECUReset_2013 | Writes to the transmit queue a request for UDS service ECUReset. |
|  | SvcSecurityAccess_2013 | Writes to the transmit queue a request for UDS service SecurityAccess. |
|  | SvcCommunicationControl_2013 | Writes to the transmit queue a request for UDS service CommunicationControl. |
|  | SvcTesterPresent_2013 | Writes to the transmit queue a request for UDS service TesterPresent. |
|  | SvcSecuredDataTransmission_2013 | Writes to the transmit queue a request for UDS service SecuredDataTransmission(ISO-14229-1:2013). |
|  | SvcSecuredDataTransmission_2020 | Writes to the transmit queue a request for UDS service SecuredDataTransmission(ISO-14229-1:2020). |
|  | SvcControlDTCSetting_2013 | Writes to the transmit queue a request for UDS service ControlDTCSetting. |
|  | SvcResponseOnEvent_2013 | Writes to the transmit queue a request for UDS service ResponseOnEvent. |
|  | SvcLinkControl_2013 | Writes to the transmit queue a request for UDS service LinkControl. |
|  | SvcReadDataByIdentifier_2013 | Writes to the transmit queue a request for UDS service ReadDataByIdentifier. |
|  | SvcReadMemoryByAddress_2013 | Writes to the transmit queue a request for UDS service ReadMemoryByAddress. |
|  | SvcReadScalingDataByIdentifier_2013 | Writes to the transmit queue a request for UDS service ReadScalingDataByIdentifier. |
|  | SvcReadDataByPeriodicIdentifier_2013 | Writes to the transmit queue a request for UDS service ReadDataByPeriodicIdentifier. |
|  | SvcDynamicallyDefineDataIdentifierDBID_2013 | Writes to the transmit queue a request for UDS service DynamicallyDefineDataIdentifier. |
|  | SvcDynamicallyDefineDataIdentifierDBMA_2013 | Writes to the transmit queue a request for UDS service DynamicallyDefineDataIdentifier. |
|  | SvcDynamicallyDefineDataIdentifierCDDDI_2013 | Writes to the transmit queue a request for UDS service DynamicallyDefineDataIdentifier. |
|  | SvcDynamicallyDefineDataIdentifierClearAllIDDDI_2013 | Writes to the transmit queue a request for UDS service DynamicallyDefineDataIdentifier/clearDynamicallyDefinedDataIdentifier subfunction. |
|  | SvcWriteDataByIdentifier_2013 | Writes to the transmit queue a request for UDS service WriteDataByIdentifier. |
|  | SvcWriteMemoryByAddress_2013 | Writes to the transmit queue a request for UDS service WriteMemoryByAddress. |
|  | SvcClearDiagnosticInformation_2013 | Writes to the transmit queue a request for UDS service ClearDiagnosticInformation. |
|  | SvcClearDiagnosticInformation_2020 | Writes to the transmit queue a request for UDS service ClearDiagnosticInformation with memory selection parameter (ISO-14229-1:2020). |
|  | SvcReadDTCInformation_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |

| | Method | Description |
|---|--|--|
|  | SvcReadDTCInformationRDTCSBDTC_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | SvcReadDTCInformationRDTCSBRN_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | SvcReadDTCInformationReportExtended_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | SvcReadDTCInformationReportSeverity_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | SvcReadDTCInformationRSIODTC_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | SvcReadDTCInformationNoParam_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | SvcReadDTCInformationRDTCEDBR_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | SvcReadDTCInformationRUDMDTCBSM_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | SvcReadDTCInformationRUDMDTCSSBDTC_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | SvcReadDTCInformationRUDMDTCEDRBDN_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | SvcReadDTCInformationRDTCEDI_2020 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | SvcReadDTCInformationRWWHOBDDTCBMR_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | SvcReadDTCInformationRWWHOBDDTCWPS_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | SvcReadDTCInformationRDTCBRGI_2020 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | SvcInputOutputControlByIdentifier_2013 | Writes to the transmit queue a request for UDS service InputOutputControlByIdentifier. |
|  | SvcRoutineControl_2013 | Writes to the transmit queue a request for UDS service RoutineControl. |
|  | SvcRequestDownload_2013 | Writes to the transmit queue a request for UDS service RequestDownload. |
|  | SvcRequestUpload_2013 | Writes to the transmit queue a request for UDS service RequestUpload. |
|  | SvcTransferData_2013 | Writes to the transmit queue a request for UDS service TransferData. |
|  | SvcRequestTransferExit_2013 | Writes to the transmit queue a request for UDS service RequestTransferExit. |
|  | SvcAccessTimingParameter_2013 | Writes to the transmit queue a request for UDS service AccessTimingParameter. |
|  | SvcRequestFileTransfer_2013 | Writes to the transmit queue a request for UDS service RequestFileTransfer. |
|  | SvcAuthenticationDA_2020 | Writes to the transmit queue a request for UDS service Authentication with deAuthenticate subfunction (ISO-14229-1:2020). |
|  | SvcAuthenticationVCU_2020 | Writes to the transmit queue a request for UDS service Authentication with verifyCertificateUnidirectional subfunction (ISO-14229-1:2020). |
|  | SvcAuthenticationVCB_2020 | Writes to the transmit queue a request for UDS service Authentication with verifyCertificateBidirectional subfunction (ISO-14229-1:2020). |
|  | SvcAuthenticationPOWN_2020 | Writes to the transmit queue a request for UDS service Authentication with proofOfOwnership subfunction (ISO-14229-1:2020). |
|  | SvcAuthenticationRCFA_2020 | Writes to the transmit queue a request for UDS service Authentication with requestChallengeForAuthentication subfunction (ISO-14229-1:2020). |

| | Method | Description |
|---|------------------------------|---|
|  | SvcAuthenticationVPOWNU_2020 | Writes to the transmit queue a request for UDS service Authentication with verifyProofOfOwnershipUnidirectional subfunction (ISO-14229-1:2020). |
|  | SvcAuthenticationVPOWNB_2020 | Writes to the transmit queue a request for UDS service Authentication with verifyProofOfOwnershipBidirectional subfunction (ISO-14229-1:2020). |
|  | SvcAuthenticationAC_2020 | Writes to the transmit queue a request for UDS service Authentication with authenticationConfiguration subfunction (ISO-14229-1:2020). |



Helper (C# and VB specifics methods)

| | Method | Description |
|---|-----------------------|--|
|  | GetDataServiceId_2013 | Gets PUDS message data service identifier in a safe way. |
|  | SetDataServiceId_2013 | Sets PUDS message data service id in a safe way. |
|  | GetDataNrc_2013 | Gets PUDS message data negative response code (nrc) in a safe way. |
|  | SetDataNrc_2013 | Sets PUDS message data negative response code (nrc) in a safe way. |
|  | GetDataParameter_2013 | Gets PUDS message data parameter in a safe way. |
|  | SetDataParameter_2013 | Sets PUDS message data parameter in a safe way. |

3.7.1 Initialize_2013

Initializes a PUDS channel based on a PCANTP handle (without CAN FD support).

Overloads

| | Method | Description |
|---|---|---|
|  | Initialize_2013(cantp_handle, cantp_baudrate) | Initializes a Plug-And-Play PUDS channel based on a PCANTP handle (without CAN FD support). |
|  | Initialize_2013(cantp_handle, cantp_baudrate, cantp_hwtype, UInt32, UInt16) | Initializes a Non-Plug-And-Play PUDS channel based on a PCANTP handle (without CAN FD support). |

Plain function version: [UDS_Initialize_2013](#) on page 624.

3.7.2 Initialize_2013(cantp_handle, cantp_baudrate)

Initializes a PUDS channel which represents a Plug & Play PCAN-Device (without CAN FD support).

Syntax

Pascal OO

```
class function Initialize_2013(
    channel: cantp_handle;
    baudrate: cantp_baudrate
): uds_status; overload;
```

C#

```
public static uds_status Initialize_2013(
    cantp_handle channel,
    cantp_baudrate baudrate);
```

C++ / CLR

```
static uds_status Initialize_2013(
    cantp_handle channel,
    cantp_baudrate baudrate);
```

Visual Basic

```
Public Shared Function Initialize_2013(
    ByVal channel As cantp_handle,
    ByVal baudrate As cantp_baudrate) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| baudrate | The speed for the communication (see cantp_baudrate on page 116). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---------------------------------|---|
| PUDS_STATUS_ALREADY_INITIALIZED | Indicates that the desired PUDS channel is already in use. |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory. |
| PUDS_STATUS_NOT_INITIALIZED | Channel not available. |
| PUDS_STATUS_FLAG_PCAN_STATUS | This error flag states that the error is composed of a more precise PCAN-Basic error. |

Remarks

As indicated by its name, the `Initialize_2013` method initiates a PUDS channel, preparing it for communication within the CAN bus connected to it. Calls to the other methods will fail if they are used with a channel handle, different than `PCANTP_HANDLE_NONEBUS`, that has not been initialized yet. Each initialized channel should be released when it is not needed anymore.

Initializing a PUDS channel means:

- To reserve the channel for the calling application/process.
- To allocate channel resources, like receive and transmit queues.
- To forward initialization to PCAN-ISO-TP 3.x API and PCAN-Basic API, hence registering/connecting the Hardware denoted by the channel handle.
- To set up the default values of the different parameters (see `SetValue_2013` on page 153).
- To configure default standard ISO-TP mappings (see UDS and ISO-TP Network Addressing Information on page 770):
 - Functional request using 11 bits CAN identifier and normal addressing, from External Test Equipment address (`PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT`) to OBD functional address (`PUDS_ISO_15765_4_ADDR_OBD_FUNCTIONAL`)
 - Physical requests and responses using 11 bits CAN identifier and normal addressing, between the External Test Equipment address (`PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT`) and standard ECU addresses (ECU #1 to #8)
- To configure PCAN-ISO-TP 3.x to filter CAN frames to increase performance (frames that do not match a mapping or a CAN ID in the white-list filter are ignored and discarded).

The initialization process will fail if an application tries to initialize a PUDS channel that has already been initialized within the same process.

Take into consideration that initializing a channel causes a reset of the CAN hardware. In this way errors like BUSOFF, BUSHEAVY, and BUSLIGHT, are removed.

Example

The following example shows the initialize and uninitialized processes for a Plug-And-Play channel (channel 2 of a PCAN-PCI hardware).

C#

```
uds_status result;

// The Plug & Play channel (PCAN-PCI) is initialized
result = UDSApi.Initialize_2013(cantp_handle.PCANTP_HANDLE_PCIBUS2,
    cantp_baudrate.PCANTP_BAUDRATE_500K);
if (!UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Initialization failed", "Error");
else
    MessageBox.Show("PCAN-PCI (Ch-2) was initialized", "Success");

// All initialized channels are released
UDSApi.Uninitialize_2013(cantp_handle.PCANTP_HANDLE_NONEBUS);
```

C++/CLR

```
uds_status result;

// The Plug & Play channel (PCAN-PCI) is initialized
result = UDSApi::Initialize_2013(PCANTP_HANDLE_PCIBUS2, PCANTP_BAUDRATE_500K);
if (!UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Initialization failed", "Error");
else
    MessageBox::Show("PCAN-PCI (Ch-2) was initialized", "Success");

// All initialized channels are released
UDSApi::Uninitialize_2013(PCANTP_HANDLE_NONEBUS);
```

Visual Basic

```
Dim result As uds_status

' The Plug & Play channel (PCAN-PCI) Is initialized
result = UDSApi.Initialize_2013(cantp_handle.PCANTP_HANDLE_PCIBUS2,
    cantp_baudrate.PCANTP_BAUDRATE_500K)
If Not UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Initialization failed", "Error")
Else
    MessageBox.Show("PCAN-PCI (Ch-2) was initialized", "Success")
End If

' All initialized channels are released
UDSApi.Uninitialize_2013(cantp_handle.PCANTP_HANDLE_NONEBUS)
```

Pascal OO

```
var
    result: uds_status;
begin

    // The Plug & Play channel (PCAN-PCI) is initialized
    result := TUDSApi.Initialize_2013(cantp_handle.PCANTP_HANDLE_PCIBUS2,
        cantp_baudrate.PCANTP_BAUDRATE_500K);
    if not TUDSApi.StatusIsOk_2013(result) then
    begin
        MessageBox(0, 'Initialization failed', 'Error', MB_OK);
    end
    else
    begin
        MessageBox(0, 'PCAN-PCI (Ch-2) was initialized', 'Success', MB_OK);
    end
```



```
end;

// All initialized channels are released
TUDSApi.Uninitialize_2013(cantp_handle.PCANTP_HANDLE_NONEBUS);
end;
```

See also: [Uninitialize_2013](#) on page 148, [InitializeFD_2013](#) on page 148, Using PCAN-UDS 2.x on page 11.
Plain function version: [UDS_Initialize_2013](#) on page 624.

3.7.3 Initialize_2013(cantp_handle, cantp_baudrate, cantp_hwtype, UInt32, UInt16)

Initializes a PUDS channel which represents a Non-Plug & Play PCAN-Device (without CAN FD support).

Syntax

Pascal OO

```
class function Initialize_2013(
    channel: cantp_handle;
    baudrate: cantp_baudrate;
    hw_type: cantp_hwtype;
    io_port: UInt32;
    interrupt: UInt16
): uds_status; overload;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_Initialize_2013")]
public static extern uds_status Initialize_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    [MarshalAs(UnmanagedType.U4)]
    cantp_baudrate baudrate,
    [MarshalAs(UnmanagedType.U4)]
    cantp_hwtype hw_type,
    UInt32 io_port,
    UInt16 interrupt);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_Initialize_2013")]
static uds_status Initialize_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    [MarshalAs(UnmanagedType::U4)]
    cantp_baudrate baudrate,
    [MarshalAs(UnmanagedType::U4)]
    cantp_hwtype hw_type,
    UInt32 io_port,
    UInt16 interrupt);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_Initialize_2013")>
Public Shared Function Initialize_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    <MarshalAs(UnmanagedType.U4)>
    ByVal baudrate As cantp_baudrate,
```

```

<MarshalAs(UnmanagedType.U4)>
ByVal hw_type As cantp_hwtype,
ByVal io_port As UInt32,
ByVal interrupt As UInt16) As uds_status
End Function

```

Parameters

| Parameter | Description |
|-----------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| baudrate | The speed for the communication (see cantp_baudrate on page 116). |
| hw_type | Non-plug and play: The type of hardware (see cantp_hwtype on page 113). |
| io_port | Non-plug and play: The I/O address for the parallel port. |
| interrupt | Non-plug and play: Interrupt number of the parallel port. |

Returns

The return value is a uds_status code. PUDS_STATUS_OK is returned on success. The typical errors in case of failure are:

| | |
|---------------------------------|---|
| PUDS_STATUS_ALREADY_INITIALIZED | Indicates that the desired PUDS channel is already in use. |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory. |
| PUDS_STATUS_NOT_INITIALIZED | Channel not available. |
| PUDS_STATUS_FLAG_PCAN_STATUS | This error flag states that the error is composed of a more precise PCAN-Basic error. |

Remarks

As indicated by its name, the `Initialize_2013` method initiates a PUDS channel, preparing it for communication within the CAN bus connected to it. Calls to the other methods will fail if they are used with a channel handle, different than `PCANTP_HANDLE_NONEBUS`, that has not been initialized yet. Each initialized channel should be released when it is not needed anymore.

Initializing a PUDS channel means:

- to reserve the channel for the calling application/process
- to allocate channel resources, like receive and transmit queues
- to forward initialization to PCAN-ISO-TP 3.x API and PCAN-Basic API, hence registering/connecting the Hardware denoted by the channel handle
- To set up the default values of the different parameters (see `SetValue_2013` on page 153).
- To configure default standard ISO-TP mappings (see UDS and ISO-TP Network Addressing Information on page 770):
 - Functional request using 11 bits CAN identifier and normal addressing, from External Test Equipment address (`PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT`) to OBD functional address (`PUDS_ISO_15765_4_ADDR_OBD_FUNCTIONAL`) ,
 - Physical requests and responses using 11 bits CAN identifier and normal addressing, between the External Test Equipment address (`PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT`) and standard ECU addresses (ECU #1 to #8)
- To configure PCAN-ISO-TP 3.x to filter CAN frames to increase performance (frames that do not match a mapping or a CAN ID in the white-list filter are ignored and discarded).

The initialization process will fail if an application tries to initialize a PUDS channel that has already been initialized within the same process.

Take into consideration that initializing a channel causes a reset of the CAN hardware. In this way errors like BUSOFF, BUSHEAVY, and BUSLIGHT, are removed.

Example

The following example shows the initialize and uninitialized processes for a Non-Plug-And-Play channel (channel 1 of the PCAN-DNG).

C#

```
uds_status result;

// The Plug & Play channel (PCAN-PCI) is initialized
result = UDSApi.Initialize_2013(cantp_handle.PCANTP_HANDLE_DNGBUS1,
    cantp_baudrate.PCANTP_BAUDRATE_500K, cantp_hwtype.PCANTP_HWTYPE_DNG_SJA, 0x378, 7);
if (!UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Initialization failed", "Error");
else
    MessageBox.Show("PCAN-PCI (Ch-2) was initialized", "Success");

// All initialized channels are released
UDSApi.Uninitialize_2013(cantp_handle.PCANTP_HANDLE_NONEBUS);
```

C++/CLR

```
uds_status result;

// The Plug & Play channel (PCAN-PCI) is initialized
result = UDSApi::Initialize_2013(PCANTP_HANDLE_DNGBUS1, PCANTP_BAUDRATE_500K,
    PCANTP_HWTYPE_DNG_SJA, 0x378, 7);
if (!UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Initialization failed", "Error");
else
    MessageBox::Show("PCAN-PCI (Ch-2) was initialized", "Success");

// All initialized channels are released
UDSApi::Uninitialize_2013(PCANTP_HANDLE_NONEBUS);
```

Visual Basic

```
Dim result As uds_status

' The Plug & Play channel (PCAN-PCI) Is initialized
result = UDSApi.Initialize_2013(cantp_handle.PCANTP_HANDLE_DNGBUS1,
    cantp_baudrate.PCANTP_BAUDRATE_500K, cantp_hwtype.PCANTP_HWTYPE_DNG_SJA, &H378, 7)
If Not UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Initialization failed", "Error")
Else
    MessageBox.Show("PCAN-PCI (Ch-2) was initialized", "Success")
End If

' All initialized channels are released
UDSApi.Uninitialize_2013(cantp_handle.PCANTP_HANDLE_NONEBUS)
```

Pascal OO

```
var
    result: uds_status;
begin

    // The Plug & Play channel (PCAN-PCI) is initialized
    result := TUDSApi.Initialize_2013(cantp_handle.PCANTP_HANDLE_DNGBUS1,
        cantp_baudrate.PCANTP_BAUDRATE_500K,
        cantp_hwtype.PCANTP_HWTYPE_DNG_SJA, $378, 7);
    if not TUDSApi.StatusIsOk_2013(result) then
    begin
        MessageBox(0, 'Initialization failed', 'Error', MB_OK);
```

```

end
else
begin
    MessageBox(0, 'PCAN-PCI (Ch-2) was initialized', 'Success', MB_OK);
end;

// All initialized channels are released
TUDSApi.Uninitialize_2013(cantp_handle.PCANTP_HANDLE_NONEBUS);
end;

```

See also: [Uninitialize_2013](#) on page 148, [InitializeFD_2013](#) on page 148, Using PCAN-UDS 2.x on page 11.
Plain function version: [UDS_Initialize_2013](#) on page 624.

3.7.4 InitializeFD_2013

Initializes a PUDS channel based on a PCANTP handle (including CAN FD support).

Syntax

Pascal OO

```

class function InitializeFD_2013(
    channel: cantp_handle;
    const bitrate_fd: cantp_bitrate
): uds_status;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_InitializeFD_2013")]
public static extern uds_status InitializeFD_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    cantp_bitrate bitrate_fd);

```

C++ / CLR

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_InitializeFD_2013")]
static uds_status InitializeFD_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    cantp_bitrate bitrate_fd);

```

Visual Basic

```

<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_InitializeFD_2013")>
Public Shared Function InitializeFD_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal bitrate_fd As cantp_bitrate) As uds_status
End Function

```

Parameters

| Parameter | Description |
|------------|---|
| channel | The handle of a FD capable PUDS channel (see cantp_handle on page 105). |
| bitrate_fd | The speed for the communication (see cantp_bitrate on page 101, FD Bit Rate Parameter Definitions on page 102). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_ALREADY_INITIALIZED</code> | Indicates that the desired PUDS channel is already in use. |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory. |
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Channel not available. |
| <code>PUDS_STATUS_FLAG_PCAN_STATUS</code> | This error flag states that the error is composed of a more precise PCAN-Basic error. |

Remarks

The `InitializeFD_2013` method initiates a FD capable PUDS channel, preparing it for communicate within the CAN bus connected to it. Calls to the other methods will fail, if they are used with a channel handle, different than `PCANTP_HANDLE_NONEBUS`, that has not been initialized yet. Each initialized channel should be released when it is not needed anymore.

Initializing a PUDS channel means:

- └ To reserve the channel for the calling application/process.
- └ To allocate channel resources, like receive and transmit queues.
- └ To forward initialization to PCAN-ISO-TP 3.x API and PCAN-Basic API, hence registering/connecting the Hardware denoted by the channel handle.
- └ To set up the default values of the different parameters (see `SetValue_2013` on page 153).
- └ To configure default standard ISO-TP mappings (see UDS and ISO-TP Network Addressing Information on page 770):
 - Functional request using 11 bits CAN identifier and normal addressing, from External Test Equipment address (`PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT`) to OBD functional address (`PUDS_ISO_15765_4_ADDR_OBD_FUNCTIONAL`),
 - Physical requests and responses using 11 bits CAN identifier and normal addressing, between the External Test Equipment address (`PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT`) and standard ECU addresses (ECU #1 to #8)
- └ To configure PCAN-ISO-TP 3.x to filter CAN frames to increase performance (frames that do not match a mapping or a CAN ID in the white-list filter are ignored and discarded).

The initialization process will fail if an application tries to initialize a PUDS channel that has already been initialized within the same process.

Take into consideration, that initializing a channel causes a reset of the CAN hardware. In this way errors like BUSOFF, BUSHEAVY, and BUSLIGHT, are removed.

Example

The following example shows the initialize and uninitialized processes for a Plug and Play, FD capable channel (channel 2 of a PCAN-USB hardware).

C#

```
uds_status result;

// The Plug and Play channel (PCAN-USB) is initialized @500kbps/2Mbps.
result = UDSApi.InitializeFD_2013(cantp_handle.PCANTP_HANDLE_USBBUS2, "f_clock=80000000,
nom_brp=10, nom_tseg1=12, nom_tseg2=3, nom_sjw=1, data_brp=4, data_tseg1=7, data_tseg2=2,
data_sjw=1");

if (!UDSApi.StatusIsOk_2013(result))
```

```

        MessageBox.Show("Initialization failed", "Error");
else
    MessageBox.Show("PCAN-USB (Ch-2) was initialized", "Success");

// All initialized channels are released.
UDSApi.Uninitialize_2013(cantp_handle.PCANTP_HANDLE_NONEBUS);

```

C++ / CLR

```

uds_status result;

// The Plug and Play channel (PCAN-USB) is initialized @500kbps/2Mbps.
result = UDSApi::InitializeFD_2013(PCANTP_HANDLE_USBBUS2, "f_clock=80000000, nom_brp=10,
nom_tseg1=12, nom_tseg2=3, nom_sjw=1, data_brp=4, data_tseg1=7, data_tseg2=2, data_sjw=1");
if (!UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Initialization failed", "Error");
else
    MessageBox::Show("PCAN-USB (Ch-2) was initialized", "Success");

// All initialized channels are released.
UDSApi::Uninitialize_2013(PCANTP_HANDLE_NONEBUS);

```

Visual Basic

```

Dim result As uds_status

' The Plug And Play channel (PCAN-USB) Is initialized @500kbps/2Mbps.
result = UDSApi.InitializeFD_2013(cantp_handle.PCANTP_HANDLE_USBBUS2, "f_clock=80000000,
nom_brp=10, nom_tseg1=12, nom_tseg2=3, nom_sjw=1, data_brp=4, data_tseg1=7, data_tseg2=2,
data_sjw=1")
If Not UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Initialization failed", "Error")
Else
    MessageBox.Show("PCAN-USB (Ch-2) was initialized", "Success")
End If

' All initialized channels are released.
UDSApi.Uninitialize_2013(cantp_handle.PCANTP_HANDLE_NONEBUS)

```

Pascal OO

```

var
    result: uds_status;
begin
    // The Plug and Play channel (PCAN-USB) is initialized @500kbps/2Mbps.
    result := TUDSApi.InitializeFD_2013(cantp_handle.PCANTP_HANDLE_USBBUS2,
    'f_clock=80000000, nom_brp=10, nom_tseg1=12, nom_tseg2=3, nom_sjw=1, data_brp=4,
    data_tseg1=7, data_tseg2=2, data_sjw=1');
    if not TUDSApi.StatusIsOk_2013(result) then
        begin
            MessageBox(0, 'Initialization failed', 'Error', MB_OK);
        end
    else
        begin
            MessageBox(0, 'PCAN-USB (Ch-2) was initialized', 'Success', MB_OK);
        end;

    // All initialized channels are released.
    TUDSApi.Uninitialize_2013(cantp_handle.PCANTP_HANDLE_NONEBUS);
end;

```

See also: [Uninitialize_2013](#) on page 148, Using PCAN-UDS 2.x on page 12, [cantp_bitrate](#) on page 101, FD Bit Rate Parameter Definitions on page 102.

Plain function version: [UDS_InitializeFD_2013](#) on page 626.

3.7.5 Uninitialize_2013

Uninitializes a PUDS channel.

Syntax

Pascal OO

```
class function Uninitialize_2013(
    channel: cantp_handle
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_Uninitialize_2013")]
public static extern uds_status Uninitialize_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_Uninitialize_2013")]
static uds_status Uninitialize_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_Uninitialize_2013")>
Public Shared Function Uninitialize_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |

Returns

The return value is a [uds_status](#) code. [PUDS_STATUS_OK](#) is returned on success. The typical error in case of failure is:

| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel cannot be uninitialized because it was not found in the list of reserved channels of the calling application. |
|---|---|

Remarks

A PUDS channel can be released using one of these possibilities:

- **Single-Release:** Given a handle of a PUDS channel initialized before with the method [Initialize_2013](#). If the given channel cannot be found, then an error is returned.
- **Multiple-Release:** Giving the handle value [PCANTP_HANDLE_NONEBUS](#) which instructs the API to search for all channels initialized by the calling application and release them all. This option causes no errors if no hardware were uninitialized.

Example

The following example shows the initialize and uninitialized processes for a Plug-And-Play channel (channel 2 of a PCAN-PCI hardware).

C#

```
uds_status result;

// The Plug & Play channel (PCAN-PCI) is initialized
result = UDSApi.Initialize_2013(cantp_handle.PCANTP_HANDLE_PCIBUS2,
    cantp_baudrate.PCANTP_BAUDRATE_500K);
if (!UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Initialization failed", "Error");
else
    MessageBox.Show("PCAN-PCI (Ch-2) was initialized", "Success");

// Release channel
result = UDSApi.Uninitialize_2013(cantp_handle.PCANTP_HANDLE_PCIBUS2);
if (!UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Uninitialization failed", "Error");
else
    MessageBox.Show("PCAN-PCI (Ch-2) was released", "Success");
```

C++/CLR

```
uds_status result;

// The Plug & Play channel (PCAN-PCI) is initialized
result = UDSApi::Initialize_2013(PCANTP_HANDLE_PCIBUS2, PCANTP_BAUDRATE_500K);
if (!UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Initialization failed", "Error");
else
    MessageBox::Show("PCAN-PCI (Ch-2) was initialized", "Success");

// Release channel
result = UDSApi::Uninitialize_2013(PCANTP_HANDLE_PCIBUS2);
if (!UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Uninitialization failed", "Error");
else
    MessageBox::Show("PCAN-PCI (Ch-2) was released", "Success");
```

Visual Basic

```
Dim result As uds_status

' The Plug & Play channel (PCAN-PCI) Is initialized
result = UDSApi.Initialize_2013(cantp_handle.PCANTP_HANDLE_PCIBUS2,
    cantp_baudrate.PCANTP_BAUDRATE_500K)
If Not UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Initialization failed", "Error")
Else
    MessageBox.Show("PCAN-PCI (Ch-2) was initialized", "Success")
End If

' Release channel
result = UDSApi.Uninitialize_2013(cantp_handle.PCANTP_HANDLE_PCIBUS2)
If Not UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Uninitialization failed", "Error")
```



```
Else
    MessageBox.Show("PCAN-PCI (Ch-2) was released", "Success")
End If
```

Pascal OO

```
var
    result: uds_status;
begin
    // The Plug & Play channel (PCAN-PCI) is initialized
    result := TUDSApi.Initialize_2013(cantp_handle.PCANTP_HANDLE_PCIBUS2,
        cantp_baudrate.PCANTP_BAUDRATE_500K);
    if not TUDSApi.StatusIsOk_2013(result) then
    begin
        MessageBox(0, 'Initialization failed', 'Error', MB_OK);
    end
    else
    begin
        MessageBox(0, 'PCAN-PCI (Ch-2) was initialized', 'Success', MB_OK);
    end;

    // Release channel
    result := TUDSApi.Uninitialize_2013(cantp_handle.PCANTP_HANDLE_PCIBUS2);
    if not TUDSApi.StatusIsOk_2013(result) then
    begin
        MessageBox(0, 'Uninitialization failed', 'Error', MB_OK);
    end
    else
    begin
        MessageBox(0, 'PCAN-PCI (Ch-2) was released', 'Success', MB_OK);
    end;
end;
```





See also: [Initialize_2013](#) on page 142, [InitializeFD_2013](#) on page 148.

Plain function version: [UDS_Uninitialize_2013](#) on page 628.

3.7.6 setValue_2013

Sets a configuration or information value within a PUDS channel.

Overloads

| | Method | Description |
|---|--|---|
|  | SetValue_2013(cantp_handle, uds_parameter, UInt32, UInt32) | Sets a configuration or information numeric value within a PUDS channel. |
|  | SetValue_2013(cantp_handle, uds_parameter, String, UInt32) | Sets a configuration or information string value within a PUDS channel. |
|  | SetValue_2013(cantp_handle, uds_parameter, Byte[], UInt32) | Sets a configuration or information with an array of bytes within a PUDS channel. |
|  | SetValue_2013(cantp_handle, uds_parameter, IntPtr, UInt32) | Sets a configuration or information within a PUDS channel. |

Plain function version: [UDS_SetValue_2013](#) on page 629.

3.7.7 SetValue_2013(cantp_handle, uds_parameter, UInt32, UInt32)

Sets a configuration or information numeric value within a PUDS channel.

Syntax

Pascal OO

```
class function SetValue_2013(
  channel: cantp_handle;
  parameter: uds_parameter;
  buffer: PLongWord;
  buffer_size: UInt32
): uds_status; overload;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SetValue_2013")]
public static extern uds_status SetValue_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    [MarshalAs(UnmanagedType.U4)]
    uds_parameter parameter,
    ref UInt32 buffer,
    UInt32 buffer_size);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SetValue_2013")]
static uds_status SetValue_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    [MarshalAs(UnmanagedType::U4)]
    uds_parameter parameter,
    UInt32 %buffer,
    UInt32 buffer_size);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SetValue_2013")>
Public Shared Function SetValue_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    <MarshalAs(UnmanagedType.U4)>
    ByVal parameter As uds_parameter,
    ByRef buffer As UInt32,
    ByVal buffer_size As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| parameter | The code of the value to be set (see uds_parameter on page 41). |
| buffer | The buffer containing the numeric value to be set. |
| buffer_size | The length in bytes of the given buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | Indicates that the parameters passed to the method are invalid. Check the value of the buffer and assert it is compatible with the buffer length. |

Remarks

Use the method `SetValue_2013` to set configuration information or environment values of a PUDS channel.

Note: That any calls with non PCAN-UDS 2.x API parameters (i.e. `uds_parameter`) will be forwarded to PCAN-ISO-TP 3.x API or PCAN-Basic API.

More information about the parameters and values that can be set can be found in Detailed Parameters Characteristics on page 45.

Since most of the PCAN-UDS 2.x API parameters require a numeric value (byte or integer) this is the most common and useful override.

Example

The following example shows the use of the method `SetValue_2013` on the channel `PCANTP_HANDLE_PCIBUS2` to enable debug mode.

Note: It is assumed that the channel was already initialized.

C#

```
uds_status result;
UInt32 buffer;

// Enable error messages
buffer = UDSApi.PUDS_DEBUG_LVL_ERROR;
result = UDSApi.SetValue_2013(cantp_handle.PCANTP_HANDLE_PCIBUS2,
    uds_parameter.PUDS_PARAMETER_DEBUG, ref buffer, sizeof(UInt32));
if (!UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Failed to set value", "Error");
else
    MessageBox.Show("Value changed successfully ", "Success");
```

C++/CLR

```
uds_status result;
UInt32 buffer;

// Enable error messages
buffer = UDSApi::PUDS_DEBUG_LVL_ERROR;
result = UDSApi::SetValue_2013(PCANTP_HANDLE_PCIBUS2, PUDS_PARAMETER_DEBUG, buffer,
    sizeof(UInt32));
if (!UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Failed to set value", "Error");
else
    MessageBox::Show("Value changed successfully ", "Success");
```

Visual Basic

```
Dim result As uds_status
Dim buffer As UInt32

' Enable error messages
buffer = UDSApi.PUDS_DEBUG_LVL_ERROR
result = UDSApi.SetValue_2013(cantp_handle.PCANTP_HANDLE_PCIBUS2,
    uds_parameter.PUDS_PARAMETER_DEBUG, buffer, CType(Len(buffer), UInt32))
If Not UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Failed to set value", "Error")
Else
    MessageBox.Show("Value changed successfully ", "Success")
End If
```

Pascal OO

```
var
    result: uds_status;
    buffer: UInt32;
begin

    // Enable error messages
    buffer := TUDSApi.PUDS_DEBUG_LVL_ERROR;
    result := TUDSApi.SetValue_2013(cantp_handle.PCANTP_HANDLE_PCIBUS2,
        uds_parameter.PUDS_PARAMETER_DEBUG, PLongWord(@buffer),
        UInt32(sizeof(buffer)));
    if not TUDSApi.StatusIsOk_2013(result) then
    begin
        MessageBox(0, 'Failed to set value', 'Error', MB_OK);
    end
    else
    begin
        MessageBox(0, 'Value changed successfully', 'Success', MB_OK);
    end;
end;
```

See also: `uds_parameter` on page 41, Detailed Parameters Characteristics on page 45, `GetValue_2013` on page 178.

Plain function version: `UDS_SetValue_2013` on page 629.

3.7.8 setValue_2013(cantp_handle, uds_parameter, string, UInt32)

Sets a configuration or information string value within a PUDS channel.

Syntax

Pascal OO

```
class function SetValue_2013(
    channel: cantp_handle;
    parameter: uds_parameter;
    buffer: PAnsiChar;
    buffer_size: UInt32
): uds_status; overload;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SetValue_2013")]
public static extern uds_status SetValue_2013(
    [MarshalAs(UnmanagedType.U4)]
```

```

cantp_handle channel,
[MarshalAs(UnmanagedType.U4)]
uds_parameter parameter,
[MarshalAs(UnmanagedType.LPStr, SizeParamIndex = 3)]
String buffer,
UInt32 buffer_size);

```

C++ / CLR

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SetValue_2013")]
static uds_status SetValue_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    [MarshalAs(UnmanagedType.U4)]
    uds_parameter parameter,
    [MarshalAs(UnmanagedType.LPStr, SizeParamIndex = 3)]
    String ^buffer,
    UInt32 buffer_size);

```

Visual Basic

```

<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SetValue_2013")>
Public Shared Function SetValue_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    <MarshalAs(UnmanagedType.U4)>
    ByVal parameter As uds_parameter,
    <MarshalAs(UnmanagedType.LPStr, SizeParamIndex:=3)>
    ByVal buffer As String,
    ByVal buffer_size As UInt32) As uds_status
End Function

```

Parameters

| Parameter | Description |
|-------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| parameter | The code of the value to be set (see uds_parameter on page 41). |
| buffer | The buffer containing the string value to be set. |
| buffer_size | The length in bytes of the given buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | Indicates that the parameters passed to the method are invalid. Check the value of the buffer and assert it is compatible with the buffer length. |

Remarks

This override is only defined for users who wishes to configure PCAN-Basic API through the PCAN-UDS 2.x API.

See also: `GetValue_2013` on page 178, `uds_parameter` on page 41, Detailed Parameters Characteristics on page 45.

Plain function version: `UDS_SetValue_2013` on page 629.

3.7.9 SetValue_2013(cantp_handle, uds_parameter, Byte[], UInt32)

Sets a configuration or information value as a byte array within a PUDS channel.

Syntax

Pascal OO

```
class function SetValue_2013(
  channel: cantp_handle;
  parameter: uds_parameter;
  buffer: PByte;
  buffer_size: UInt32
): uds_status; overload;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SetValue_2013")]
public static extern uds_status SetValue_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    [MarshalAs(UnmanagedType.U4)]
    uds_parameter parameter,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 3)]
    Byte[] buffer,
    UInt32 buffer_size);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SetValue_2013")]
static uds_status SetValue_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    [MarshalAs(UnmanagedType::U4)]
    uds_parameter parameter,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 3)]
    array<Byte> ^buffer,
    UInt32 buffer_size);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SetValue_2013")>
Public Shared Function SetValue_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    <MarshalAs(UnmanagedType.U4)>
    ByVal parameter As uds_parameter,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=3)>
    ByVal buffer As Byte(),
    ByVal buffer_size As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| parameter | The code of the value to be set (see uds_parameter on page 41). |
| buffer | The buffer containing the array value to be set. |
| buffer_size | The length in bytes of the given buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | Indicates that the parameters passed to the method are invalid. Check the value of the buffer and assert it is compatible with the buffer length. |

Remarks

Use the method `SetValue_2013` to set configuration information or environment values of a PUDS channel.

Note: That any calls with non PCAN-UDS 2.x API parameters (i.e. `uds_parameter`) will be forwarded to PCAN-ISO-TP 3.x API or PCAN-Basic API.

More information about the parameters and values that can be set can be found in Detailed Parameters Characteristics on page 45.

Example

The following example shows the use of the method `SetValue_2013` on the channel `PCANTP_HANDLE_USBBUS1` to set an unlimited block size.

C#

```
uds_status result;

// Define unlimited blocksize
result = UDSApi.SetValue_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
uds_parameter.PUDS_PARAMETER_BLOCK_SIZE, new byte[] { 0 }, 1);
if (!UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Failed to set value");
else
    MessageBox.Show("Value changed successfully ");
```

C++/CLR

```
uds_status result;

// Define unlimited blocksize
result = UDSApi::SetValue_2013(PCANTP_HANDLE_USBBUS1, PUDS_PARAMETER_BLOCK_SIZE, gcnew
    array<Byte> { 0 }, 1);
if (!UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Failed to set value");
else
    MessageBox::Show("Value changed successfully ");
```

Visual Basic

```
Dim result As uds_status
Dim buffer_array(2) As Byte

' Define unlimited blocksize
buffer_array(0) = 0
result = UDSApi.SetValue_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    uds_parameter.PUDS_PARAMETER_BLOCK_SIZE, buffer_array, 1)
If Not UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Failed to set value")
Else
    MessageBox.Show("Value changed successfully ")
```

End If

Pascal OO

```

var
  result: uds_status;
  buffer_array: array [0 .. 0] of Byte;
begin

  // Define unlimited blocksize
  buffer_array[0] := 0;
  result := TUDSApi.SetValue_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    uds_parameter.PUDS_PARAMETER_BLOCK_SIZE, PByte(@buffer_array), 1);
  if not TUDSApi.StatusIsOk_2013(result) then
  begin
    MessageBox(0, 'Failed to set value', 'Error', MB_OK);
  end
  else
  begin
    MessageBox(0, 'Value changed successfully', 'Success', MB_OK);
  end;
end;

```

See also: `uds_parameter` on page 41, Detailed Parameters Characteristics on page 45, `GetValue_2013` on page 178.

Plain function version: `UDS_SetValue_2013` on page 629.

3.7.10 SetValue_2013(cantp_handle, uds_parameter, IntPtr, UInt32)

Sets a configuration or information value from pointer within a PUDS channel.

Syntax**Pascal OO**

```

class function SetValue_2013(
  channel: cantp_handle;
  parameter: uds_parameter;
  buffer: Pointer;
  buffer_size: UInt32
): uds_status; overload;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SetValue_2013")]
public static extern uds_status SetValue_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    [MarshalAs(UnmanagedType.U4)]
    uds_parameter parameter,
    IntPtr buffer,
    UInt32 buffer_size);

```

C++ / CLR

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SetValue_2013")]
static uds_status SetValue_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    [MarshalAs(UnmanagedType::U4)]

```



```
uds_parameter parameter,
IntPtr buffer,
UInt32 buffer_size);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SetValue_2013")>
Public Shared Function SetValue_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    <MarshalAs(UnmanagedType.U4)>
    ByVal parameter As uds_parameter,
    ByVal buffer As IntPtr,
    ByVal buffer_size As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| parameter | The code of the value to be set (see uds_parameter on page 41). |
| buffer | Pointer on the value to be set. |
| buffer_size | The length in bytes of the given buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---------------------------------|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | Indicates that the parameters passed to the method are invalid. Check the value of the buffer and assert it is compatible with the buffer length. |

Remarks

Use the method `SetValue_2013` to set configuration information or environment values of a PUDS channel.

Note: That any calls with non PCAN-UDS 2.x API parameters (i.e. `uds_parameter`) will be forwarded to PCAN-ISO-TP 3.x API or PCAN-Basic API.

More information about the parameters and values that can be set can be found in Detailed Parameters Characteristics on page 45.

Example

The following example shows the use of the method `SetValue_2013` on the channel `PCANTP_HANDLE_USBBUS1` to change the current UDS Session Information.

Note: This only affects the API client side ONLY, no communication with any ECUs is made. If a user wants to disable the automatic transmission of TesterPresent requests that keeps alive a non-default diagnostic session, he/she should set the session type to the default diagnostic session (`uds_svc_param_dsc.PUDS_SVC_PARAM_DSC_DS`). It is assumed that the channel was already initialized, and a session exists.

C#

```
uds_sessioninfo new_sessioninfo;
new_sessioninfo.can_msg_type = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
new_sessioninfo.session_type = (Byte)UDSApi.uds_svc_param_dsc.PUDS_SVC_PARAM_DSC_DS;
```

```

new_sessioninfo.timeout_p2can_server_max = 1;
new_sessioninfo.timeout_enhanced_p2can_server_max = 2;
new_sessioninfo.s3_client_ms = 8;
new_sessioninfo.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
new_sessioninfo.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
new_sessioninfo.nai.source_addr =
    (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
new_sessioninfo.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
new_sessioninfo.nai.extension_addr = 0;

// Get pointer from structure
int session_size = Marshal.SizeOf(new_sessioninfo);
IntPtr session_ptr = Marshal.AllocHGlobal(session_size);
Marshal.StructureToPtr(new_sessioninfo, session_ptr, true);

// Set new session info
uds_status result = UDSApi.SetValue_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    uds_parameter.PUDS_PARAMETER_SESSION_INFO, session_ptr, (UInt32)session_size);
if (!UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Failed to set value");
else
    MessageBox.Show("Value changed successfully ");

// free resource
Marshal.FreeHGlobal(session_ptr);

```

C++/CLR

```

uds_sessioninfo new_sessioninfo;
new_sessioninfo.can_msg_type = PCANTP_CAN_MSGTYPE_STANDARD;
new_sessioninfo.session_type = (Byte)UDSApi::uds_svc_param_dsc::PUDS_SVC_PARAM_DSC_DS;
new_sessioninfo.timeout_p2can_server_max = 1;
new_sessioninfo.timeout_enhanced_p2can_server_max = 2;
new_sessioninfo.s3_client_ms = 8;
new_sessioninfo.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
new_sessioninfo.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
new_sessioninfo.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
new_sessioninfo.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
new_sessioninfo.nai.extension_addr = 0;

// Get pointer from structure
int session_size = Marshal::SizeOf(new_sessioninfo);
IntPtr session_ptr = Marshal::AllocHGlobal(session_size);
Marshal::StructureToPtr(new_sessioninfo, session_ptr, true);

// Set new session info
uds_status result = UDSApi::SetValue_2013(PCANTP_HANDLE_USBBUS1, PUDS_PARAMETER_SESSION_INFO,
    session_ptr, (UInt32)session_size);
if (!UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Failed to set value");
else
    MessageBox::Show("Value changed successfully ");

// free resource
Marshal::FreeHGlobal(session_ptr);

```

Visual Basic

```

Dim new_sessioninfo As uds_sessioninfo
new_sessioninfo.can_msg_type = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
new_sessioninfo.session_type = UDSApi.uds_svc_param_dsc.PUDS_SVC_PARAM_DSC_DS
new_sessioninfo.timeout_p2can_server_max = 1

```

```

new_sessioninfo.timeout_enhanced_p2can_server_max = 2
new_sessioninfo.s3_client_ms = 8
new_sessioninfo.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
new_sessioninfo.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
new_sessioninfo.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
new_sessioninfo.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
new_sessioninfo.nai.extension_addr = 0

' Get pointer from structure
Dim session_size As Integer
Dim session_ptr As IntPtr
session_size = Marshal.SizeOf(new_sessioninfo)
session_ptr = Marshal.AllocHGlobal(session_size)
Marshal.StructureToPtr(new_sessioninfo, session_ptr, True)

' Set New session info
Dim result As uds_status
result = UDSApi.SetValue_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    uds_parameter.PUDS_PARAMETER_SESSION_INFO, session_ptr, CType(session_size, UInt32))
If Not UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Failed to set value")
Else
    MessageBox.Show("Value changed successfully ")
End If

' free ressource
Marshal.FreeHGlobal(session_ptr)

```

Pascal OO

```

var
    result: uds_status;
    new_sessioninfo: uds_sessioninfo;
begin
    new_sessioninfo.can_msg_type := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    new_sessioninfo.session_type := Byte(uds_svc_param_dsc.PUDS_SVC_PARAM_DSC_DS);
    new_sessioninfo.timeout_p2can_server_max := 1;
    new_sessioninfo.timeout_enhanced_p2can_server_max := 2;
    new_sessioninfo.s3_client_ms := 8;
    new_sessioninfo.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    new_sessioninfo.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    new_sessioninfo.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    new_sessioninfo.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    new_sessioninfo.nai.extension_addr := 0;

    // Set new session info
    result := TUDSApi.SetValue_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        uds_parameter.PUDS_PARAMETER_SESSION_INFO, Pointer(@new_sessioninfo),
        UInt32(sizeof(new_sessioninfo)));
    if not TUDSApi.StatusIsOk_2013(result) then
    begin
        MessageBox(0, 'Failed to set value', 'Error', MB_OK);
    end
    else
    begin
        MessageBox(0, 'Value changed successfully', 'Success', MB_OK);
    end;
end;

```

See also: [uds_parameter](#) on page 41, Detailed Parameters Characteristics on page 45, [GetValue_2013](#) on page 178.

Plain function version: [UDS_SetValue_2013](#) on page 629.

3.7.11 AddMapping_2013

Adds a user-defined mapping between a CAN identifier and a network address information. Defining a mapping enables PCANTP communication with 11BITS CAN identifier or with opened Addressing Formats (like [PCANTP_ISOTP_FORMAT_NORMAL](#) or [PCANTP_ISOTP_FORMAT_EXTENDED](#)).

Syntax

Pascal OO

```
class function AddMapping_2013(
  channel: cantp_handle;
  mapping: Puds_mapping
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_AddMapping_2013")]
public static extern uds_status AddMapping_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    ref uds_mapping mapping);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_AddMapping_2013")]
static uds_status AddMapping_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_mapping %mapping);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_AddMapping_2013")>
Public Shared Function AddMapping_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByRef mapping As uds_mapping) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| mapping | Mapping to be added (see uds_mapping on page 25). |

Remark

By default, some mappings are initialized in the PCAN-UDS 2.x API (see UDS and ISO-TP Network Addressing Information on page 770).

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

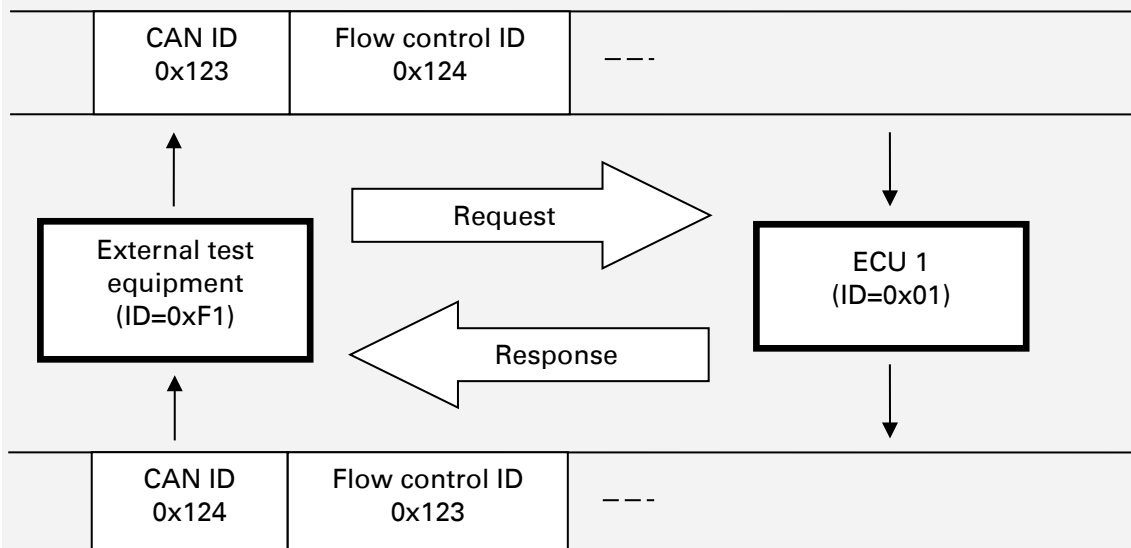
| | |
|--|--|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | Indicates that the given mapping is null. |
| <code>PUDS_STATUS_ALREADY_INITIALIZED</code> | A mapping with the same CAN identifier already exists. |
| <code>PUDS_STATUS_MAPPING_INVALID</code> | Mapping is not valid regarding UDS standard. |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory to define mapping. |

Example

The following example shows the use of the method `AddMapping_2013` on the USB channel 1. It creates mappings to communicate with custom CAN identifiers between test equipment and ECU 1 in ISO15765-2 11bits normal addressing on initialized USB channel 1:

Note: It is assumed that the channel was already initialized.

Set CAN identifier = 0x123 and flow control id = 0x124 for request and set CAN identifier = 0x124 and flow control id = 0x123 for response. Test equipment address corresponds to 0xF1 and ECU 1 address corresponds to 0x01. Here is a small scheme of the mapping:



C#

```
uds_mapping request_mapping = new uds_mapping();
request_mapping.can_id = 0x123;
request_mapping.can_id_flow_ctrl = 0x124;
request_mapping.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
request_mapping.nai.extension_addr = 0;
request_mapping.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
request_mapping.can_tx_dlc = 8;
request_mapping.nai.source_addr =
    (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
request_mapping.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
request_mapping.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;

uds_mapping response_mapping;
response_mapping = request_mapping;
response_mapping.can_id = request_mapping.can_id_flow_ctrl;
```

```

response_mapping.can_id_flow_ctrl = request_mapping.can_id;
response_mapping.nai.source_addr = request_mapping.nai.target_addr;
response_mapping.nai.target_addr = request_mapping.nai.source_addr;

uds_status status;
status = UDSApi.AddMapping_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request_mapping);
if (UDSApi.StatusIsOk_2013(status))
    MessageBox.Show("Add request mapping", "Success");
else
    MessageBox.Show("Failed to add request mapping", "Error");
status = UDSApi.AddMapping_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref response_mapping);
if (UDSApi.StatusIsOk_2013(status))
    MessageBox.Show("Add response mapping", "Success");
else
    MessageBox.Show("Failed to add response mapping", "Error");

```

C++ / CLR

```

uds_mapping request_mapping = {};
request_mapping.can_id = 0x123;
request_mapping.can_id_flow_ctrl = 0x124;
request_mapping.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
request_mapping.nai.extension_addr = 0;
request_mapping.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
request_mapping.can_tx_dlc = 8;
request_mapping.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
request_mapping.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
request_mapping.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;

uds_mapping response_mapping;
response_mapping = request_mapping;
response_mapping.can_id = request_mapping.can_id_flow_ctrl;
response_mapping.can_id_flow_ctrl = request_mapping.can_id;
response_mapping.nai.source_addr = request_mapping.nai.target_addr;
response_mapping.nai.target_addr = request_mapping.nai.source_addr;

uds_status status;
status = UDSApi::AddMapping_2013(PCANTP_HANDLE_USBBUS1, request_mapping);
if (UDSApi::StatusIsOk_2013(status))
    MessageBox::Show("Add request mapping", "Success");
else
    MessageBox::Show("Failed to add request mapping", "Error");

status = UDSApi::AddMapping_2013(PCANTP_HANDLE_USBBUS1, response_mapping);
if (UDSApi::StatusIsOk_2013(status))
    MessageBox::Show("Add response mapping", "Success");
else
    MessageBox::Show("Failed to add response mapping", "Error");

```

Visual Basic

```

Dim request_mapping As uds_mapping
request_mapping.can_id = &H123
request_mapping.can_id_flow_ctrl = &H124
request_mapping.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
request_mapping.nai.extension_addr = 0
request_mapping.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
request_mapping.can_tx_dlc = 8
request_mapping.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
request_mapping.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
request_mapping.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL

```

```

Dim response_mapping As uds_mapping
response_mapping = request_mapping
response_mapping.can_id = request_mapping.can_id_flow_ctrl
response_mapping.can_id_flow_ctrl = request_mapping.can_id
response_mapping.nai.source_addr = request_mapping.nai.target_addr
response_mapping.nai.target_addr = request_mapping.nai.source_addr

Dim status As uds_status
status = UDSApi.AddMapping_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request_mapping)
If UDSApi.StatusIsOk_2013(status) Then
    MessageBox.Show("Add request mapping", "Success")
Else
    MessageBox.Show("Failed to add request mapping", "Error")
End If
status = UDSApi.AddMapping_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, response_mapping)
If UDSApi.StatusIsOk_2013(status) Then
    MessageBox.Show("Add response mapping", "Success")
Else
    MessageBox.Show("Failed to add response mapping", "Error")
End If

```

Pascal OO

```

var
    request_mapping: uds_mapping;
    response_mapping: uds_mapping;
    status: uds_status;
begin
    request_mapping.can_id := $123;
    request_mapping.can_id_flow_ctrl := $124;
    request_mapping.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    request_mapping.nai.extension_addr := 0;
    request_mapping.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    request_mapping.can_tx_dlc := 8;
    request_mapping.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    request_mapping.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    request_mapping.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;

    response_mapping := request_mapping;
    response_mapping.can_id := request_mapping.can_id_flow_ctrl;
    response_mapping.can_id_flow_ctrl := request_mapping.can_id;
    response_mapping.nai.source_addr := request_mapping.nai.target_addr;
    response_mapping.nai.target_addr := request_mapping.nai.source_addr;

    status := TUDSApi.AddMapping_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request_mapping);
    if TUDSApi.StatusIsOk_2013(status) then
        begin
            MessageBox(0, 'Add request mapping', 'Success', MB_OK);
        end
    else
        begin
            MessageBox(0, 'Failed to add request mapping', 'Error', MB_OK);
        end
    end;
    status := TUDSApi.AddMapping_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @response_mapping);
    if TUDSApi.StatusIsOk_2013(status) then
        begin

```

```

    MessageBox(0, 'Add response mapping', 'Success', MB_OK);
end
else
begin
    MessageBox(0, 'Failed to add response mapping', 'Error', MB_OK);
end;
end;

```

See also: [RemoveMapping_2013](#) on page 168, [RemoveMappingByCanId_2013](#) on page 171.

Plain function version: [UDS_AddMapping_2013](#) on page 631.

3.7.12 RemoveMapping_2013

Removes a user defined PUDS mapping.

Syntax

Pascal OO

```

class function RemoveMapping_2013(
    channel: cantp_handle;
    mapping: uds_mapping
): uds_status;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_RemoveMapping_2013")]
public static extern uds_status RemoveMapping_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_mapping mapping);

```

C++ / CLR

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_RemoveMapping_2013")]
static uds_status RemoveMapping_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_mapping mapping);

```

Visual Basic

```

<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_RemoveMapping_2013")>
Public Shared Function RemoveMapping_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal mapping As uds_mapping) As uds_status
End Function

```

Parameters

| Parameter | Description |
|-----------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| mapping | The mapping to remove (see uds_mapping on page 25). |


Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|--|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The mapping is not a valid mapping. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping is not in the mapping list. |

Example

The following example shows the use of the method `RemoveMapping_2013` on the USB channel 1. It creates a mapping then removes it.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_mapping request_mapping = new uds_mapping();
request_mapping.can_id = 0x123;
request_mapping.can_id_flow_ctrl = 0x124;
request_mapping.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
request_mapping.nai.extension_addr = 0;
request_mapping.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
request_mapping.can_tx_dlc = 8;
request_mapping.nai.source_addr =
    (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
request_mapping.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
request_mapping.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;

uds_status status;
status = UDSApi.AddMapping_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request_mapping);
if (UDSApi.StatusIsOk_2013(status))
{
    // Remove the request mapping
    status = UDSApi.RemoveMapping_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request_mapping);
    if (UDSApi.StatusIsOk_2013(status))
    {
        MessageBox.Show("Remove request mapping", "Success");
    }
    else
    {
        MessageBox.Show("Failed to remove request mapping", "Error");
    }
}
else
{
    MessageBox.Show("Failed to add request mapping", "Error");
}
```

C++ / CLR

```
uds_mapping request_mapping = {};
request_mapping.can_id = 0x123;
request_mapping.can_id_flow_ctrl = 0x124;
request_mapping.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
request_mapping.nai.extension_addr = 0;
request_mapping.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
request_mapping.can_tx_dlc = 8;
request_mapping.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
```

```

request_mapping.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
request_mapping.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;

uds_status status;
status = UDSApi::AddMapping_2013(PCANTP_HANDLE_USBBUS1, request_mapping);
if (UDSApi::StatusIsOk_2013(status))
{
    // Remove the request mapping
    status = UDSApi::RemoveMapping_2013(PCANTP_HANDLE_USBBUS1, request_mapping);
    if (UDSApi::StatusIsOk_2013(status))
    {
        MessageBox::Show("Remove request mapping", "Success");
    }
    else
    {
        MessageBox::Show("Failed to remove request mapping", "Error");
    }
}
else
{
    MessageBox::Show("Failed to add request mapping", "Error");
}

```

Visual Basic

```

Dim request_mapping As uds_mapping
request_mapping.can_id = &H123
request_mapping.can_id_flow_ctrl = &H124
request_mapping.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
request_mapping.nai.extension_addr = 0
request_mapping.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
request_mapping.can_tx_dlc = 8
request_mapping.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
request_mapping.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
request_mapping.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL

Dim status As uds_status
status = UDSApi.AddMapping_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request_mapping)
If (UDSApi.StatusIsOk_2013(status)) Then

    ' Remove the request mapping
    status = UDSApi.RemoveMapping_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request_mapping)
    If (UDSApi.StatusIsOk_2013(status)) Then
        MessageBox.Show("Remove request mapping", "Success")
    Else
        MessageBox.Show("Failed to remove request mapping", "Error")
    End If
Else
    MessageBox.Show("Failed to add request mapping", "Error")
End If

```

Pascal OO

```

var
    request_mapping: uds_mapping;
    status: uds_status;
begin
    request_mapping.can_id := $123;
    request_mapping.can_id_flow_ctrl := $124;
    request_mapping.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    request_mapping.nai.extension_addr := 0;

```

```

request_mapping.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
request_mapping.can_tx_dlc := 8;
request_mapping.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
request_mapping.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
request_mapping.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;

status := TUDSApi.AddMapping_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    @request_mapping);
if TUDSApi.StatusIsOk_2013(status) then
begin

    // Remove the request mapping
    status := TUDSApi.RemoveMapping_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        request_mapping);
    if TUDSApi.StatusIsOk_2013(status) then
    begin
        MessageBox(0, 'Remove request mapping', 'Success', MB_OK);
    end
    else
    begin
        MessageBox(0, 'Failed to remove request mapping', 'Error', MB_OK);
    end;
end
else
begin
    MessageBox(0, 'Failed to add request mapping', 'Error', MB_OK);
end;
end;

```

See also: [uds_mapping](#) on page 25, [RemoveMappingByCanId_2013](#) on page 171, [AddMapping_2013](#) on page 164.

Plain function version: [UDS_RemoveMapping_2013](#) on page 634.

3.7.13 RemoveMappingByCanId_2013

Removes all user defined PUDS mappings corresponding to a CAN identifier.

Syntax

Pascal OO

```

class function RemoveMappingByCanId_2013(
    channel: cantp_handle;
    can_id: UInt32
): uds_status;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_RemoveMappingByCanId_2013")]
public static extern uds_status RemoveMappingByCanId_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    UInt32 can_id);

```

C++ / CLR

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_RemoveMappingByCanId_2013")]
static uds_status RemoveMappingByCanId_2013(

```

```
[MarshalAs(UnmanagedType::U4)]
cantp_handle channel,
UInt32 can_id);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_RemoveMappingByCanId_2013")>
Public Shared Function RemoveMappingByCanId_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal can_id As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| can_id | The mapped CAN identifier to search for that identifies the mappings to remove (see predefined uds_can_id values on page 58). |


Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|-------------------------------------|--|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | The CAN identifier to remove is not specified in a mapping. |

Example

The following example shows the use of the method `RemoveMappingByCanId_2013` on the USB channel 1. It creates a mapping then removes it using its CAN identifier on initialized USB channel 1:

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_mapping request_mapping = new uds_mapping();
request_mapping.can_id = 0x123;
request_mapping.can_id_flow_ctrl = 0x124;
request_mapping.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
request_mapping.nai.extension_addr = 0;
request_mapping.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
request_mapping.can_tx_dlc = 8;
request_mapping.nai.source_addr =
    (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
request_mapping.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
request_mapping.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;

uds_status status;
status = UDSApi.AddMapping_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request_mapping);
if (UDSApi.StatusIsOk_2013(status))
{
    // Remove mapping using its can identifier
    status = UDSApi.RemoveMappingByCanId_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, 0x123);
    if (UDSApi.StatusIsOk_2013(status))
    {
        MessageBox.Show("Remove request mapping using its can identifier", "Success");
    }
}
```

```

    else
    {
        MessageBox.Show("Failed to remove request mapping", "Error");
    }
}
else
{
    MessageBox.Show("Failed to add request mapping", "Error");
}

```

C++ / CLR

```

uds_mapping request_mapping = {};
request_mapping.can_id = 0x123;
request_mapping.can_id_flow_ctrl = 0x124;
request_mapping.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
request_mapping.nai.extension_addr = 0;
request_mapping.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
request_mapping.can_tx_dlc = 8;
request_mapping.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
request_mapping.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
request_mapping.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;

uds_status status;
status = UDSApi::AddMapping_2013(PCANTP_HANDLE_USBBUS1, request_mapping);
if (UDSApi::StatusIsOk_2013(status))
{
    // Remove mapping using its can identifier
    status = UDSApi::RemoveMappingByCanId_2013(PCANTP_HANDLE_USBBUS1, 0x123);
    if (UDSApi::StatusIsOk_2013(status))
    {
        MessageBox::Show("Remove request mapping using its can identifier", "Success");
    }
    else
    {
        MessageBox::Show("Failed to remove request mapping", "Error");
    }
}
else
{
    MessageBox::Show("Failed to add request mapping", "Error");
}

```

Visual Basic

```

Dim request_mapping As uds_mapping
request_mapping.can_id = &H123
request_mapping.can_id_flow_ctrl = &H124
request_mapping.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
request_mapping.nai.extension_addr = 0
request_mapping.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
request_mapping.can_tx_dlc = 8
request_mapping.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
request_mapping.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
request_mapping.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL

Dim status As uds_status
status = UDSApi.AddMapping_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request_mapping)
If (UDSApi.StatusIsOk_2013(status)) Then
    ' Remove mapping using its can identifier

```

```

status = UDSApi.RemoveMappingByCanId_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, &H123)
If (UDSApI.StatusIsOk_2013(status)) Then
    MessageBox.Show("Remove request mapping using its can identifier", "Success")
Else
    MessageBox.Show("Failed to remove request mapping", "Error")
End If
Else
    MessageBox.Show("Failed to add request mapping", "Error")
End If

```

Pascal OO

```

var
    request_mapping: uds_mapping;
    status: uds_status;
begin
    request_mapping.can_id := $123;
    request_mapping.can_id_flow_ctrl := $124;
    request_mapping.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    request_mapping.nai.extension_addr := 0;
    request_mapping.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    request_mapping.can_tx_dlc := 8;
    request_mapping.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    request_mapping.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    request_mapping.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;

    status := TUDSApI.AddMapping_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request_mapping);
    if TUDSApI.StatusIsOk_2013(status) then
        begin
            // Remove mapping using its can identifier
            status := TUDSApI.RemoveMappingByCanId_2013
                (cantp_handle.PCANTP_HANDLE_USBBUS1, $123);
            if TUDSApI.StatusIsOk_2013(status) then
                begin
                    MessageBox(0, 'Remove request mapping using its can identifier',
                        'Success', MB_OK);
                end
            else
                begin
                    MessageBox(0, 'Failed to remove request mapping', 'Error', MB_OK);
                end;
            end
        else
            begin
                MessageBox(0, 'Failed to add request mapping', 'Error', MB_OK);
            end;
        end;
    end;
end;

```

See also: [uds_mapping](#) on page 25, [RemoveMapping_2013](#) on page 168, [AddMapping_2013](#) on page 164.

Plain function version: [UDS_RemoveMappingByCanId_2013](#) on page 633.

3.7.14 AddCanIdFilter_2013

Adds an entry to the CAN identifier white-list filtering.

Syntax

Pascal OO

```
class function AddCanIdFilter_2013(
    channel: cantp_handle;
    can_id: UInt32
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_AddCanIdFilter_2013")]
public static extern uds_status AddCanIdFilter_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    UInt32 can_id);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_AddCanIdFilter_2013")]
static uds_status AddCanIdFilter_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    UInt32 can_id);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_AddCanIdFilter_2013")>
Public Shared Function AddCanIdFilter_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal can_id As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| can_id | CAN identifier to add in the white-list (see predefined uds_can_id values on page 58). |


Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---------------------------------|--|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_ALREADY_INITIALIZED | The CAN identifier is already in the white list. |
| PUDS_STATUS_NO_MEMORY | Memory allocation error when adding the new element in the white list. |

Example

The following example shows the use of the method `AddCanIdFilter_2013` the channel `PCANTP_HANDLE_USBBUS1`. It adds a filter on 0xD1 CAN identifier.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
result = UDSApi.AddCanIdFilter_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, 0xD1);
if (!UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Error adding CAN ID filter.", "Error");
```

C++ / CLR

```
uds_status result;
result = UDSApi::AddCanIdFilter_2013(PCANTP_HANDLE_USBBUS1, 0xD1);
if (!UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Error adding CAN ID filter.", "Error");
```

Visual Basic

```
Dim result As uds_status
result = UDSApi.AddCanIdFilter_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, &HD1)
If Not UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Error adding CAN ID filter.", "Error")
End If
```

Pascal OO

```
var
    result: uds_status;
begin
    result := TUDSApi.AddCanIdFilter_2013
        (cantp_handle.PCANTP_HANDLE_USBBUS1, $D1);
    if not TUDSApi.StatusIsOk_2013(result) then
        begin
            MessageBox(0, 'Error adding CAN ID filter.', 'Error', MB_OK);
        end;
    end;
```

See also: [RemoveCanIdFilter_2013](#) on page 176, [uds_can_id](#) on page 58.

Plain function version: [UDS_AddCanIdFilter_2013](#) on page 635.

3.7.15 RemoveCanIdFilter_2013

Removes an entry from the CAN identifier white-list filtering.

Syntax

Pascal OO

```
class function RemoveCanIdFilter_2013(
    channel: cantp_handle;
    can_id: UInt32
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_RemoveCanIdFilter_2013")]
public static extern uds_status RemoveCanIdFilter_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    UInt32 can_id);
```


C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_RemoveCanIdFilter_2013")]
static uds_status RemoveCanIdFilter_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    UInt32 can_id);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_RemoveCanIdFilter_2013")>
Public Shared Function RemoveCanIdFilter_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal can_id As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| can_id | CAN identifier to remove (see predefined uds_can_id values on page 58). |


Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical error in case of failure is:

| | |
|-----------------------------|--|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. Or the CAN identifier is not in the white list. |
|-----------------------------|--|

Example

The following example shows the use of the method `RemoveCanIdFilter_2013` on the channel `PCANTP_HANDLE_USBBUS1`. It adds a filter on 0xD1 CAN identifier then removes it.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
result = UDSApi.AddCanIdFilter_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, 0xD1);
if (!UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Error adding CAN ID filter.", "Error");

// Remove previously added can identifier filter
result = UDSApi.RemoveCanIdFilter_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, 0xD1);
if (!UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Error removing CAN ID filter.", "Error");
```

C++ / CLR

```
uds_status result;
result = UDSApi::AddCanIdFilter_2013(PCANTP_HANDLE_USBBUS1, 0xD1);
if (!UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Error adding CAN ID filter.", "Error");

// Remove previously added can identifier filter
result = UDSApi::RemoveCanIdFilter_2013(PCANTP_HANDLE_USBBUS1, 0xD1);
if (!UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Error removing CAN ID filter.", "Error");
```

Visual Basic

```
Dim result As uds_status
result = UDSApi.AddCanIdFilter_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, &HD1)
If Not UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Error adding CAN ID filter.", "Error")
End If

' Remove previously added can identifier filter
result = UDSApi.RemoveCanIdFilter_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, &HD1)
If Not UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Error removing CAN ID filter.", "Error")
End If
```

Pascal OO

```
var
    result: uds_status;
begin
    result := TUDSApi.AddCanIdFilter_2013
        (cantp_handle.PCANTP_HANDLE_USBBUS1, $D1);
    if not TUDSApi.StatusIsOk_2013(result) then
        begin
            MessageBox(0, 'Error adding CAN ID filter.', 'Error', MB_OK);
        end;
    // Remove previously added can identifier filter
    result := TUDSApi.RemoveCanIdFilter_2013
        (cantp_handle.PCANTP_HANDLE_USBBUS1, $D1);
    if not TUDSApi.StatusIsOk_2013(result) then
        begin
            MessageBox(0, 'Error removing CAN ID filter.', 'Error', MB_OK);
        end;
    end;
```





See also: [AddCanIdFilter_2013](#) on page 175.

Plain function version: [UDS_RemoveCanIdFilter_2013](#) on page 636.

3.7.16 GetValue_2013

Retrieves information from a PUDS channel.

overloads

| | Method | Description |
|---|--|---|
|  | GetValue_2013(cantp_handle, uds_parameter, UInt32, UInt32) | Retrieves information from a PUDS channel in numeric form. |
|  | GetValue_2013(cantp_handle, uds_parameter, String, UInt32) | Retrieves information from a PUDS channel in text form. |
|  | GetValue_2013(cantp_handle, uds_parameter, Byte[], UInt32) | Retrieves information from a PUDS channel in byte array form. |
|  | GetValue_2013(cantp_handle, uds_parameter, IntPtr, UInt32) | Retrieves information from a PUDS channel in pointer form. |

Plain function version: [UDS_GetValue_2013](#) on page 630.

3.7.17 GetValue_2013(cantp_handle, uds_parameter, String, UInt32)

Retrieves information from a PUDS channel in text form.

Syntax

Pascal OO

```
class function GetValue_2013(
  channel: cantp_handle;
  parameter: uds_parameter;
  buffer: PAnsiChar;
  buffer_size: UInt32
): uds_status; overload;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_GetValue_2013")]
public static extern uds_status GetValue_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    [MarshalAs(UnmanagedType.U4)]
    uds_parameter parameter,
    StringBuilder buffer,
    UInt32 buffer_size);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_GetValue_2013")]
static uds_status GetValue_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    [MarshalAs(UnmanagedType::U4)]
    uds_parameter parameter,
    StringBuilder ^buffer,
    UInt32 buffer_size);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_GetValue_2013")>
Public Shared Function GetValue_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    <MarshalAs(UnmanagedType.U4)>
    ByVal parameter As uds_parameter,
    ByVal buffer As StringBuilder,
    ByVal buffer_size As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| parameter | The code of the value to be set (see uds_parameter on page 41). |
| buffer | The buffer to return the required string value. |
| buffer_size | The length in bytes of the given buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | Indicates that the parameters passed to the method are invalid. Check the value of the buffer and assert it is compatible with the buffer length. |

Example

The following example shows the use of the method `GetValue_2013` to retrieve the version of the PCAN-UDS API. Depending on the result, a message will be shown to the user.

C#

```
uds_status result;
StringBuilder buffer;

// Get API version
buffer = new StringBuilder(255);
result = UDSApi.GetValue_2013(cantp_handle.PCANTP_HANDLE_NONEBUS,
    uds_parameter.PUDS_PARAMETER_API_VERSION, buffer, 255);
if (!UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Failed to get value");
else
    MessageBox.Show(buffer.ToString());
```

C++ / CLR

```
uds_status result;
StringBuilder ^buffer;

// Get API version
buffer = gcnew StringBuilder(255);
result = UDSApi::GetValue_2013(PCANTP_HANDLE_NONEBUS, PUDS_PARAMETER_API_VERSION, buffer,
    255);
if (!UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Failed to get value");
else
    MessageBox::Show(buffer->ToString());
```

Visual Basic

```
Dim result As uds_status
Dim buffer As StringBuilder

' Get API version
buffer = New StringBuilder(255)
result = UDSApi.GetValue_2013(cantp_handle.PCANTP_HANDLE_NONEBUS,
    uds_parameter.PUDS_PARAMETER_API_VERSION, buffer, 255)
If Not UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Failed to get value")
Else
    MessageBox.Show(buffer.ToString())
End If
```

Pascal OO

```
var
  result: uds_status;
  buffer: array [0 .. 256] of ansichar;
begin

  // Get API version
  result := TUDSApi.GetValue_2013(cantp_handle.PCANTP_HANDLE_NONEBUS,
    uds_parameter.PUDS_PARAMETER_API_VERSION, PAnsichar(@buffer), 255);
  if not TUDSApi.StatusIsOk_2013(result) then
  begin
    MessageBox(0, 'Failed to get value', 'Error', MB_OK);
  end
  else
  begin
    MessageBox(0, PWideChar(String(buffer)), 'Success', MB_OK);
  end;
end;
```

See also: `SetValue_2013` on page 153, `uds_parameter` on page 41, Detailed Parameters Characteristics on page 45.

Plain function version: `UDS_GetValue_2013` on page 630.

3.7.18 GetValue_2013(cantp_handle, uds_parameter, UInt32, UInt32)

Retrieves information from a PUDS channel in numeric form.

Syntax

Pascal OO

```
class function GetValue_2013(
  channel: cantp_handle;
  parameter: uds_parameter;
  buffer: PLongWord;
  buffer_size: UInt32
): uds_status; overload;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_GetValue_2013")]
public static extern uds_status GetValue_2013(
  [MarshalAs(UnmanagedType.U4)]
  cantp_handle channel,
  [MarshalAs(UnmanagedType.U4)]
  uds_parameter parameter,
  out UInt32 buffer,
  UInt32 buffer_size);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_GetValue_2013")]
static uds_status GetValue_2013(
  [MarshalAs(UnmanagedType::U4)]
  cantp_handle channel,
  [MarshalAs(UnmanagedType::U4)]
  uds_parameter parameter,
  UInt32 %buffer,
  UInt32 buffer_size);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_GetValue_2013")>
Public Shared Function GetValue_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    <MarshalAs(UnmanagedType.U4)>
    ByVal parameter As uds_parameter,
    ByRef buffer As UInt32,
    ByVal buffer_size As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| parameter | The code of the value to be set (see uds_parameter on page 41). |
| buffer | The buffer to return the required numeric value. |
| buffer_size | The length in bytes of the given buffer. |


Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | Indicates that the parameters passed to the method are invalid. Check the value of the buffer and assert it is compatible with the buffer length. |

Example

The following example shows the use of the method `GetValue_2013` on the channel `PCANTP_HANDLE_USBBUS1` to retrieve the separation time value (STmin). Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
UInt32 buffer = 0;

// Get the value of the Separation Time (STmin) parameter
result = UDSApi.GetValue_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    uds_parameter.PUDS_PARAMETER_SEPARATION_TIME, out buffer, sizeof(UInt32));
if (!UDSApi.StatusIsOk_2013(result))
{
    MessageBox.Show("Failed to get value", "Error");
}
else
{
    MessageBox.Show(buffer.ToString(), "Success");
}
```

C++ / CLR

```
uds_status result;
UInt32 buffer = 0;

// Get the value of the Separation Time (STmin) parameter
```

```

result = UDSApi::GetValue_2013(PCANTP_HANDLE_USBBUS1, PUDS_PARAMETER_SEPARATION_TIME, buffer,
    sizeof(UInt32));
if (!UDSApi::StatusIsOk_2013(result))
{
    MessageBox::Show("Failed to get value", "Error");
}
else
{
    MessageBox::Show(buffer.ToString(), "Success");
}

```

Visual Basic

```

Dim result As uds_status
Dim buffer As UInt32 = 0

' Get the value of the Separation Time (STmin) parameter
result = UDSApi.GetValue_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    uds_parameter.PUDS_PARAMETER_SEPARATION_TIME, buffer, CType(Len(buffer), UInt32))
If Not UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Failed to get value", "Error")
Else
    MessageBox.Show(buffer.ToString(), "Success")
End If

```

Pascal OO

```

var
    result: uds_status;
    buffer: UInt32;
begin
    buffer := 0;

    // Get the value of the Separation Time (STmin) parameter
    result := TUDSApi.GetValue_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        uds_parameter.PUDS_PARAMETER_SEPARATION_TIME, PLongWord(@buffer),
        sizeof(UInt32));
    if not TUDSApi.StatusIsOk_2013(result) then
    begin
        MessageBox(0, 'Failed to get value', 'Error', MB_OK);
    end
    else
    begin
        MessageBox(0, PWideChar(format('%d', [Integer(buffer)])),
            'Success', MB_OK);
    end;
end;

```

See also: [SetValue_2013](#) on page 153, [uds_parameter](#) on page 41, Detailed Parameters Characteristics on page 45.
Plain function version: [UDS_GetValue_2013](#) on page 630.

3.7.19 GetValue_2013(cantp_handle, uds_parameter, Byte[], UInt32)

Retrieves information from a PUDS channel in a byte array.

Syntax

Pascal OO

```
class function GetValue_2013(
  channel: cantp_handle;
  parameter: uds_parameter;
  buffer: PByte;
  buffer_size: UInt32
): uds_status; overload;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_GetValue_2013")]
public static extern uds_status GetValue_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    [MarshalAs(UnmanagedType.U4)]
    uds_parameter parameter,
    [MarshalAs(UnmanagedType.LPArray)]
    [Out] Byte[] buffer,
    UInt32 buffer_size);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_GetValue_2013")]
static uds_status GetValue_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    [MarshalAs(UnmanagedType::U4)]
    uds_parameter parameter,
    [MarshalAs(UnmanagedType::LPArray)]
    [Out] array<Byte> ^buffer,
    UInt32 buffer_size);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_GetValue_2013")>
Public Shared Function GetValue_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    <MarshalAs(UnmanagedType.U4)>
    ByVal parameter As uds_parameter,
    <MarshalAs(UnmanagedType.LPArray)>
    <Out> ByVal buffer As Byte(),
    ByVal buffer_size As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| parameter | The code of the value to be set (see uds_parameter on page 41). |
| buffer | The buffer containing the array value to retrieve. |
| buffer_size | The length in bytes of the given buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | Indicates that the parameters passed to the method are invalid. Check the value of the buffer and assert it is compatible with the buffer length. |

Example

The following example shows the use of the method `GetValue_2013` on the channel `PCANTP_HANDLE_USBBUS1` to retrieve the separation time value (STmin). Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

C#

```
uds_status result;
uint buffer_size = 1;
byte[] byte_buffer = new byte[buffer_size];

// Get the value of the ISO-TP Separation Time (STmin) parameter
result = UDSApi.GetValue_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    uds_parameter.PUDS_PARAMETER_SEPARATION_TIME, byte_buffer, sizeof(byte) * buffer_size);
if (!UDSApi.StatusIsOk_2013(result))
{
    MessageBox.Show("Failed to get value");
}
else
{
    MessageBox.Show(byte_buffer[0].ToString());
}
```

C++ / CLR

```
uds_status result;
UInt32 buffer_size = 1;
array<Byte>^ byte_buffer = gcnew array<Byte>(buffer_size);

// Get the value of the ISO-TP Separation Time (STmin) parameter
result = UDSApi::GetValue_2013(PCANTP_HANDLE_USBBUS1, PUDS_PARAMETER_SEPARATION_TIME,
    byte_buffer, buffer_size);
if (!UDSApi::StatusIsOk_2013(result))
{
    MessageBox::Show("Failed to get value");
}
else
{
    MessageBox::Show(byte_buffer[0].ToString());
}
```

Visual Basic

```
Dim result As uds_status
Dim byte_buffer(2) As Byte

' Get the value of the ISO-TP Separation Time (STmin) parameter
result = UDSApi.GetValue_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    uds_parameter.PUDS_PARAMETER_SEPARATION_TIME, byte_buffer, 1)
```

```
If Not UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Failed to get value")
Else
    MessageBox.Show(byte_buffer(0).ToString())
End If
```

Pascal OO

```
var
    result: uds_status;
    byte_buffer: array [0 .. 0] of Byte;
begin

    // Get the value of the ISO-TP Separation Time (STmin) parameter
    result := TUDSApi.GetValue_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        uds_parameter.PUDS_PARAMETER_SEPARATION_TIME, PByte(@byte_buffer), 1);
    if not TUDSApi.StatusIsOk_2013(result) then
    begin
        MessageBox(0, 'Failed to get value', 'Error', MB_OK);
    end
    else
    begin
        MessageBox(0, PWideChar(format('%d', [Integer(byte_buffer[0])])),
            'Success', MB_OK);
    end;
end;
```

See also: [SetValue_2013](#) on page 153, [uds_parameter](#) on page 41, Detailed Parameters Characteristics on page 45.

Plain function version: [UDS_GetValue_2013](#) on page 630.

3.7.20 GetValue_2013(cantp_handle, uds_parameter, IntPtr, UInt32)

Retrieves information from a PUDS channel through a pointer.

Syntax

Pascal OO

```
class function GetValue_2013(
    channel: cantp_handle;
    parameter: uds_parameter;
    buffer: Pointer;
    buffer_size: UInt32
): uds_status; overload;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_GetValue_2013")]
public static extern uds_status GetValue_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    [MarshalAs(UnmanagedType.U4)]
    uds_parameter parameter,
    IntPtr buffer,
    UInt32 buffer_size);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_GetValue_2013")]
static uds_status GetValue_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    [MarshalAs(UnmanagedType::U4)]
    uds_parameter parameter,
    IntPtr buffer,
    UInt32 buffer_size);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_GetValue_2013")>
Public Shared Function GetValue_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    <MarshalAs(UnmanagedType.U4)>
    ByVal parameter As uds_parameter,
    ByVal buffer As IntPtr,
    ByVal buffer_size As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| parameter | The code of the value to be set (see uds_parameter on page 41). |
| buffer | The pointer on the buffer containing the value to retrieve. |
| buffer_size | The length in bytes of the given buffer. |


Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---------------------------------|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | Indicates that the parameters passed to the method are invalid. Check the value of the buffer and assert it is compatible with the buffer length. |

Example

The following example shows the use of the method `GetValue_2013` on the channel `PCANTP_HANDLE_USBBUS1` to retrieve the UDS Session Information. A console message will be written with the information retrieved.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_sessioninfo session_info = new uds_sessioninfo();

// Mapping to search
session_info.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
session_info.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
session_info.nai.source_addr =
    (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
session_info.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
session_info.nai.extension_addr = 0;
```

```

int session_size = Marshal.SizeOf(session_info);
IntPtr session_ptr = Marshal.AllocHGlobal(session_size);
Marshal.StructureToPtr(session_info, session_ptr, false);

// Get Session information
result = UDSApi.GetValue_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    uds_parameter.PUDS_PARAMETER_SESSION_INFO, session_ptr, (uint)session_size);
if (UDSApi.StatusIsOk_2013(result))
{
    session_info = (uds_sessioninfo)Marshal.PtrToStructure(session_ptr,
        typeof(uds_sessioninfo));

    Console.WriteLine("Current session info:");
    Console.WriteLine("\t- CAN message type: " + session_info.can_msg_type);
    Console.WriteLine("\t- Extension address: " + session_info.nai.extension_addr);
    Console.WriteLine("\t- Protocol: " + session_info.nai.protocol);
    Console.WriteLine("\t- Source address: " + session_info.nai.source_addr);
    Console.WriteLine("\t- Target address: " + session_info.nai.target_addr);
    Console.WriteLine("\t- Target type: " + session_info.nai.target_type);
    Console.WriteLine("\t- S3 client value: " + session_info.s3_client_ms);
    Console.WriteLine("\t- Session type: " + session_info.session_type);
    Console.WriteLine("\t- P2CAN server max enhanced timeout: " +
        session_info.timeout_enhanced_p2can_server_max);
    Console.WriteLine("\t- P2CAN server max timeout: " +
        session_info.timeout_p2can_server_max);
}
else
{
    Console.WriteLine("Error, cannot get session information.");
}
Marshal.FreeHGlobal(session_ptr);

```

C++/CLR

```

uds_status result;
uds_sessioninfo session_info = {};

// Mapping to search
session_info.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
session_info.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
session_info.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
session_info.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
session_info.nai.extension_addr = 0;

int session_size = Marshal::SizeOf(session_info);
IntPtr session_ptr = Marshal::AllocHGlobal(session_size);
Marshal::StructureToPtr(session_info, session_ptr, false);

// Get Session information
result = UDSApi::GetValue_2013(PCANTP_HANDLE_USBBUS1, PUDS_PARAMETER_SESSION_INFO, session_ptr,
    session_size);
if (UDSApi::StatusIsOk_2013(result))
{
    session_info = (uds_sessioninfo)Marshal::PtrToStructure(session_ptr,
        uds_sessioninfo::typeid);

    Console::WriteLine("Current session info:");
    Console::WriteLine("\t- CAN message type: {0}", (int)session_info.can_msg_type);
    Console::WriteLine("\t- Extension address: " + session_info.nai.extension_addr);
    Console::WriteLine("\t- Protocol: {0}", (int)session_info.nai.protocol);
    Console::WriteLine("\t- Source address: " + session_info.nai.source_addr);
    Console::WriteLine("\t- Target address: " + session_info.nai.target_addr);
}

```

```

        Console.WriteLine("\t- Target type: {0}", (int)session_info.nai.target_type);
        Console.WriteLine("\t- S3 client value: " + session_info.s3_client_ms);
        Console.WriteLine("\t- Session type: " + session_info.session_type);
        Console.WriteLine("\t- P2CAN server max enhanced timeout: " +
            session_info.timeout_enhanced_p2can_server_max);
        Console.WriteLine("\t- P2CAN server max timeout: " +
            session_info.timeout_p2can_server_max);
    }
    else
    {
        Console.WriteLine("Error, cannot get session information.");
    }
    Marshal::FreeHGlobal(session_ptr);

```

Visual Basic

```

Dim result As uds_status
Dim session_info As uds_sessioninfo

' Mapping to search
session_info.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
session_info.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
session_info.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
session_info.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
session_info.nai.extension_addr = 0

Dim session_size As Integer
session_size = Marshal.SizeOf(session_info)
Dim session_ptr As IntPtr
session_ptr = Marshal.AllocHGlobal(session_size)
Marshal.StructureToPtr(session_info, session_ptr, False)

' Get Session information
result = UDSApi.GetValue_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    uds_parameter.PUDS_PARAMETER_SESSION_INFO, session_ptr, CType(session_size, UInt32))
If UDSApi.StatusIsOk_2013(result) Then
    session_info = CType(Marshal.PtrToStructure(session_ptr, GetType(uds_sessioninfo)),
        uds_sessioninfo)
    Console.WriteLine("Current session info:")
    Console.WriteLine("    - CAN message type: " + session_info.can_msg_type.ToString())
    Console.WriteLine("    - Extension address: " + session_info.nai.extension_addr.ToString())
    Console.WriteLine("    - Protocol: " + session_info.nai.protocol.ToString())
    Console.WriteLine("    - Source address: " + session_info.nai.source_addr.ToString())
    Console.WriteLine("    - Target address: " + session_info.nai.target_addr.ToString())
    Console.WriteLine("    - Target type: " + session_info.nai.target_type.ToString())
    Console.WriteLine("    - S3 client value: " + session_info.s3_client_ms.ToString())
    Console.WriteLine("    - Session type: " + session_info.session_type.ToString())
    Console.WriteLine("    - P2CAN server max enhanced timeout: " +
        session_info.timeout_enhanced_p2can_server_max.ToString())
    Console.WriteLine("    - P2CAN server max timeout: " +
        session_info.timeout_p2can_server_max.ToString())
Else
    Console.WriteLine("Error, cannot get session information.")
End If
Marshal.FreeHGlobal(session_ptr)

```

Pascal OO

```

var
    result: uds_status;
    session_info: uds_sessioninfo;
begin

```

```

// Mapping to search
session_info.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
session_info.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
session_info.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
session_info.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
session_info.nai.extension_addr := 0;

// Get Session information
result := TUDSApi.GetValue_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    uds_parameter.PUDS_PARAMETER_SESSION_INFO, Pointer(@session_info),
    UInt32(sizeof(session_info)));
if TUDSApi.StatusIsOk_2013(result) then
begin
    WriteLn('Current session info:');
    WriteLn(format(' - CAN message type: %d',
        [Integer(session_info.can_msg_type)]));
    WriteLn(format(' - Extension address: %d',
        [Integer(session_info.nai.extension_addr)]));
    WriteLn(format(' - Protocol: %d', [Integer(session_info.nai.protocol)]));
    WriteLn(format(' - Source address: %d',
        [Integer(session_info.nai.source_addr)]));
    WriteLn(format(' - Target address: %d',
        [Integer(session_info.nai.target_addr)]));
    WriteLn(format(' - Target type: %d',
        [Integer(session_info.nai.target_type)]));
    WriteLn(format(' - S3 client value: %d',
        [Integer(session_info.s3_client_ms)]));
    WriteLn(format(' - Session type: %d',
        [Integer(session_info.session_type)]));
    WriteLn(format(' - P2CAN server max enhanced timeout: %d',
        [Integer(session_info.timeout_enhanced_p2can_server_max)]));
    WriteLn(format(' - P2CAN server max timeout: %d',
        [Integer(session_info.timeout_p2can_server_max)]));
end
else
begin
    WriteLn('Error, cannot get session information.');
```

See also: [SetValue_2013](#) on page 153, [uds_parameter](#) on page 41, Detailed Parameters Characteristics on page 45.

Plain function version: [UDS_GetValue_2013](#) on page 630.

3.7.21 GetCanBusStatus_2013

Gets information about the internal BUS status of a PUDS channel.

Syntax

Pascal OO

```

class function GetCanBusStatus_2013(
    channel: cantp_handle
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_GetCanBusStatus_2013")]
public static extern uds_status GetCanBusStatus_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_GetCanBusStatus_2013")]
static uds_status GetCanBusStatus_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_GetCanBusStatus_2013")>
Public Shared Function GetCanBusStatus_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|--|
| <code>PUDS_STATUS_FLAG_BUS_LIGHT</code> | Indicates a bus error within the given PUDS channel. The hardware is in bus-light status. |
| <code>PUDS_STATUS_FLAG_BUS_HEAVY</code> | Indicates a bus error within the given PUDS channel. The hardware is in bus-heavy status. |
| <code>PUDS_STATUS_FLAG_BUS_OFF</code> | Indicates a bus error within the given PUDS channel. The hardware is in bus-off status. |
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |


Remarks

When the hardware status is bus-off, an application cannot communicate anymore. Consider using the PCAN-Basic property `PCAN_BUSOFF_AUTORESET` which instructs the API to automatically reset the CAN controller when a bus-off state is detected.

Another way to reset errors like bus-off, bus-heavy and bus-light, is to uninitialized and initialize again the channel used. This causes a hardware reset.

Example

The following example shows the use of the method `GetCanBusStatus_2013` on the channel `PCANTP_HANDLE_PCIBUS1`. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;

// Check the status of the PCI channel
```

```

result = UDSApi.GetCanBusStatus_2013(cantp_handle.PCANTP_HANDLE_PCIBUS1);
switch (result)
{
    case uds_status.PUDS_STATUS_FLAG_BUS_LIGHT:
        MessageBox.Show("PCAN-PCI (Ch-1): Handling a BUS-LIGHT status...", "Success");
        break;
    case uds_status.PUDS_STATUS_FLAG_BUS_HEAVY:
        MessageBox.Show("PCAN-PCI (Ch-1): Handling a BUS-HEAVY status...", "Success");
        break;
    case uds_status.PUDS_STATUS_FLAG_BUS_OFF:
        MessageBox.Show("PCAN-PCI (Ch-1): Handling a BUS-OFF status...", "Success");
        break;
    case uds_status.PUDS_STATUS_OK:
        MessageBox.Show("PCAN-PCI (Ch-1): Status is OK", "Success");
        break;
    default:
        // An error occurred
        MessageBox.Show("Failed to retrieve status", "Error");
        break;
}

```

C++ / CLR

```

uds_status result;

// Check the status of the PCI channel
result = UDSApi::GetCanBusStatus_2013(PCANTP_HANDLE_PCIBUS1);
switch (result)
{
    case PUDS_STATUS_FLAG_BUS_LIGHT:
        MessageBox::Show("PCAN-PCI (Ch-1): Handling a BUS-LIGHT status...", "Success");
        break;
    case PUDS_STATUS_FLAG_BUS_HEAVY:
        MessageBox::Show("PCAN-PCI (Ch-1): Handling a BUS-HEAVY status...", "Success");
        break;
    case PUDS_STATUS_FLAG_BUS_OFF:
        MessageBox::Show("PCAN-PCI (Ch-1): Handling a BUS-OFF status...", "Success");
        break;
    case PUDS_STATUS_OK:
        MessageBox::Show("PCAN-PCI (Ch-1): Status is OK", "Success");
        break;
    default:
        // An error occurred
        MessageBox::Show("Failed to retrieve status", "Error");
        break;
}

```

Visual Basic

```

Dim result As uds_status

' Check the status of the PCI channel
result = UDSApi.GetCanBusStatus_2013(cantp_handle.PCANTP_HANDLE_PCIBUS1)
Select Case (result)
    Case uds_status.PUDS_STATUS_FLAG_BUS_LIGHT
        MessageBox.Show("PCAN-PCI (Ch-1): Handling a BUS-LIGHT status...", "Success")
    Case uds_status.PUDS_STATUS_FLAG_BUS_HEAVY
        MessageBox.Show("PCAN-PCI (Ch-1): Handling a BUS-HEAVY status...", "Success")
    Case uds_status.PUDS_STATUS_FLAG_BUS_OFF
        MessageBox.Show("PCAN-PCI (Ch-1): Handling a BUS-OFF status...", "Success")
    Case uds_status.PUDS_STATUS_OK
        MessageBox.Show("PCAN-PCI (Ch-1): Status is OK", "Success")
    Case Else

```



```
' An error occurred
MessageBox.Show("Failed to retrieve status", "Error")
End Select
```

Pascal OO

```
var
  result: uds_status;
begin
  // Check the status of the PCI channel
  result := TUDSApi.GetCanBusStatus_2013(cantp_handle.PCANTP_HANDLE_PCIBUS1);
  case result of
    uds_status.PUDS_STATUS_FLAG_BUS_LIGHT:
      MessageBox(0, 'PCAN-PCI (Ch-1): Handling a BUS-LIGHT status...',
        'Success', MB_OK);
    uds_status.PUDS_STATUS_FLAG_BUS_HEAVY:
      MessageBox(0, 'PCAN-PCI (Ch-1): Handling a BUS-HEAVY status...',
        'Success', MB_OK);
    uds_status.PUDS_STATUS_FLAG_BUS_OFF:
      MessageBox(0, 'PCAN-PCI (Ch-1): Handling a BUS-OFF status...',
        'Success', MB_OK);
    uds_status.PUDS_STATUS_OK:
      MessageBox(0, 'PCAN-PCI (Ch-1): Status is OK', 'Success', MB_OK);
  else
    // An error occurred
    MessageBox(0, 'Failed to retrieve status', 'Error', MB_OK);
  end;
end;
```

See also: [uds_status](#) on page 32.

Plain function version: [UDS_GetCanBusStatus_2013](#) on page 642.

3.7.22 GetMapping_2013

Retrieves a mapping matching the given CAN identifier and message type (11bits, 29 bits, FD, etc.).

Syntax

Pascal OO

```
class function GetMapping_2013(
  channel: cantp_handle;
  var buffer: uds_mapping;
  can_id: UInt32;
  can_msgtype: cantp_can_msgtype
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_GetMapping_2013")]
public static extern uds_status GetMapping_2013(
  [MarshalAs(UnmanagedType.U4)]
  cantp_handle channel,
  out uds_mapping buffer,
  UInt32 can_id,
  cantp_can_msgtype can_msgtype);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_GetMapping_2013")]
static uds_status GetMapping_2013(
```

```
[MarshalAs(UnmanagedType::U4)]
cantp_handle channel,
uds_mapping %buffer,
UInt32 can_id,
cantp_can_msgtype can_msgtype);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_GetMapping_2013")>
Public Shared Function GetMapping_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByRef buffer As uds_mapping,
    ByVal can_id As UInt32,
    ByVal can_msgtype As cantp_can_msgtype) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| buffer | Output, buffer to store the searched mapping (see uds_mapping on page 25). |
| can_id | The CAN identifier to look for (see predefined uds_can_id values on page 58). |
| can_msgtype | The CAN message type to look for (11bits, 29 bits, FD, etc.). See cantp_can_msgtype on page 121. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical error in case of failure is:

| | |
|-------------------------------------|--|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The buffer is invalid. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | Indicates that no matching mapping was found in the registered mapping list. |

Example

The following example shows the use of the method `GetMapping_2013` on the channel `PCANTP_HANDLE_USBBUS1`. It gets the mapping for the 0x7E0 CAN identifier then prints it.

Note: It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_mapping result_mapping;
result = UDSApi.GetMapping_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, out result_mapping,
    (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1,
    cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD);

if (UDSApi.StatusIsOk_2013(result))
{
    Console.WriteLine("Mapping defined for the " + result_mapping.can_id + " can identifier:");
    Console.WriteLine("\t- Flow control identifier: " + result_mapping.can_id_flow_ctrl);
    Console.WriteLine("\t- CAN message type: " + result_mapping.can_msgtype);
    Console.WriteLine("\t- TX DLC: " + result_mapping.can_tx_dlc);
    Console.WriteLine("\t- Extension address: " + result_mapping.nai.extension_addr);
    Console.WriteLine("\t- Protocol: " + result_mapping.nai.protocol);
    Console.WriteLine("\t- Source address: " + result_mapping.nai.source_addr);
    Console.WriteLine("\t- Target address: " + result_mapping.nai.target_addr);
    Console.WriteLine("\t- Target type: " + result_mapping.nai.target_type);
}
```

```
else
{
    Console.WriteLine("Error, cannot get mapping information.");
}
```

C++ / CLR

```
uds_status result;
uds_mapping result_mapping;
result = UDSApi::GetMapping_2013(PCANTP_HANDLE_USBBUS1, result_mapping,
    PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1, PCANTP_CAN_MSGTYPE_STANDARD);

if (UDSApi::StatusIsOk_2013(result))
{
    Console::WriteLine("Mapping defined for the " + result_mapping.can_id +
        " can identifier:");
    Console::WriteLine("\t- Flow control identifier: " + result_mapping.can_id_flow_ctrl);
    Console::WriteLine("\t- CAN message type: {0}", (int)result_mapping.can_msgtype);
    Console::WriteLine("\t- TX DLC: " + result_mapping.can_tx_dlc);
    Console::WriteLine("\t- Extension address: " + result_mapping.nai.extension_addr);
    Console::WriteLine("\t- Protocol: {0}", (int)result_mapping.nai.protocol);
    Console::WriteLine("\t- Source address: " + result_mapping.nai.source_addr);
    Console::WriteLine("\t- Target address: " + result_mapping.nai.target_addr);
    Console::WriteLine("\t- Target type: {0}", (int)result_mapping.nai.target_type);
}
else
{
    Console::WriteLine("Error, cannot get mapping information.");
}
```

Visual Basic

```
Dim result As uds_status
Dim result_mapping As uds_mapping
result = UDSApi.GetMapping_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, result_mapping,
    uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1,
    cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD)

If UDSApi.StatusIsOk_2013(result) Then
    Console.WriteLine("Mapping defined for the " + result_mapping.can_id.ToString() +
        " can identifier:")
    Console.WriteLine(" - Flow control identifier: " +
        result_mapping.can_id_flow_ctrl.ToString())
    Console.WriteLine(" - CAN message type: " + result_mapping.can_msgtype.ToString())
    Console.WriteLine(" - TX DLC: " + result_mapping.can_tx_dlc.ToString())
    Console.WriteLine(" - Extension address: " + result_mapping.nai.extension_addr.ToString())
    Console.WriteLine(" - Protocol: " + result_mapping.nai.protocol.ToString())
    Console.WriteLine(" - Source address: " + result_mapping.nai.source_addr.ToString())
    Console.WriteLine(" - Target address: " + result_mapping.nai.target_addr.ToString())
    Console.WriteLine(" - Target type: " + result_mapping.nai.target_type.ToString())
Else
    Console.WriteLine("Error, cannot get mapping information.")
End If
```

Pascal OO

```
var
    result: uds_status;
    result_mapping: uds_mapping;
begin
    result := TUDSApi.GetMapping_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        result_mapping,
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1),
```

```

cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD);

if TUDSApi.StatusIsOk_2013(result) then
begin
  WriteLn(format('Mapping defined for the %d can identifier:',
    [Integer(result_mapping.can_id)]));
  WriteLn(format(' - Flow control identifier: %d',
    [Integer(result_mapping.can_id_flow_ctrl)]));
  WriteLn(format(' - CAN message type: %d',
    [Integer(result_mapping.can_msgtype)]));
  WriteLn(format(' - TX DLC: %d', [Integer(result_mapping.can_tx_dlc)]));
  WriteLn(format(' - Extension address: %d',
    [Integer(result_mapping.nai.extension_addr)]));
  WriteLn(format(' - Protocol: %d', [Integer(result_mapping.nai.protocol)]));
  WriteLn(format(' - Source address: %d',
    [Integer(result_mapping.nai.source_addr)]));
  WriteLn(format(' - Target address: %d',
    [Integer(result_mapping.nai.target_addr)]));
  WriteLn(format(' - Target type: %d',
    [Integer(result_mapping.nai.target_type)]));
end
else
begin
  WriteLn('Error, cannot get mapping information.');

```

See also: [uds_mapping](#) on page 25, [GetMappings_2013](#) on page 196.

Plain function version: [UDS_GetMapping_2013](#) on page 637.

3.7.23 GetMappings_2013

Retrieves all the mappings defined for a PUDS channel.

Syntax

Pascal OO

```

class function GetMappings_2013(
  channel: cantp_handle;
  buffer: Puds_mapping;
  buffer_length: UInt16;
  count: PUInt16
): uds_status;
```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_GetMappings_2013")]
public static extern uds_status GetMappings_2013(
  [MarshalAs(UnmanagedType.U4)]
  cantp_handle channel,
  [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 2)]
  [Out] uds_mapping[] buffer,
  UInt16 buffer_length,
  ref UInt16 count);
```

C++ / CLR

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_GetMappings_2013")]
static uds_status GetMappings_2013(
  [MarshalAs(UnmanagedType::U4)]
```

```

cantp_handle channel,
[MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 2)]
[Out] array<uds_mapping> ^buffer,
UInt16 buffer_length,
UInt16 %count);

```

Visual Basic

```

<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_GetMappings_2013")>
Public Shared Function GetMappings_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=2)>
    <Out> ByVal buffer As uds_mapping(),
    ByVal buffer_length As UInt16,
    ByRef count As UInt16) As uds_status
End Function

```

Parameters

| Parameter | Description |
|---------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| buffer | Output, a buffer to store an array of uds_mapping (see uds_mapping on page 25). |
| buffer_length | The number of uds_mapping elements the buffer can store. |
| count | Output, the actual number of elements copied in the buffer. |

Remark

By default, some mappings are initialized in the PCAN-UDS 2.x API (see UDS and ISO-TP Network Addressing Information on page 770).


Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|------------------------------------|--|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_BUFFER_TOO_SMALL | The given buffer is too small to store all mappings. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The buffer is invalid. |

Example

The following example shows the use of the method `GetMappings_2013` on `PCANTP_HANDLE_USBBUS1`. It displays all mappings added on the channel.

 **Note:** It is assumed that the channel was already initialized.

C#

```

uds_status result;
UInt16 count = 256;
uds_mapping[] mappings = new uds_mapping[256];
result = UDSApi.GetMappings_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, mappings, count, ref
    count);

if (UDSApi.StatusIsOk_2013(result))
{
    for (int i = 0; i < count; i++)
    {
        Console.WriteLine("mappings[" + i + "]:");
    }
}

```

```

        Console.WriteLine("\t- CAN identifier: " + mappings[i].can_id);
        Console.WriteLine("\t- Flow control identifier: " + mappings[i].can_id_flow_ctrl);
        Console.WriteLine("\t- CAN message type: " + mappings[i].can_msgtype);
        Console.WriteLine("\t- TX DLC: " + mappings[i].can_tx_dlc);
        Console.WriteLine("\t- Extension address: " + mappings[i].nai.extension_addr);
        Console.WriteLine("\t- Protocol: " + mappings[i].nai.protocol);
        Console.WriteLine("\t- Source address: " + mappings[i].nai.source_addr);
        Console.WriteLine("\t- Target address: " + mappings[i].nai.target_addr);
        Console.WriteLine("\t- Target type: " + mappings[i].nai.target_type);
    }
}
else
{
    Console.WriteLine("Error, cannot get mappings.");
}

```

C++ / CLR

```

uds_status result;
UInt16 count = 256;
array<uds_mapping>^ mappings = gcnew array<uds_mapping>(256);
result = UDSApi::GetMappings_2013(PCANTP_HANDLE_USBBUS1, mappings, count, count);

if (UDSApi::StatusIsOk_2013(result))
{
    for (int i = 0; i < count; i++)
    {
        Console::WriteLine("mappings[" + i + "]:");
        Console::WriteLine("\t- CAN identifier: " + mappings[i].can_id);
        Console::WriteLine("\t- Flow control identifier: " +
            mappings[i].can_id_flow_ctrl);
        Console::WriteLine("\t- CAN message type: {0}", (int)mappings[i].can_msgtype);
        Console::WriteLine("\t- TX DLC: " + mappings[i].can_tx_dlc);
        Console::WriteLine("\t- Extension address: " + mappings[i].nai.extension_addr);
        Console::WriteLine("\t- Protocol: {0}", (int)mappings[i].nai.protocol);
        Console::WriteLine("\t- Source address: " + mappings[i].nai.source_addr);
        Console::WriteLine("\t- Target address: " + mappings[i].nai.target_addr);
        Console::WriteLine("\t- Target type: {0} ", (int)mappings[i].nai.target_type);
    }
}
else
{
    Console::WriteLine("Error, cannot get mappings.");
}

```

Visual Basic

```

Dim result As uds_status
Dim count As UInt16 = 256
Dim mappings(256) As uds_mapping

result = UDSApi.GetMappings_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, mappings, count, count)
If UDSApi.StatusIsOk_2013(result) Then
    For i As UInt32 = 0 To count - 1
        Console.WriteLine("mappings[" + i.ToString() + "]:")
        Console.WriteLine("  - CAN identifier: " + mappings(i).can_id.ToString())
        Console.WriteLine("  - Flow control identifier: " +
            mappings(i).can_id_flow_ctrl.ToString())
        Console.WriteLine("  - CAN message type: " + mappings(i).can_msgtype.ToString())
        Console.WriteLine("  - TX DLC: " + mappings(i).can_tx_dlc.ToString())
        Console.WriteLine("  - Extension address: " +
            mappings(i).nai.extension_addr.ToString())
    Next

```

```

        Console.WriteLine(" - Protocol: " + mappings(i).nai.protocol.ToString())
        Console.WriteLine(" - Source address: " + mappings(i).nai.source_addr.ToString())
        Console.WriteLine(" - Target address: " + mappings(i).nai.target_addr.ToString())
        Console.WriteLine(" - Target type: " + mappings(i).nai.target_type.ToString())
    Next
Else
    Console.WriteLine("Error, cannot get mappings.")
End If

```

Pascal OO

```

var
    result: uds_status;
    i: UInt32;
    count: UInt16;
    mappings: array [0 .. 255] of uds_mapping;
begin
    count := 256;
    result := TUDSApi.GetMappings_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @mappings, count, @count);

    if TUDSApi.StatusIsOk_2013(result) then
        begin
            for i := 0 to count - 1 do
                begin
                    WriteLn(format('mappings[%d]:', [Integer(i)]));
                    WriteLn(format(' - CAN identifier: %d', [Integer(mappings[i].can_id)]));
                    WriteLn(format(' - Flow control identifier: %d',
                        [Integer(mappings[i].can_id_flow_ctrl)]));
                    WriteLn(format(' - CAN message type: %d',
                        [Integer(mappings[i].can_msgtype)]));
                    WriteLn(format(' - TX DLC: %d', [Integer(mappings[i].can_tx_dlc)]));
                    WriteLn(format(' - Extension address: %d',
                        [Integer(mappings[i].nai.extension_addr)]));
                    WriteLn(format(' - Protocol: %d', [Integer(mappings[i].nai.protocol)]));
                    WriteLn(format(' - Source address: %d',
                        [Integer(mappings[i].nai.source_addr)]));
                    WriteLn(format(' - Target address: %d',
                        [Integer(mappings[i].nai.target_addr)]));
                    WriteLn(format(' - Target type: %d',
                        [Integer(mappings[i].nai.target_type)]));
                end
            end
        end
    else
        begin
            WriteLn('Error, cannot get mappings.');
```

See also: [uds_mapping](#) on page 25, [GetMapping_2013](#) on page 193.

Plain function version: [UDS_GetMappings_2013](#) on page 638.

3.7.24 GetSessionInformation_2013

Gets current ECU session information.

Syntax

Pascal OO

```
class function GetSessionInformation_2013(
    channel: cantp_handle;
    var session_info: uds_sessioninfo
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_GetSessionInformation_2013")]
public static extern uds_status GetSessionInformation_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    out uds_sessioninfo session_info);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_GetSessionInformation_2013")]
static uds_status GetSessionInformation_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_sessioninfo %session_info);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_GetSessionInformation_2013")>
Public Shared Function GetSessionInformation_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByRef session_info As uds_sessioninfo) As uds_status
End Function
```

Parameters

| Parameter | Description |
|--------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| session_info | Input, the mapping to search for. Output, the session information filled if an ECU session exists (see uds_sessioninfo on page 22). |


Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---------------------------------|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. Or the ECU session information is not initialized. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is invalid. |

Example

The following example shows the use of the method `GetSessionInformation_2013` on the channel `PCANTP_HANDLE_USBBUS1`. It gets the session information for a given mapping and prints it.

 **Note:** It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_sessioninfo session_info;

// Mapping to search
session_info.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
session_info.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
session_info.nai.source_addr =
    (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
session_info.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
session_info.nai.extension_addr = 0;

result = UDSApi.GetSessionInformation_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, out
    session_info);

if (UDSApi.StatusIsOk_2013(result))
{
    Console.WriteLine("Current session info:");
    Console.WriteLine("\t- CAN message type: " + session_info.can_msg_type);
    Console.WriteLine("\t- Extension address: " + session_info.nai.extension_addr);
    Console.WriteLine("\t- Protocol: " + session_info.nai.protocol);
    Console.WriteLine("\t- Source address: " + session_info.nai.source_addr);
    Console.WriteLine("\t- Target address: " + session_info.nai.target_addr);
    Console.WriteLine("\t- Target type: " + session_info.nai.target_type);
    Console.WriteLine("\t- S3 client value: " + session_info.s3_client_ms);
    Console.WriteLine("\t- Session type: " + session_info.session_type);
    Console.WriteLine("\t- P2CAN server max enhanced timeout: " +
        session_info.timeout_enhanced_p2can_server_max);
    Console.WriteLine("\t- P2CAN server max enhanced timeout: " +
        session_info.timeout_p2can_server_max);
}
else
{
    Console.WriteLine("Error, cannot get session information.");
}

```

C++ / CLR

```

uds_status result;
uds_sessioninfo session_info;

// Mapping to search
session_info.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
session_info.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
session_info.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
session_info.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
session_info.nai.extension_addr = 0;

result = UDSApi::GetSessionInformation_2013(PCANTP_HANDLE_USBBUS1, session_info);

if (UDSApi::StatusIsOk_2013(result))
{
    Console::WriteLine("Current session info:");
    Console::WriteLine("\t- CAN message type: {0}", (int)session_info.can_msg_type);
    Console::WriteLine("\t- Extension address: " + session_info.nai.extension_addr);
    Console::WriteLine("\t- Protocol: {0}", (int)session_info.nai.protocol);
    Console::WriteLine("\t- Source address: " + session_info.nai.source_addr);
    Console::WriteLine("\t- Target address: " + session_info.nai.target_addr);
    Console::WriteLine("\t- Target type: {0}", (int)session_info.nai.target_type);
    Console::WriteLine("\t- S3 client value: " + session_info.s3_client_ms);
    Console::WriteLine("\t- Session type: " + session_info.session_type);
}

```

```

        Console.WriteLine("\t- P2CAN server max enhanced timeout: " +
            session_info.timeout_enhanced_p2can_server_max);
        Console.WriteLine("\t- P2CAN server max enhanced timeout: " +
            session_info.timeout_p2can_server_max);
    }
    else
    {
        Console.WriteLine("Error, cannot get session information.");
    }
}

```

Visual Basic

```

Dim result As uds_status
Dim session_info As uds_sessioninfo

' Mapping to search
session_info.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
session_info.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
session_info.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
session_info.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
session_info.nai.extension_addr = 0

result = UDSApi.GetSessionInformation_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, session_info)
If UDSApi.StatusIsOk_2013(result) Then
    Console.WriteLine("Current session info:")
    Console.WriteLine(" - CAN message type: " + session_info.can_msg_type.ToString())
    Console.WriteLine(" - Extension address: " + session_info.nai.extension_addr.ToString())
    Console.WriteLine(" - Protocol: " + session_info.nai.protocol.ToString())
    Console.WriteLine(" - Source address: " + session_info.nai.source_addr.ToString())
    Console.WriteLine(" - Target address: " + session_info.nai.target_addr.ToString())
    Console.WriteLine(" - Target type: " + session_info.nai.target_type.ToString())
    Console.WriteLine(" - S3 client value: " + session_info.s3_client_ms.ToString())
    Console.WriteLine(" - Session type: " + session_info.session_type.ToString())
    Console.WriteLine(" - P2CAN server max enhanced timeout: " +
        session_info.timeout_enhanced_p2can_server_max.ToString())
    Console.WriteLine(" - P2CAN server max enhanced timeout: " +
        session_info.timeout_p2can_server_max.ToString())
Else
    Console.WriteLine("Error, cannot get session information.")
End If

```

Pascal OO

```

var
    result: uds_status;
    session_info: uds_sessioninfo;
begin

    // Mapping to search
    session_info.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    session_info.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    session_info.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    session_info.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    session_info.nai.extension_addr := 0;

    result := TUDSApi.GetSessionInformation_2013
        (cantp_handle.PCANTP_HANDLE_USBBUS1, session_info);

```

```

if TUDSApi.StatusIsOk_2013(result) then
begin
  WriteLn('Current session info:');
  WriteLn(format(' - CAN message type: %d',
    [Integer(session_info.can_msg_type)]));
  WriteLn(format(' - Extension address: %d',
    [Integer(session_info.nai.extension_addr)]));
  WriteLn(format(' - Protocol: %d', [Integer(session_info.nai.protocol)]));
  WriteLn(format(' - Source address: %d',
    [Integer(session_info.nai.source_addr)]));
  WriteLn(format(' - Target address: %d',
    [Integer(session_info.nai.target_addr)]));
  WriteLn(format(' - Target type: %d',
    [Integer(session_info.nai.target_type)]));
  WriteLn(format(' - S3 client value: %d',
    [Integer(session_info.s3_client_ms)]));
  WriteLn(format(' - Session type: %d',
    [Integer(session_info.session_type)]));
  WriteLn(format(' - P2CAN server max enhanced timeout: %d',
    [Integer(session_info.timeout_enhanced_p2can_server_max)]));
  WriteLn(format(' - P2CAN server max enhanced timeout: %d',
    [Integer(session_info.timeout_p2can_server_max)]));
end
else
begin
  WriteLn('Error, cannot get session information.');
```




See also: [uds_sessioninfo](#) on page 22, [SetValue_2013](#) on page 153, [SvcDiagnosticSessionControl_2013](#) on page 258.

Plain function version: [UDS_GetSessionInformation_2013](#) on page 639.

3.7.25 StatusIsOk_2013

Checks if a PUDS status matches an expected result (default is [PUDS_STATUS_OK](#)).

Overloads

| | Method | Description |
|---|---|--|
|  | StatusIsOk_2013(uds_status) | Checks if the uds_status matches PUDS_STATUS_OK in a non-strict mode. |
|  | StatusIsOk_2013(uds_status, uds_status) | Checks if a uds_status matches the expected result in a non-strict mode. |
|  | StatusIsOk_2013(uds_status, uds_status, bool) | Checks if a uds_status matches an expected result. |

Plain function version: [UDS_StatusIsOk_2013](#) on page 640.

3.7.26 StatusIsOk_2013(uds_status)

Checks if the `uds_status` matches `PUDS_STATUS_OK` in a non-strict mode.

Syntax

Pascal OO

```
class function StatusIsOk_2013(
    status: uds_status
): Boolean; overload;
```

C#

```
public static bool StatusIsOk_2013(
    uds_status status);
```

C++ / CLR

```
static bool StatusIsOk_2013(
    uds_status status);
```

Visual Basic

```
Public Shared Function StatusIsOk_2013(
    ByVal status As uds_status) As Boolean
End Function
```

Parameters

| Parameter | Description |
|-----------|---|
| status | The status to analyze (see <code>uds_status</code> on page 32). |

Returns

The return value is true if the status matches `PUDS_STATUS_OK` (in non-strict mode)

Remarks

When checking a `uds_status`, it is preferred to use `UDS_StatusIsOk_2013` instead of comparing it with the `==` operator because `UDS_StatusIsOk_2013` can remove information flag (in non-strict mode).

Example

The following example shows the use of the method `StatusIsOk_2013` after initializing the channel `PCANTP_HANDLE_PCIBUS2`.

C#

```
uds_status result;
result = UDSApi.Initialize_2013(cantp_handle.PCANTP_HANDLE_PCIBUS2,
cantp_baudrate.PCANTP_BAUDRATE_500K);
if (!UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Initialization failed", "Error");
else
    MessageBox.Show("PCAN-PCI (Ch-2) was initialized", "Success");
```

C++ / CLR

```
uds_status result;
result = UDSApi::Initialize_2013(PCANTP_HANDLE_PCIBUS2, PCANTP_BAUDRATE_500K);
if (!UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Initialization failed", "Error");
else
    MessageBox::Show("PCAN-PCI (Ch-2) was initialized", "Success");
```

Visual Basic

```
Dim result As uds_status
result = UDSApi.Initialize_2013(cantp_handle.PCANTP_HANDLE_PCIBUS2,
    cantp_baudrate.PCANTP_BAUDRATE_500K)
If Not UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Initialization failed", "Error")
Else
    MessageBox.Show("PCAN-PCI (Ch-2) was initialized", "Success")
End If
```

Pascal OO

```
var
    result: uds_status;
begin
    result := TUDSApi.Initialize_2013(cantp_handle.PCANTP_HANDLE_PCIBUS2,
        cantp_baudrate.PCANTP_BAUDRATE_500K);
    if not TUDSApi.StatusIsOk_2013(result) then
        begin
            MessageBox(0, 'Initialization failed', 'Error', MB_OK);
        end
    else
        begin
            MessageBox(0, 'PCAN-PCI (Ch-2) was initialized', 'Success', MB_OK);
        end;
end;
```

See also: `uds_status` on page 32.

Plain function version: `UDS_StatusIsOk_2013` on page 640.

3.7.27 StatusIsOk_2013(uds_status, uds_status)

Checks if a `uds_status` matches the expected result in a non-strict mode.

Syntax**Pascal OO**

```
class function StatusIsOk_2013(
    status: uds_status;
    status_expected: uds_status
): Boolean; overload;
```

C#

```
public static bool StatusIsOk_2013(
    uds_status status,
    uds_status status_expected);
```

C++ / CLR

```
static bool StatusIsOk_2013(
    uds_status status,
    uds_status status_expected);
```

Visual Basic

```
Public Shared Function StatusIsOk_2013(
    ByVal status As uds_status,
    ByVal status_expected As uds_status) As Boolean
End Function
```

Parameters

| Parameter | Description |
|-----------------|---|
| status | The status to analyze (see uds_status on page 32). |
| status_expected | The expected status (see uds_status on page 32). The default value is PUDS_STATUS_OK. |

Returns

The return value is true if the status matches expected parameter (in non-strict mode).

Remarks

When checking a `uds_status`, it is preferred to use `UDS_StatusIsOk_2013` instead of comparing it with the `==` operator because `UDS_StatusIsOk_2013` can remove information flag (in non-strict mode).

Example

The following example shows the use of the method `StatusIsOk_2013` after initializing the channel `PCANTP_HANDLE_PCIBUS2`.

C#

```
uds_status result;
result = UDSApi.Initialize_2013(cantp_handle.PCANTP_HANDLE_PCIBUS2,
    cantp_baudrate.PCANTP_BAUDRATE_500K);
if (!UDSApi.StatusIsOk_2013(result, uds_status.PUDS_STATUS_OK))
    MessageBox.Show("Initialization failed", "Error");
else
    MessageBox.Show("PCAN-PCI (Ch-2) was initialized", "Success");
```

C++ / CLR

```
uds_status result;
result = UDSApi::Initialize_2013(PCANTP_HANDLE_PCIBUS2, PCANTP_BAUDRATE_500K);
if (!UDSApi::StatusIsOk_2013(result, PUDS_STATUS_OK))
    MessageBox::Show("Initialization failed", "Error");
else
    MessageBox::Show("PCAN-PCI (Ch-2) was initialized", "Success");
```

Visual Basic

```
Dim result As uds_status
result = UDSApi.Initialize_2013(cantp_handle.PCANTP_HANDLE_PCIBUS2,
    cantp_baudrate.PCANTP_BAUDRATE_500K)
If Not UDSApi.StatusIsOk_2013(result, uds_status.PUDS_STATUS_OK) Then
    MessageBox.Show("Initialization failed", "Error")
Else
    MessageBox.Show("PCAN-PCI (Ch-2) was initialized", "Success")
End If
```

Pascal OO

```
var
  result: uds_status;
begin
  result := TUDSApi.Initialize_2013(cantp_handle.PCANTP_HANDLE_PCIBUS2,
    cantp_baudrate.PCANTP_BAUDRATE_500K);
  if not TUDSApi.StatusIsOk_2013(result, uds_status.PUDS_STATUS_OK) then
  begin
    MessageBox(0, 'Initialization failed', 'Error', MB_OK);
  end
  else
  begin
    MessageBox(0, 'PCAN-PCI (Ch-2) was initialized', 'Success', MB_OK);
  end;
end;
```

See also: `uds_status` on page 32.

Plain function version: `UDS_StatusIsOk_2013` on page 640.

3.7.28 StatusIsOk_2013(uds_status, uds_status, bool)

Checks if a `uds_status` matches an expected result.

Syntax

Pascal OO

```
class function StatusIsOk_2013(
  status: uds_status;
  status_expected: uds_status;
  strict_mode: Boolean
): Boolean; overload;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_StatusIsOk_2013")]
public static extern bool StatusIsOk_2013(
  [MarshalAs(UnmanagedType.U4)]
  uds_status status,
  [MarshalAs(UnmanagedType.U4)]
  uds_status status_expected,
  [MarshalAs(UnmanagedType.I1)]
  bool strict_mode);
```

C++ / CLR

```
static bool StatusIsOk_2013(
  uds_status status,
  uds_status status_expected,
  bool strict_mode);
```

Visual Basic

```
Public Shared Function StatusIsOk_2013(
  ByVal status As uds_status,
  ByVal status_expected As uds_status,
  ByVal strict_mode As Boolean) As Boolean
End Function
```

Parameters

| Parameter | Description |
|-----------------|--|
| status | The status to analyze (see uds_status on page 32). |
| status_expected | The expected status (see uds_status on page 32). The default value is PUDS_STATUS_OK. |
| strict_mode | Enable strict mode (default is false). Strict mode ensures that bus or extra information are the same. |

Returns

The return value is true if the status matches expected parameter.

Remarks

When checking a `uds_status`, it is preferred to use `UDS_StatusIsOk_2013` instead of comparing it with the `==` operator because `UDS_StatusIsOk_2013` can remove information flag (in non-strict mode).

Example

The following example shows the use of the method `StatusIsOk_2013` after initializing the channel `PCANTP_HANDLE_PCIBUS2`.

C#

```
uds_status result;
result = UDSApi.Initialize_2013(cantp_handle.PCANTP_HANDLE_PCIBUS2,
    cantp_baudrate.PCANTP_BAUDRATE_500K);
if (!UDSApi.StatusIsOk_2013(result, uds_status.PUDS_STATUS_OK, false))
    MessageBox.Show("Initialization failed", "Error");
else
    MessageBox.Show("PCAN-PCI (Ch-2) was initialized", "Success");
```

C++ / CLR

```
uds_status result;
result = UDSApi::Initialize_2013(PCANTP_HANDLE_PCIBUS2, PCANTP_BAUDRATE_500K);
if (!UDSApi::StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    MessageBox::Show("Initialization failed", "Error");
else
    MessageBox::Show("PCAN-PCI (Ch-2) was initialized", "Success");
```

Visual Basic

```
Dim result As uds_status
result = UDSApi.Initialize_2013(cantp_handle.PCANTP_HANDLE_PCIBUS2,
    cantp_baudrate.PCANTP_BAUDRATE_500K)
If Not UDSApi.StatusIsOk_2013(result, uds_status.PUDS_STATUS_OK, False) Then
    MessageBox.Show("Initialization failed", "Error")
Else
    MessageBox.Show("PCAN-PCI (Ch-2) was initialized", "Success")
End If
```

Pascal OO

```
var
    result: uds_status;
begin
    result := TUDSApi.Initialize_2013(cantp_handle.PCANTP_HANDLE_PCIBUS2,
        cantp_baudrate.PCANTP_BAUDRATE_500K);
    if not TUDSApi.StatusIsOk_2013(result, uds_status.PUDS_STATUS_OK, false) then
        begin
            MessageBox(0, 'Initialization failed', 'Error', MB_OK);
        end
    else
```



```
begin
    MessageBox(0, 'PCAN-PCI (Ch-2) was initialized', 'Success', MB_OK);
end;
end;
```

See also: `uds_status` on page 32.

Plain function version: `UDS_StatusIsOk_2013` on page 640.

3.7.29 GetErrorText_2013

Gets a descriptive text for a PUDS error code.

Syntax

Pascal OO

```
class function GetErrorText_2013(
    error_code: uds_status;
    language: UInt16;
    buffer: PAnsiChar;
    buffer_size: UInt32
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_GetErrorText_2013")]
public static extern uds_status GetErrorText_2013(
    [MarshalAs(UnmanagedType.U4)]
    uds_status error_code,
    UInt16 language,
    StringBuilder buffer,
    UInt32 buffer_size);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_GetErrorText_2013")]
static uds_status GetErrorText_2013(
    [MarshalAs(UnmanagedType::U4)]
    uds_status error_code,
    UInt16 language,
    StringBuilder ^buffer,
    UInt32 buffer_size);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_GetErrorText_2013")>
Public Shared Function GetErrorText_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal error_code As uds_status,
    ByVal language As UInt16,
    ByVal buffer As StringBuilder,
    ByVal buffer_size As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|------------|---|
| error_code | The <code>uds_status</code> error code (see <code>uds_status</code> on page 32). |
| language | The current languages available for translation are: Neutral (0x00), German (0x07), English (0x09), Spanish (0x0A), Italian (0x10) and French (0x0C). |

| Parameter | Description |
|-------------|--|
| buffer | A buffer for a null-terminated char array. |
| buffer_size | Buffer length in bytes. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical error in case of failure is:

| | |
|--|--|
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | Indicates that the parameters passed to the method are invalid. Check the parameter 'buffer'; it should point to a char array, big enough to allocate the text for the given error code. |
|--|--|

Remarks

The Primary Language IDs are codes used by Windows OS from Microsoft, to identify a human language. The PCAN-Basic API currently supports the following languages:

| Language | Primary Language ID |
|----------------------------|---------------------|
| Neutral (System dependent) | 00h (0) |
| English | 09h (9) |
| German | 07h (7) |
| French | 0Ch (12) |
| Italian | 10h (16) |
| Spanish | 0Ah (10) |

Note: If the buffer is too small for the resulting text, the error `0x80008000` (`PUDS_STATUS_MASK_PCAN|PCAN_ERROR_ILLPARAMVAL`) is returned. Even when only short texts are being currently returned, a text within this method can have a maximum of 255 characters. For this reason, it is recommended to use a buffer with a length of at least 256 bytes.

Example

The following example shows the use of the method `GetErrorText_2013` to get the description of an error. The language of the description's text will be the same used by the operating system (if its language is supported; otherwise English is used).

Note: It is assumed that the channel was NOT initialized (to generate an error).

C#

```
StringBuilder str_msg = new StringBuilder(256);
uds_status result;
uds_status error_result;
error_result = UDSApi.Uninitialize_2013(cantp_handle.PCANTP_HANDLE_USBBUS1);
result = UDSApi.GetErrorText_2013(error_result, 0x0, str_msg, 256);
if (UDSApi.StatusIsOk_2013(result) && !UDSApi.StatusIsOk_2013(error_result))
    MessageBox.Show(str_msg.ToString(), "Error on uninitialized");
```

C++ / CLR

```
StringBuilder^ str_msg = gcnew StringBuilder(256);
uds_status result;
uds_status error_result;
error_result = UDSApi::Uninitialize_2013(PCANTP_HANDLE_USBBUS1);
result = UDSApi::GetErrorText_2013(error_result, 0x0, str_msg, 256);
if (UDSApi::StatusIsOk_2013(result) && !UDSApi::StatusIsOk_2013(error_result))
    MessageBox::Show(str_msg->ToString(), "Error on uninitialized");
```

Visual Basic

```
Dim str_msg As StringBuilder
str_msg = New StringBuilder(256)
Dim result As uds_status
Dim error_result As uds_status
error_result = UDSApi.Uninitialize_2013(cantp_handle.PCANTP_HANDLE_USBBUS1)
result = UDSApi.GetErrorText_2013(error_result, &H0, str_msg, 256)
If UDSApi.StatusIsOk_2013(result) And Not UDSApi.StatusIsOk_2013(error_result) Then
    MessageBox.Show(str_msg.ToString(), "Error on uninitialized")
End If
```

Pascal OO

```
var
    result: uds_status;
    error_result: uds_status;
    str_msg: array [0 .. 255] of ansichar;
begin
    error_result := TUDSApi.Uninitialize_2013(cantp_handle.PCANTP_HANDLE_USBBUS1);
    result := TUDSApi.GetErrorText_2013(error_result, $0,
        PAnsichar(@str_msg), 256);
    if TUDSApi.StatusIsOk_2013(result) and not TUDSApi.StatusIsOk_2013
        (error_result) then
        begin
            MessageBox(0, PWideChar(String(str_msg)), 'Error on uninitialized', MB_OK);
        end;
    end;
```

See also: `uds_status` on page 32.

Plain function version: `UDS_GetErrorText_2013` on page 641.

3.7.30 MsgAlloc_2013

Allocates a PUDS message using the given configuration.

Syntax

Pascal OO

```
class function MsgAlloc_2013(
    var msg_buffer: uds_msg;
    msg_configuration: uds_msgconfig;
    msg_data_length: UInt32
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_MsgAlloc_2013")]
public static extern uds_status MsgAlloc_2013(
    out uds_msg msg_buffer,
    uds_msgconfig msg_configuration,
    UInt32 msg_data_length);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_MsgAlloc_2013")]
static uds_status MsgAlloc_2013(
    uds_msg %msg_buffer,
    uds_msgconfig msg_configuration,
    UInt32 msg_data_length);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_MsgAlloc_2013")>
Public Shared Function MsgAlloc_2013(
    ByRef msg_buffer As uds_msg,
    ByVal msg_configuration As uds_msgconfig,
    ByVal msg_data_length As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-------------------|--|
| msg_buffer | A uds_msg structure buffer. It will be freed if required (see uds_msg on page 21). |
| msg_configuration | Configuration of the message to allocate (see uds_msgconfig on page 25). |
| msg_data_length | Size of the message data. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---|--|
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the parameters is not valid. |
| PUDS_STATUS_CAUTION_INPUT_MODIFIED | Extra information, the message has been modified by the API. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

The `uds_msg` structure is automatically initialized and allocated by the PCAN-UDS 2.x API using:

- `MsgAlloc_2013` method
- UDS services method (suffixed `Svc`)
- `Read_2013` method
- `WaitFor` methods

Once processed, the `uds_msg` structure should be released using `MsgFree_2013` method.

Example

The following example shows the use of the method `MsgAlloc_2013`. It allocates a ClearDiagnosticInformation service positive response (fixed length of one byte), with a physical configuration between ECU 1 and test equipment using in ISO15765-2 11bits normal addressing.

C#

```
uds_msgconfig config_physical = new uds_msgconfig();
config_physical.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_1;
config_physical.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config_physical.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config_physical.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config_physical.type = uds_msgtype.PUDS_MSGTYPE_USDT;
```

```

config_physical.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config_physical.nai.target_addr =
    (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config_physical.nai.extension_addr = 0;

uds_msg response_msg = new uds_msg();
uds_status status = UDSApi.MsgAlloc_2013(out response_msg, config_physical, 1);
if (UDSApi.StatusIsOk_2013(status, uds_status.PUDS_STATUS_OK, false))
    MessageBox.Show("Allocate ClearDiagnosticInformation response.", "Success");
else
    MessageBox.Show("Message allocation failed.", "Error");

```

C++ / CLR

```

uds_msgconfig config_physical = {};
config_physical.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_1;
config_physical.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config_physical.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config_physical.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config_physical.type = PUDS_MSGTYPE_USDT;
config_physical.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config_physical.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config_physical.nai.extension_addr = 0;

uds_msg response_msg = {};
uds_status status = UDSApi::MsgAlloc_2013(response_msg, config_physical, 1);
if (UDSApi::StatusIsOk_2013(status, PUDS_STATUS_OK, false))
    MessageBox::Show("Allocate ClearDiagnosticInformation response.", "Success");
else
    MessageBox::Show("Message allocation failed.", "Error");

```

Visual Basic

```

Dim config_physical As uds_msgconfig = New uds_msgconfig()
config_physical.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_1
config_physical.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config_physical.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config_physical.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config_physical.type = uds_msgtype.PUDS_MSGTYPE_USDT
config_physical.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config_physical.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config_physical.nai.extension_addr = 0

Dim response_msg As uds_msg = New uds_msg()
Dim status As uds_status
status = UDSApi.MsgAlloc_2013(response_msg, config_physical, 1)
If UDSApi.StatusIsOk_2013(status, uds_status.PUDS_STATUS_OK, False) Then
    MessageBox.Show("Allocate ClearDiagnosticInformation response.", "Success")
Else
    MessageBox.Show("Message allocation failed.", "Error")
End If

```

Pascal OO

```

var
    config_physical: uds_msgconfig;
    response_msg: uds_msg;
    status: uds_status;
begin
    FillChar(config_physical, sizeof(config_physical), 0);
    config_physical.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_1);
    config_physical.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;

```

```

config_physical.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config_physical.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config_physical.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;
config_physical.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config_physical.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
config_physical.nai.extension_addr := 0;

FillChar(response_msg, sizeof(response_msg), 0);
status := TUDSApi.MsgAlloc_2013(response_msg, config_physical, 1);
if TUDSApi.StatusIsOk_2013(status, uds_status.PUDS_STATUS_OK, false) then
begin
    MessageBox(0, 'Allocate ClearDiagnosticInformation response.',
        'Success', MB_OK);
end
else
begin
    MessageBox(0, 'Message allocation failed.', 'Error', MB_OK);
end;
end;

```

See also: [MsgFree_2013](#) on page 214.

Plain function version: [UDS_MsgAlloc_2013](#) on page 643.

3.7.31 MsgFree_2013

Deallocates a PUDS message.

Syntax

Pascal OO

```

class function MsgFree_2013(
    var msg_buffer: uds_msg
): uds_status;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_MsgFree_2013")]
public static extern uds_status MsgFree_2013(
    ref uds_msg msg_buffer);

```

C++ / CLR

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_MsgFree_2013")]
static uds_status MsgFree_2013(
    uds_msg %msg_buffer);

```

Visual Basic

```

Public Shared Function MsgFree_2013(
    ByRef msg_buffer As uds_msg) As uds_status
End Function

```

Parameters

| Parameter | Description |
|------------|--|
| msg_buffer | An allocated uds_msg structure buffer. |


Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|--|
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The uds_msg structure is invalid. |
| <code>PUDS_STATUS_CAUTION_BUFFER_IN_USE</code> | The message structure is currently in use. It cannot be deleted. |

Example

The following example shows the use of the method `MsgFree_2013`. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the message was already allocated.

C#

```
uds_status result;
result = UDSApi.MsgFree_2013(ref msg);
if (!UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Free message error", "Error");
else
    MessageBox.Show("Message released", "Success");
```

C++ / CLR

```
uds_status result;
result = UDSApi::MsgFree_2013(msg);
if (!UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Free message error", "Error");
else
    MessageBox::Show("Message released", "Success");
```

Visual Basic

```
Dim result As uds_status
result = UDSApi.MsgFree_2013(msg)
If Not UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Free message error", "Error")
Else
    MessageBox.Show("Message released", "Success")
End If
```

Pascal OO

```
var
    result: uds_status;
begin
    result := TUDSApi.MsgFree_2013(msg);
    if not TUDSApi.StatusIsOk_2013(result) then
    begin
        MessageBox(0, 'Free message error', 'Error', MB_OK);
    end
    else
    begin
        MessageBox(0, 'Message released', 'Success', MB_OK);
    end;
end;
```

See also: [uds_msg](#) on page 21, [MsgAlloc_2013](#) on page 211.

Plain function version: [UDS_MsgFree_2013](#) on page 645.

3.7.32 MsgCopy_2013

Copies a PUDS message to another buffer.

Syntax

Pascal OO

```
class function MsgCopy_2013(
    var msg_buffer_dst: uds_msg;
    const msg_buffer_src: Puds_msg
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_MsgCopy_2013")]
public static extern uds_status MsgCopy_2013(
    out uds_msg msg_buffer_dst,
    [In]ref uds_msg msg_buffer_src
);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_MsgCopy_2013")]
static uds_status MsgCopy_2013(
    uds_msg %msg_buffer_dst,
    [In] uds_msg %msg_buffer_src
);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_MsgCopy_2013")>
Public Shared Function MsgCopy_2013(
    ByRef msg_buffer_dst As uds_msg,
    ByRef msg_buffer_src As uds_msg) As uds_status
End Function
```

Parameters

| Parameter | Description |
|----------------|---|
| msg_buffer_dst | A uds_msg structure buffer to store the copied message. |
| msg_buffer_src | The uds_msg structure to copy. |

Returns

The return value is a [uds_status](#) code. [PUDS_STATUS_OK](#) is returned on success. The typical errors in case of failure are:

| | |
|---|---|
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the uds_msg structure is invalid. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see MsgFree_2013 on page 214). |

Remarks

When a message is copied, a new buffer is allocated. Once processed, user has to release the source and the destination messages (see [MsgFree_2013](#) on page 214).

Example

The following example shows the use of the method `MsgCopy_2013`. Depending on the result, a message will be shown to the user.

Note: It is assumed that the source message was already allocated.

C#

```
uds_msg destination = new uds_msg();
uds_status result = UDSApi.MsgCopy_2013(out destination, ref source);
if (!UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Copy error", "Error");
else
    MessageBox.Show("Message copied", "Success");
```

C++ / CLR

```
uds_msg destination = {};
uds_status result = UDSApi::MsgCopy_2013(destination, source);
if (!UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Copy error", "Error");
else
    MessageBox::Show("Message copied", "Success");
```

Visual Basic

```
Dim destination As uds_msg = New uds_msg()
Dim result As uds_status = UDSApi.MsgCopy_2013(destination, source)
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Message copied", "Success")
Else
    MessageBox.Show("Copy error", "Error")
End If
```

Pascal OO

```
var
    destination: uds_msg;
    result: uds_status;
begin
    FillChar(destination, sizeof(destination), 0);
    result := TUDSApi.MsgCopy_2013(destination, @source);
    if TUDSApi.StatusIsOk_2013(result) then
    begin
        MessageBox(0, 'Message copied', 'Success', MB_OK);
    end
    else
    begin
        MessageBox(0, 'Copy error', 'Error', MB_OK);
    end;
end;
```

See also: `uds_msg` on page 21.

Plain function version: `UDS_MsgCopy_2013` on page 645.

3.7.33 MsgMove_2013

Moves a PUDS message to another buffer (and cleans the original message structure).

Syntax

Pascal OO

```
class function MsgMove_2013(
    var msg_buffer_dst: uds_msg;
    var msg_buffer_src: uds_msg
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_MsgMove_2013")]
public static extern uds_status MsgMove_2013(
    out uds_msg msg_buffer_dst,
    ref uds_msg msg_buffer_src
);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_MsgMove_2013")]
static uds_status MsgMove_2013(
    uds_msg %msg_buffer_dst,
    uds_msg %msg_buffer_src
);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_MsgMove_2013")>
Public Shared Function MsgMove_2013(
    ByRef msg_buffer_dst As uds_msg,
    ByRef msg_buffer_src As uds_msg) As uds_status
End Function
```

Parameters

| Parameter | Description |
|----------------|--|
| msg_buffer_dst | A uds_msg structure buffer to store the message. |
| msg_buffer_src | The uds_msg structure buffer used as the source (will be cleaned). |

Remarks

When a message is moved, the source buffer is cleaned: once processed, user only has to release the destination message (see [MsgFree_2013](#) on page 214).


Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---|---|
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the uds_msg structure is invalid. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see MsgFree_2013 on page 214). |

Example

The following example shows the use of the method [MsgMove_2013](#). Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the source message was already allocated.

C#

```
uds_msg destination = new uds_msg();
uds_status result = UDSApi.MsgMove_2013(out destination, ref source);
if (!UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Move error", "Error");
else
    MessageBox.Show("Message moved", "Success");
```

C++ / CLR

```
uds_msg destination = {};
uds_status result = UDSApi::MsgMove_2013(destination, source);
if (!UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Move error", "Error");
else
    MessageBox::Show("Message moved", "Success");
```

Visual Basic

```
Dim destination As uds_msg = New uds_msg()
Dim result As uds_status = UDSApi.MsgMove_2013(destination, source)
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Message moved", "Success")
Else
    MessageBox.Show("Move error", "Error")
End If
```

Pascal OO

```
var
    destination: uds_msg;
    result: uds_status;
begin
    FillChar(destination, sizeof(destination), 0);
    result := TUDSApi.MsgMove_2013(destination, source);
    if TUDSApi.StatusIsOk_2013(result) then
        begin
            MessageBox(0, 'Message moved', 'Success', MB_OK);
        end
    else
        begin
            MessageBox(0, 'Move error', 'Error', MB_OK);
        end;
end;
```

See also: [uds_msg](#) on page 21.



Plain function version: [UDS_MsgMove_2013](#) on page 646.

3.7.34 Read_2013

Reads a CAN UDS message from the receive queue of a PUDS channel.

Overloads

| | Method | Description |
|---|---|---|
|  | Read_2013 (cantp_handle, uds_msg) | Reads a CAN UDS message from the receive queue of a PUDS channel. |

| | | |
|---|---|---|
|  | <code>Read_2013(cantp_handle, uds_msg, uds_msg)</code> | Reads a CAN UDS response (matching the request message) from the receive queue of a PUDS channel. |
|  | <code>Read_2013(cantp_handle, uds_msg, uds_msg, cantp_timestamp)</code> | Reads a CAN UDS response (matching the request message) and its timestamp from the receive queue of a PUDS channel. |

Plain function version: `UDS_Read_2013` on page 648.

3.7.35 Read_2013(cantp_handle, uds_msg)

Reads a CAN UDS message from the receive queue of a PUDS channel.

Syntax

Pascal OO

```
class function Read_2013(
    channel: cantp_handle;
    var out_msg_buffer: uds_msg
): uds_status; overload;
```

C#

```
public static uds_status Read_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    out uds_msg out_msg_buffer);
```

C++ / CLR

```
static uds_status Read_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msg %out_msg_buffer);
```

Visual Basic

```
Public Shared Function Read_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByRef out_msg_buffer As uds_msg) As uds_status
End Function
```

Parameters

| Parameter | Description |
|----------------|--|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| out_msg_buffer | A <code>uds_msg</code> buffer to store the received message (see <code>uds_msg</code> on page 21). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|--|
| <code>PUDS_STATUS_NO_MESSAGE</code> | Indicates that the receive queue of the channel is empty. |
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

- In addition to checking `uds_status` code, the `cantp_netstatus` should be checked as it contains the network status of the message (see field `msg.msgdata.any->netstatus` in `uds_msg` on page 21).
- In case of ISO-TP message, the message type contained in the message `cantp_netaddrinfo` which indicates if the message is a complete ISO-TP message (without pending message flag) should be checked too (see field `msg.msgdata.isotp->netaddrinfo` in `uds_msg` on page 21).
- The message structure is automatically allocated and initialized in `Read_2013` method. So once the message processed, the structure must be released (see `MsgFree_2013` on page 214).

Note: That higher level methods like `WaitForService_2013` or `WaitForServiceFunctional_2013` should be preferred in cases where a client just has to read the response from a service request.

Example

The following example shows the use of the method `Read_2013` on the channel `PCANTP_HANDLE_USBBUS1`. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized. This example is basic, the preferred way to read UDS messages is Using Events (see on page 777).

C#

```
uds_status result;
uds_msg msg = new uds_msg();
bool stop = false;

do
{
    // Read the first message in the queue
    result = UDSApi.Read_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, out msg);
    if (UDSApi.StatusIsOk_2013(result))
    {
        MessageBox.Show("A message was received", "Success");

        // Process the received message
        // ProcessMessage(msg);

        result = UDSApi.MsgFree_2013(ref msg);
        if (!UDSApi.StatusIsOk_2013(result))
            MessageBox.Show("Message free error", "Error");
    }
    else
    {
        // An error occurred
        MessageBox.Show("An error occurred", "Error");
        // Here can be decided if the loop must be terminated
        // stop = HandleReadError(result);
    }
} while (!stop);
```

C++ / CLR

```
uds_status result;
uds_msg msg = {};
bool stop = false;

do
{
    // Read the first message in the queue
```

```

result = UDSApi::Read_2013(PCANTP_HANDLE_USBBUS1, msg);
if (UDSApi::StatusIsOk_2013(result))
{
    MessageBox::Show("A message was received", "Success");

    // Process the received message
    // ProcessMessage(msg);

    result = UDSApi::MsgFree_2013(msg);
    if (!UDSApi::StatusIsOk_2013(result))
        MessageBox::Show("Message free error", "Error");
}
else
{
    // An error occurred
    MessageBox::Show("An error occurred", "Error");
    // Here can be decided if the loop has must terminated
    // stop = HandleReadError(result);
}
} while (!stop);

```

Visual Basic

```

Dim result As uds_status
Dim msg As uds_msg = New uds_msg()
Dim stop_loop As Boolean = False

Do

    ' Read the first message in the queue
    result = UDSApi.Read_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, msg)
    If UDSApi.StatusIsOk_2013(result) Then
        MessageBox.Show("A message was received", "Success")

        ' Process the received message
        ' ProcessMessage(msg)

        result = UDSApi.MsgFree_2013(msg)
        If Not UDSApi.StatusIsOk_2013(result) Then
            MessageBox.Show("Message free error", "Error")
        End If
    Else

        ' An error occurred
        MessageBox.Show("An error occurred", "Error")
        ' Here can be decided if the loop has must terminated
        ' stop_loop = HandleReadError(result)
    End If
Loop While Not stop_loop

```

Pascal OO

```

var
    result: uds_status;
    msg: uds_msg;
    stop: bool;
begin
    FillChar(msg, sizeof(msg), 0);
    stop := false;
    repeat

        // Read the first message in the queue

```

```

result := TUDSApi.Read_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, msg);
if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'A message was received', 'Success', MB_OK);

    // Process the received message
    // ProcessMessage(msg);

    result := TUDSApi.MsgFree_2013(msg);
    if not TUDSApi.StatusIsOk_2013(result) then
    begin
        MessageBox(0, 'Message free error', 'Error', MB_OK);
    end;
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);

    // Here can be decided if the loop has must terminated
    // stop := HandleReadError(result);
end;
until stop;
end;

```

See also: [Write_2013](#) on page 229, [uds_msg](#) on page 21, [UUDT Read/Write Example](#) on page 775.

Plain function version: [UDS_Read_2013](#) on page 648.

3.7.36 Read_2013(cantp_handle, uds_msg, uds_msg)

Reads a CAN UDS response (matching the given request message) from the receive queue of a PUDS channel.

Syntax

Pascal OO

```

class function Read_2013(
    channel: cantp_handle;
    var out_msg_buffer: uds_msg;
    in_msg_request: Puds_msg
): uds_status; overload;

```

C#

```

public static uds_status Read_2013(
    cantp_handle channel,
    out uds_msg out_msg_buffer,
    ref uds_msg in_msg_request);

```

C++ / CLR

```

static uds_status Read_2013(
    cantp_handle channel,
    uds_msg %out_msg_buffer,
    uds_msg %in_msg_request);

```

Visual Basic

```
Public Shared Function Read_2013(  
    ByVal channel As cantp_handle,  
    ByRef out_msg_buffer As uds_msg,  
    ByRef in_msg_request As uds_msg) As uds_status  
End Function
```

Parameters

| Parameter | Description |
|----------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| out_msg_buffer | A uds_msg buffer to store the received message (see uds_msg on page 21). |
| in_msg_request | Optional, request message which contains the network address information to filter in order to read the response (see uds_msg on page 21). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|--|
| <code>PUDS_STATUS_NO_MESSAGE</code> | Indicates that the receive queue of the channel is empty. |
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

- In addition to checking `uds_status` code, the `cantp_netstatus` should be checked as it contains the network status of the message (see field `msg.msgdata.any->netstatus` in `uds_msg` on page 21).
- In case of ISO-TP message, the message type contained in the message `cantp_netaddrinfo` which indicates if the message is a complete ISO-TP message (no pending message flag) should be checked too (see field `msg.msgdata.isotp->netaddrinfo` in `uds_msg` on page 21).
- The message structure is automatically allocated and initialized in `Read_2013` method. So once the message processed, the structure must be released (see `MsgFree_2013` on page 214).

Note: That higher level methods like `WaitForService_2013` or `WaitForServiceFunctional_2013` should be preferred in cases where a client just has to read the response from a service request.

Example

The following example shows the use of the method `Read_2013` on the channel `PCANTP_HANDLE_USBBUS1`. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized, and a request message has been sent.

C#

```
uds_status result;  
uds_msg msg = new uds_msg();  
bool stop = false;  
  
do  
{  
    // Read the first message in the queue  
    result = UDSApi.Read_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, out msg, ref request_msg);  
    if (UDSApi.StatusIsOk_2013(result))  
    {  
        MessageBox.Show("A message was received", "Success");  
    }  
}
```



```

        result = UDSApi.MsgFree_2013(ref msg);
        if (!UDSApi.StatusIsOk_2013(result))
            MessageBox.Show("Message free error", "Error");
    }
    else
    {
        // Here can be decided if the loop must be terminated
        // stop = HandleReadError(result);
    }
} while (!stop);

```

C++ / CLR

```

uds_status result;
uds_msg msg = {};
bool stop = false;

do
{
    // Read the first message in the queue
    result = UDSApi::Read_2013(PCANTP_HANDLE_USBBUS1, msg, request_msg);
    if (UDSApi::StatusIsOk_2013(result))
    {
        MessageBox::Show("A message was received", "Success");
        result = UDSApi::MsgFree_2013(msg);
        if (!UDSApi::StatusIsOk_2013(result))
            MessageBox::Show("Message free error", "Error");
    }
    else
    {
        // Here can be decided if the loop has must terminated
        // stop = HandleReadError(result);
    }
} while (!stop);

```

Visual Basic

```

Dim result As uds_status
Dim msg As uds_msg = New uds_msg()
Dim stop_loop As Boolean = False

Do

    ' Read the first message in the queue
    result = UDSApi.Read_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, msg, request_msg)
    If UDSApi.StatusIsOk_2013(result) Then
        MessageBox.Show("A message was received", "Success")
        result = UDSApi.MsgFree_2013(msg)
        If Not UDSApi.StatusIsOk_2013(result) Then
            MessageBox.Show("Message free error", "Error")
        End If
    Else
        ' Here can be decided if the loop has must terminated
        ' stop_loop = HandleReadError(result)
    End If
Loop While Not stop_loop

```

Pascal OO

```

var
    result: uds_status;
    msg: uds_msg;
    stop_loop: bool;

```

```

begin
  FillChar(msg, sizeof(msg), 0);
  stop_loop := false;
  repeat
    // Read the first message in the queue
    result := TUDSApi.Read_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, msg,
      @request_msg);
    if TUDSApi.StatusIsOk_2013(result) then
      begin
        MessageBox(0, 'A message was received', 'Success', MB_OK);
        result := TUDSApi.MsgFree_2013(msg);
        if not TUDSApi.StatusIsOk_2013(result) then
          begin
            MessageBox(0, 'Message free error', 'Error', MB_OK);
          end;
        end;
      else
        begin
          // Here can be decided if the loop has must terminated
          // stop_loop := HandleReadError(result);
        end;
      until stop_loop;
    end;
  end;
end;

```

See also: [Write_2013](#) on page 229, [uds_msg](#) on page 21, UUDT Read/Write Example on page 775.

Plain function version: [UDS_Read_2013](#) on page 648.

3.7.37 Read_2013(cantp_handle, uds_msg, uds_msg, cantp_timestamp)

Reads a CAN UDS response (matching the given request message) and its timestamp from the receive queue of a PUDS channel.

Syntax

Pascal OO

```

class function Read_2013(
  channel: cantp_handle;
  var out_msg_buffer: uds_msg;
  in_msg_request: Puds_msg;
  out_timestamp: Pcantp_timestamp
): uds_status; overload;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_Read_2013")]
public static extern uds_status Read_2013(
  [MarshalAs(UnmanagedType.U4)]
  cantp_handle channel,
  out uds_msg out_msg_buffer,
  [In] ref uds_msg in_msg_request,
  out cantp_timestamp out_timestamp);

```

C++ / CLR

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_Read_2013")]
static uds_status Read_2013(
  [MarshalAs(UnmanagedType.U4)]
  cantp_handle channel,
  uds_msg %out_msg_buffer,
  [In] uds_msg %in_msg_request,

```

```
cantp_timestamp %out_timestamp);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_Read_2013")>
Public Shared Function Read_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByRef out_msg_buffer As uds_msg,
    ByRef in_msg_request As uds_msg,
    ByRef out_timestamp As cantp_timestamp) As uds_status
End Function
```

Parameters

| Parameter | Description |
|----------------|---|
| Channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| out_msg_buffer | A <code>uds_msg</code> buffer to store the received message (see <code>uds_msg</code> on page 21). |
| in_msg_request | Optional, request message which contains the network address information to filter in order to read the response (see <code>uds_msg</code> on page 21). |
| out_timestamp | A <code>cantp_timestamp</code> structure buffer to get the reception time of the message. If this value is not desired, this parameter should be passed as NULL (see <code>cantp_timestamp</code> on page 104). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|--|
| <code>PUDS_STATUS_NO_MESSAGE</code> | Indicates that the receive queue of the channel is empty. |
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

- In addition to checking `uds_status` code, the `cantp_netstatus` should be checked as it contains the network status of the message (see field `msg.msgdata.any->netstatus` in `uds_msg` on page 21).
- In case of ISO-TP message, the message type contained in the message `cantp_netaddrinfo` which indicates if the message is a complete ISO-TP message (no pending message flag) should be checked too (see field `msg.msgdata.isotp->netaddrinfo` in `uds_msg` on page 21).
- Specifying the value of `NULL` for the parameter `out_timestamp` causes reading a message without timestamp when the reception time is not desired.
- The message structure is automatically allocated and initialized in `Read_2013` method. So once the message processed, the structure must be uninitialized (see `MsgFree_2013` on page 214).

Note: Higher level methods like `WaitForService_2013` or `WaitForServiceFunctional_2013` should be preferred in cases where a client just has to read the response from a service request.

Example

The following example shows the use of the method `Read_2013` on the channel `PCANTP_HANDLE_USBBUS1`. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized, and a request message has been sent.

C#

```
uds_status result;
```

```

uds_msg msg = new uds_msg();
bool stop = false;
cantp_timestamp timestamp;
do
{
    // Read the first message in the queue

    result = UDSApi.Read_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, out msg, ref request_msg, out
        timestamp);
    if (UDSApi.StatusIsOk_2013(result))
    {
        MessageBox.Show("A message was received at t=" + timestamp, "Success");
        result = UDSApi.MsgFree_2013(ref msg);
        if (!UDSApi.StatusIsOk_2013(result))
            MessageBox.Show("Message free error", "Error");
    }
    else
    {
        // Here can be decided if the loop must be terminated
        // stop = HandleReadError(result);
    }
} while (!stop);

```

C++ / CLR

```

uds_status result;
uds_msg msg = {};
bool stop = false;
cantp_timestamp timestamp;
do
{
    // Read the first message in the queue

    result = UDSApi::Read_2013(PCANTP_HANDLE_USBBUS1, msg, request_msg, timestamp);
    if (UDSApi::StatusIsOk_2013(result))
    {
        MessageBox::Show("A message was received at t=" + timestamp, "Success");
        result = UDSApi::MsgFree_2013(msg);
        if (!UDSApi::StatusIsOk_2013(result))
            MessageBox::Show("Message free error", "Error");
    }
    else
    {
        // Here can be decided if the loop must be terminated
        // stop = HandleReadError(result);
    }
} while (!stop);

```

Visual Basic

```

Dim result As uds_status
Dim msg As uds_msg = New uds_msg()
Dim stop_loop As Boolean = False
Dim timestamp As cantp_timestamp
Do
    ' Read the first message in the queue
    result = UDSApi.Read_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, msg, request_msg, timestamp)
    If UDSApi.StatusIsOk_2013(result) Then
        MessageBox.Show("A message was received at t=" + timestamp.ToString(), "Success")
        result = UDSApi.MsgFree_2013(msg)
        If Not UDSApi.StatusIsOk_2013(result) Then
            MessageBox.Show("Message free error", "Error")
        End If
    End If

```

```

        End If
    Else
        ' Here can be decided if the loop must be terminated
        ' stop_loop = HandleReadError(result)
    End If
Loop While Not stop_loop

```

Pascal OO

```

var
    result: uds_status;
    msg: uds_msg;
    stop_loop: bool;
    timestamp: cantp_timestamp;
begin
    FillChar(msg, sizeof(msg), 0);
    stop_loop := false;
    repeat

        // Read the first message in the queue
        result := TUDSApi.Read_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, msg,
            @request_msg, @timestamp);
        if TUDSApi.StatusIsOk_2013(result) then
            begin
                MessageBox(0, PWideChar(format('A message was received at t=%d',
                    [UInt64(timestamp)])), 'Success', MB_OK);
                result := TUDSApi.MsgFree_2013(msg);
                if not TUDSApi.StatusIsOk_2013(result) then
                    begin
                        MessageBox(0, 'Message free error', 'Error', MB_OK);
                    end;
                end
            else
                begin
                    // Here can be decided if the loop must be terminated
                    // stop_loop := HandleReadError(result);
                end;
            until stop_loop;
        end;
end;

```

See also: [Write_2013](#) on page 229, [uds_msg](#) on page 21, UUDT Read/Write Example on page 775.
 Plain function version: [UDS_Read_2013](#) on page 648.

3.7.38 write_2013

Transmits a PUDS message using a connected PUDS channel.

Syntax

Pascal OO

```

class function Write_2013(
    channel: cantp_handle;
    msg_buffer: Puds_msg
): uds_status;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_Write_2013")]
public static extern uds_status Write_2013(
    [MarshalAs(UnmanagedType.U4)]

```

```
cantp_handle channel,
ref uds_msg msg_buffer);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_Write_2013")]
static uds_status Write_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msg %msg_buffer);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_Write_2013")>
Public Shared Function Write_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByRef msg_buffer As uds_msg) As uds_status
End Function
```

Parameters

| Parameter | Description |
|------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| msg_buffer | A uds_msg containing the message to be sent (see uds_msg on page 21). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of initialized channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The message is not a valid PUDS message. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_PARAM_INVALID_TYPE</code> | The message type is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping is unknown. |

Remarks

The `Write_2013` method does not actually send the UDS message, the transmission is asynchronous. Should a message fail to be transmitted, it will be added to the reception queue with a specific network error code in the `msg.msgdata.isotp->netaddrinfo` field of the `uds_msg`.

Note: To transmit a standard UDS service request, it is recommended to use the corresponding API Service method starting with `Svc` (like `SvcDiagnosticSessionControl_2013`).

Example

The following example shows the use of the method `Write_2013` on the channel `PCANTP_HANDLE_USBBUS1`. It adds to the transmit queue a UDS request then waits until a confirmation message is received. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized, mapping, message configuration and receive event were configured, the content of the `data` field is not initialized in the example.

C#

```
uds_status result;
```

```

uds_msg request_msg = new uds_msg();
uds_msg loopback_msg = new uds_msg();
bool wait_result;

// Allocate and initialize message structure
result = UDSApi.MsgAlloc_2013(out request_msg, config, 4095);
if (!UDSApI.StatusIsOk_2013(result))
{
    MessageBox.Show("Message initialization error: " + result, "Error");
}
else
{
    // The message is sent using the PCAN-USB.
    result = UDSApi.Write_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request_msg);
    if (UDSApI.StatusIsOk_2013(result))
    {
        // Read the transmission confirmation.
        wait_result = receive_event.WaitOne(5000);
        if (wait_result)
        {
            result = UDSApi.Read_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, out loopback_msg);
            if (UDSApI.StatusIsOk_2013(result))
            {
                MessageBox.Show("Read = " + result + ", type=" + loopback_msg.type +
                    ", netstatus=" + loopback_msg.msg.Msgdata_any_Copy.netstatus, "Read");
                UDSApi.MsgFree_2013(ref loopback_msg);
            }
            else
            {
                MessageBox.Show("Read error: " + result, "Error");
            }
        }
        UDSApi.MsgFree_2013(ref request_msg);
    }
    else
    {
        MessageBox.Show("Write error: " + result, "Error");
    }
}
}

```

C++ / CLR

```

uds_status result;
uds_msg request_msg = {};
uds_msg loopback_msg = {};
bool wait_result;

// Allocate and initialize message structure
result = UDSApi::MsgAlloc_2013(request_msg, config, 4095);
if (!UDSApI::StatusIsOk_2013(result))
{
    MessageBox::Show(String::Format("Message initialization error: {0}", (int)result),
        "Error");
}
else
{
    // The message is sent using the PCAN-USB.
    result = UDSApi::Write_2013(PCANTP_HANDLE_USBBUS1, request_msg);
    if (UDSApI::StatusIsOk_2013(result))
    {
        // Read the transmission confirmation.
        wait_result = receive_event.WaitOne(5000);
    }
}

```

```

    if (wait_result)
    {
        result = UDSApi::Read_2013(PCANTP_HANDLE_USBBUS1, loopback_msg);
        if (UDSApI::StatusIsOk_2013(result))
        {
            MessageBox::Show(String::Format(
                "Read = {0}, type={1}, netstatus={2}", (int)result,
                (int)loopback_msg.type,
                (int)loopback_msg.msg.msgdata.any->netstatus), "Read");
            UDSApi::MsgFree_2013(loopback_msg);
        }
        else
        {
            MessageBox::Show(String::Format("Read error: {0}", (int)result),
                "Error");
        }
    }
    UDSApi::MsgFree_2013(request_msg);
}
else
{
    MessageBox::Show(String::Format("Write error: {0}", (int)result), "Error");
}
}

```

Visual Basic

```

Dim result As uds_status
Dim request_msg As uds_msg = New uds_msg()
Dim loopback_msg As uds_msg = New uds_msg()
Dim wait_result As Boolean

' Allocate and initialize message structure
result = UDSApi.MsgAlloc_2013(request_msg, config, 4095)
If Not UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Message initialization error: " + result.ToString(), "Error")
Else
    ' The message is sent using the PCAN-USB.
    result = UDSApi.Write_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request_msg)
    If UDSApi.StatusIsOk_2013(result) Then
        ' Read the transmission confirmation.
        wait_result = receive_event.WaitOne(5000)
        If wait_result Then
            result = UDSApi.Read_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, loopback_msg)
            If UDSApi.StatusIsOk_2013(result) Then
                MessageBox.Show("Read = " + result.ToString() + ", type=" +
                    loopback_msg.type.ToString() + ", netstatus=" +
                    loopback_msg.msg.Msgdata_any_Copy.netstatus.ToString(), "Read")
                UDSApi.MsgFree_2013(loopback_msg)
            Else
                MessageBox.Show("Read error: " + result.ToString(), "Error")
            End If
        End If
        UDSApi.MsgFree_2013(request_msg)
    Else
        MessageBox.Show("Write error: " + result.ToString(), "Error")
    End If
End If

```

Pascal OO

```
var
```



```

result: uds_status;
request_msg: uds_msg;
loopback_msg: uds_msg;
wait_result: UInt32;
begin
  FillChar(request_msg, sizeof(request_msg), 0);
  FillChar(loopback_msg, sizeof(loopback_msg), 0);

  // Allocate and initialize message structure
  result := TUDSApi.MsgAlloc_2013(request_msg, config, 4095);
  if not TUDSApi.StatusIsOk_2013(result) then
    begin
      MessageBox(0, PWideChar(format('Message initialization error: %d',
        [Integer(result)])), 'Error', MB_OK);
    end
  else
    begin

      // The message is sent using the PCAN-USB.
      result := TUDSApi.Write_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request_msg);
      if TUDSApi.StatusIsOk_2013(result) then
        begin

          // Read the transmission confirmation.
          wait_result := WaitForSingleObject(receive_event, 5000);
          if wait_result = WAIT_OBJECT_0 then
            begin
              result := TUDSApi.Read_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
                loopback_msg);
              if TUDSApi.StatusIsOk_2013(result) then
                begin
                  MessageBox(0, PWideChar(format('Read=%d, type=%d, netstatus=%d',
                    [Integer(result), Integer(loopback_msg.typem),
                    Integer(loopback_msg.msg.msgdata_any.netstatus)])), 'Read', MB_OK);
                  TUDSApi.MsgFree_2013(loopback_msg);
                end
              else
                begin
                  MessageBox(0, PWideChar(format('Read error: %d', [Integer(result)])),
                    'Error', MB_OK);
                end;
              end;
            end;
            TUDSApi.MsgFree_2013(request_msg);
          end
        else
          begin
            MessageBox(0, PWideChar(format('Write error: %d', [Integer(result)])),
              'Error', MB_OK);
          end;
        end;
      end;
    end;
  end;
end;

```

See also: [Read_2013](#) on page 219, UUDT Read/Write Example on page 775.

Plain function version: [UDS_Write_2013](#) on page 649.

3.7.39 Reset_2013

Resets the receive and transmit queues of a PUDS channel.

Syntax

Pascal OO

```
class function Reset_2013(
    channel: cantp_handle
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_Reset_2013")]
public static extern uds_status Reset_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_Reset_2013")]
static uds_status Reset_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_Reset_2013")>
Public Shared Function Reset_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical error in case of failure is:


| | |
|-----------------------------|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of initialized channels of the calling application. |
|-----------------------------|---|

Remarks

Calling this method ONLY clear the queues of a channel. A reset of the CAN controller does not take place.

Example

The following example shows the use of the method `Reset_2013` on the channel `PCANTP_HANDLE_PCIBUS1`. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;

result = UDSApi.Reset_2013(cantp_handle.PCANTP_HANDLE_PCIBUS1);
if (!UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("An error occurred", "Error");
else
    MessageBox.Show("PCAN-PCI (Ch-1) was reset", "Success");
```

C++ / CLR

```
uds_status result;

result = UDSApi::Reset_2013(PCANTP_HANDLE_PCIBUS1);
if (!UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("An error occurred", "Error");
else
    MessageBox::Show("PCAN-PCI (Ch-1) was reset", "Success");
```

Visual Basic

```
Dim result As uds_status

result = UDSApi.Reset_2013(cantp_handle.PCANTP_HANDLE_PCIBUS1)
If Not UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("An error occurred", "Error")
Else
    MessageBox.Show("PCAN-PCI (Ch-1) was reset", "Success")
End If
```

Pascal OO

```
var
    result: uds_status;
begin
    result := TUDSApi.Reset_2013(cantp_handle.PCANTP_HANDLE_PCIBUS1);
    if not TUDSApi.StatusIsOk_2013(result) then
        begin
            MessageBox(0, 'An error occurred', 'Error', MB_OK);
        end
    else
        begin
            MessageBox(0, 'PCAN-PCI (Ch-1) was reset', 'Success', MB_OK);
        end;
end;
```

See also: [Uninitialize_2013](#) on page 151.

Plain function version: [UDS_Reset_2013](#) on page 651.

3.7.40 WaitForSingleMessage_2013

Waits for a message (a response or a transmit confirmation) based on a PUDS message request.

Syntax**Pascal OO**

```
class function WaitForSingleMessage_2013(
    channel: cantp_handle;
    msg_request: Puds_msg;
```

```
is_waiting_for_tx: Boolean;
timeout: UInt32;
timeout_enhanced: UInt32;
var out_msg_response: uds_msg
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_WaitForSingleMessage_2013")]
public static extern uds_status WaitForSingleMessage_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    [In] ref uds_msg msg_request,
    [MarshalAs(UnmanagedType.U1)]
    bool is_waiting_for_tx,
    UInt32 timeout,
    UInt32 timeout_enhanced,
    out uds_msg out_msg_response);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_WaitForSingleMessage_2013")]
static uds_status WaitForSingleMessage_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    [In] uds_msg %msg_request,
    [MarshalAs(UnmanagedType::U1)]
    bool is_waiting_for_tx,
    UInt32 timeout,
    UInt32 timeout_enhanced,
    uds_msg %out_msg_response);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_WaitForSingleMessage_2013")>
Public Shared Function WaitForSingleMessage_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByRef msg_request As uds_msg,
    <MarshalAs(UnmanagedType.U1)>
    ByVal is_waiting_for_tx As Boolean,
    ByVal timeout As UInt32,
    ByVal timeout_enhanced As UInt32,
    ByRef out_msg_response As uds_msg) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-------------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| msg_request | A uds_msg buffer containing the PUDS message request that was previously sent (see uds_msg on page 21). |
| is_waiting_for_tx | States whether the message to wait for is a transmit confirmation or not. |
| timeout | Maximum time to wait (in milliseconds) for a message indication corresponding to the message request. Note: A zero value means unlimited time. |
| timeout_enhanced | Maximum time to wait (in milliseconds) for a message to be complete if server (ECU) ask more time. |
| out_msg_response | A uds_msg buffer to store the PUDS response message (see uds_msg on page 21). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---|--|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of initialized channels of the calling application. |
| <code>PUDS_STATUS_SERVICE_TIMEOUT_CONFIRMATION</code> | Timeout while waiting for request confirmation (request message loopback). |
| <code>PUDS_STATUS_SERVICE_TIMEOUT_RESPONSE</code> | Timeout while waiting for response message. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is invalid. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks


The criteria to identify if a response matches the message request is based only on the network addressing information and the UDS service identifier: if the same service is requested multiple times with different parameters (like service `ReadDataByIdentifier` with different Data IDs), the user will have to ensure that the extra content matches the original request.

The timeout parameter is ignored once a message indication matching the request is received (i.e. the first frame of the message), the timeout enhanced is then used.

The parameters of `WaitFor*_2013` methods have a new order. They do not keep the order of the previous version.

Example

The following example shows the use of the method `WaitForSingleMessage_2013` on the channel `PCANTP_HANDLE_USBBUS1`. It writes a UDS message on the CAN Bus and waits for the confirmation of the transmission. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel and the mapping were already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg confirmation = new uds_msg();

// Prepare an 11bit CAN ID, physically addressed UDS message containing 4 Bytes of data
uds_msgconfig config = new uds_msgconfig();
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

result = UDSApi.MsgAlloc_2013(out request, config, 4);
if (!UDSApi.StatusIsOk_2013(result))
{
    MessageBox.Show("Error occurred while allocating request message.", "Error");
}
else
{
    // The message is sent using the PCAN-USB
    result = UDSApi.Write_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request);
}

```

```

if (UDSApi.StatusIsOk_2013(result))
{
    // Wait for the transmit confirmation
    result = UDSApi.WaitForSingleMessage_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref
        request, true, 10, 100, out confirmation);
    if (UDSApi.StatusIsOk_2013(result))
    {
        MessageBox.Show("Message was transmitted.", "Success");
        UDSApi.MsgFree_2013(ref confirmation);
    }
    else
    {
        // An error occurred
        MessageBox.Show("Error occurred while waiting for transmit confirmation.",
            "Error");
    }
    UDSApi.MsgFree_2013(ref request);
}
else
{
    // An error occurred
    MessageBox.Show("An error occurred", "Error");
}
}

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg confirmation = {};

// Prepare an 11bit CAN ID, physically addressed UDS message containing 4 Bytes of data
uds_msgconfig config = {};
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

result = UDSApi::MsgAlloc_2013(request, config, 4);
if (!UDSApi::StatusIsOk_2013(result))
{
    MessageBox::Show("Error occurred while allocating request message.", "Error");
}
else
{
    // The message is sent using the PCAN-USB
    result = UDSApi::Write_2013(PCANTP_HANDLE_USBBUS1, request);
    if (UDSApi::StatusIsOk_2013(result))
    {
        // Wait for the transmit confirmation
        result = UDSApi::WaitForSingleMessage_2013(PCANTP_HANDLE_USBBUS1, request, true,
            10, 100, confirmation);
        if (UDSApi::StatusIsOk_2013(result))
        {
            MessageBox::Show("Message was transmitted.", "Success");
            UDSApi::MsgFree_2013(confirmation);
        }
        else
    }
}

```

```

        {
            // An error occurred
            MessageBox::Show("Error occurred while waiting for transmit confirmation.",
                             "Error");
        }
        UDSApi::MsgFree_2013(request);
    }
else
    {
        // An error occurred
        MessageBox::Show("An error occurred", "Error");
    }
}

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim confirmation As uds_msg = New uds_msg()

' Prepare an 11bit CAN ID, physically addressed UDS message containing 4 Bytes of data
Dim config As uds_msgconfig = New uds_msgconfig()
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = 0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

result = UDSApi.MsgAlloc_2013(request, config, 4)
If Not UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Error occurred while allocating request message.", "Error")
Else
    ' The message is sent using the PCAN-USB
    result = UDSApi.Write_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request)
    If UDSApi.StatusIsOk_2013(result) Then
        ' Wait for the transmit confirmation
        result = UDSApi.WaitForSingleMessage_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request,
            True, 10, 100, confirmation)
        If UDSApi.StatusIsOk_2013(result) Then
            MessageBox.Show("Message was transmitted.", "Success")
            UDSApi.MsgFree_2013(confirmation)
        Else
            ' An error occurred
            MessageBox.Show("Error occurred while waiting for transmit confirmation.", "Error")
        End If
        UDSApi.MsgFree_2013(request)
    Else
        ' An error occurred
        MessageBox.Show("An error occurred", "Error")
    End If
End If

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    confirmation: uds_msg;
    config: uds_msgconfig;

```

```

begin
  FillChar(request, sizeof(request), 0);
  FillChar(confirmation, sizeof(confirmation), 0);

  // Prepare an 11bit CAN ID, physically addressed UDS message containing 4 Bytes of data
  FillChar(config, sizeof(config), 0);
  config.can_id :=
    UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
  config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
  config.nai.extension_addr := 0;
  config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
  config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
  config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
  config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
  config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

  result := TUDSApi.MsgAlloc_2013(request, config, 4);
  if not TUDSApi.StatusIsOk_2013(result) then
    begin
      MessageBox(0, 'Error occurred while allocating request message.',
        'Error', MB_OK);
    end
  else
    begin
      // The message is sent using the PCAN-USB
      result := TUDSApi.Write_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, @request);
      if TUDSApi.StatusIsOk_2013(result) then
        begin
          // Wait for the transmit confirmation
          result := TUDSApi.WaitForSingleMessage_2013
            (cantp_handle.PCANTP_HANDLE_USBBUS1, @request, true, 10, 100,
              confirmation);
          if TUDSApi.StatusIsOk_2013(result) then
            begin
              MessageBox(0, 'Message was transmitted.', 'Success', MB_OK);
              TUDSApi.MsgFree_2013(confirmation);
            end
          else
            begin
              // An error occurred
              MessageBox(0, 'Error occurred while waiting for transmit confirmation.',
                'Error', MB_OK);
            end;
            TUDSApi.MsgFree_2013(request);
          end
        else
          begin
            // An error occurred
            MessageBox(0, 'An error occurred', 'Error', MB_OK);
          end;
        end;
      end;
    end;
  end;
end;

```

See also: [WaitForService_2013](#) on page 248, [WaitForServiceFunctional_2013](#) on page 252,
[WaitForFunctionalResponses_2013](#) on page 241.

Plain function version: [UDS_WaitForSingleMessage_2013](#) on page 652.

3.7.41 WaitForFunctionalResponses_2013

Waits for multiple messages (multiple responses from a functional request for instance) based on a PUDS message request.

Syntax

Pascal OO

```
class function WaitForFunctionalResponses_2013(
  channel: cantp_handle;
  msg_request: Puds_msg;
  timeout: UInt32;
  timeout_enhanced: UInt32;
  wait_until_timeout: Boolean;
  max_msg_count: UInt32;
  out_msg_responses: Puds_msg;
  var out_msg_count: UInt32
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_WaitForFunctionalResponses_2013")]
public static extern uds_status WaitForFunctionalResponses_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    [In] ref uds_msg msg_request,
    UInt32 timeout,
    UInt32 timeout_enhanced,
    [MarshalAs(UnmanagedType.U1)]
    bool wait_until_timeout,
    UInt32 max_msg_count,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex=5)]
    [Out] uds_msg[] out_msg_responses,
    out UInt32 out_msg_count
);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_WaitForFunctionalResponses_2013")]
static uds_status WaitForFunctionalResponses_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    [In] uds_msg %msg_request,
    UInt32 timeout,
    UInt32 timeout_enhanced,
    [MarshalAs(UnmanagedType::U1)]
    bool wait_until_timeout,
    UInt32 max_msg_count,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 5)]
    [Out] array<uds_msg> ^out_msg_responses,
    UInt32 %out_msg_count);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_WaitForFunctionalResponses_2013")>
Public Shared Function WaitForFunctionalResponses_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByRef msg_request As uds_msg,
    ByVal timeout As UInt32,
    ByVal timeout_enhanced As UInt32,
```

```

<MarshalAs(UnmanagedType.U1)>
ByVal wait_until_timeout As Boolean,
ByVal max_msg_count As UInt32,
<MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=5)>
<Out> ByVal out_msg_responses As uds_msg(),
ByRef out_msg_count As UInt32) As uds_status
End Function

```

Parameters

| Parameter | Description |
|--------------------|---|
| channel | The handle of a PUDS channel (see <code>can_tp_handle</code> on page 105). |
| msg_request | A <code>uds_msg</code> containing the PUDS message request that was previously sent (see <code>uds_msg</code> on page 21). |
| timeout | Maximum time to wait (in milliseconds) for a message indication corresponding to the message request. Note: A zero value means unlimited time. |
| timeout_enhanced | Maximum time to wait (in milliseconds) for a message to be complete if server/ECU ask more time. |
| wait_until_timeout | If false the method is interrupted if <code>out_msg_count</code> reaches <code>max_msg_count</code> . |
| max_msg_count | Size of the responses buffer array (maximum messages that can be received). |
| out_msg_responses | A <code>uds_msg</code> buffer array to store the PUDS response message (see <code>uds_msg</code> on page 21). |
| out_msg_count | Output, number of read messages. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|--|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of initialized channels of the calling application. |
| <code>PUDS_STATUS_NO_MESSAGE</code> | Indicates that no matching message was received in the given time. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is invalid. |
| <code>PUDS_STATUS_SERVICE_RX_OVERFLOW</code> | Service received more messages than input buffer expected. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks


The criteria to identify if a response matches the message request is based only on the network addressing information and the UDS service identifier: if the same service is requested multiple times with different parameters (like service `ReadDataByIdentifier` with different data identifiers), the user will have to ensure that the extra content matches the original request.

The timeout parameter is ignored once a message indication matching the request is received (i.e. the first frame of the message), the timeout enhanced is then used.

The parameters of `WaitFor*_2013` methods have a new order. They do not keep the order of the previous version.

Example

The following example shows the use of the method `WaitForFunctionalResponses_2013` on the channel `PCANTP_HANDLE_USBBUS1`. It writes a PUDS functional request on the CAN Bus, waits for the confirmation of the transmission, and then waits to receive responses from ECUs until a timeout occurs. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg confirmation = new uds_msg();
UInt32 response_array_length = 5;
uds_msg[] response_array = new uds_msg[5];
UInt32 count;

// prepare an 11bit CAN ID, functionally addressed UDS message
uds_msgconfig config = new uds_msgconfig();
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_OBD_FUNCTIONAL;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_FUNCTIONAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

result = UDSApi.MsgAlloc_2013(out request, config, 4);
if (!UDSApi.StatusIsOk_2013(result))
{
    MessageBox.Show("Error occurred while allocating request message.", "Error");
}
else
{
    // [...] fill data (functional message is limited to 1 CAN frame)

    // The message is sent using the PCAN-USB
    result = UDSApi.Write_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request);
    if (UDSApi.StatusIsOk_2013(result))
    {
        // Wait for the transmit confirmation
        result = UDSApi.WaitForSingleMessage_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref
            request, true, 10, 100, out confirmation);
        if (UDSApi.StatusIsOk_2013(result) && confirmation.msg.Msgdata_any_Copy.netstatus ==
            cantp_netstatus.PCANTP_NETSTATUS_OK)
        {
            MessageBox.Show("Message was transmitted.", "Success");
            // wait for the responses
            result = UDSApi.WaitForFunctionalResponses_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
                ref request, 10, 100, true, response_array_length, response_array, out count);
            if (count > 0)
            {
                MessageBox.Show("Responses were received", "Success");
                for (int i = 0; i < count; i++)
                {
                    UDSApi.MsgFree_2013(ref response_array[i]);
                }
            }
            else
            {
                MessageBox.Show("No response was received", "Error");
            }
            UDSApi.MsgFree_2013(ref confirmation);
        }
        else
        {
            // An error occurred
            MessageBox.Show("Error occurred while waiting for transmit confirmation.",
                "Error");
        }
    }
}

```

```

        UDSApi.MsgFree_2013(ref request);
    }
    else
    {
        // An error occurred
        MessageBox.Show("An error occurred", "Error");
    }
}

```

C++ / CLR

```

uds_status result;
uds_msg request = {};
uds_msg confirmation = {};
UInt32 response_array_length = 5;
array<uds_msg>^ response_array = gcnew array<uds_msg>(5);
UInt32 count;

// prepare an 11bit CAN ID, functionally addressed UDS message
uds_msgconfig config = {};
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_OBD_FUNCTIONAL;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_FUNCTIONAL;
config.type = PUDS_MSGTYPE_USDT;

result = UDSApi::MsgAlloc_2013(request, config, 4);
if (!UDSApi::StatusIsOk_2013(result))
{
    MessageBox::Show("Error occurred while allocating request message.", "Error");
}
else
{
    // [...] fill data (functional message is limited to 1 CAN frame)

    // The message is sent using the PCAN-USB
    result = UDSApi::Write_2013(PCANTP_HANDLE_USBBUS1, request);
    if (UDSApi::StatusIsOk_2013(result))
    {
        // Wait for the transmit confirmation
        result = UDSApi::WaitForSingleMessage_2013(PCANTP_HANDLE_USBBUS1, request, true,
            10, 100, confirmation);
        if (UDSApi::StatusIsOk_2013(result) && confirmation.msg.msgdata.any->netstatus ==
            PCANTP_NETSTATUS_OK)
        {
            MessageBox::Show("Message was transmitted.", "Success");
            // wait for the responses
            result = UDSApi::WaitForFunctionalResponses_2013(PCANTP_HANDLE_USBBUS1,
                request, 10, 100, true, response_array_length, response_array,
                count);
            if (count > 0)
            {
                MessageBox::Show("Responses were received", "Success");
                for (UInt32 i = 0; i < count; i++)
                {
                    UDSApi::MsgFree_2013(response_array[i]);
                }
            }
            else
            {

```

```

        MessageBox::Show("No response was received", "Error");
    }
    UDSApi::MsgFree_2013(confirmation);
}
else
{
    // An error occurred
    MessageBox::Show("Error occurred while waiting for transmit confirmation.",
        "Error");
}
    UDSApi::MsgFree_2013(request);
}
else
{
    // An error occurred
    MessageBox::Show("An error occurred", "Error");
}
}

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim confirmation As uds_msg = New uds_msg()
Dim response_array_length As UInt32 = 5
Dim response_array(5) As uds_msg
Dim count As UInt32

' prepare an 11bit CAN ID, functionally addressed UDS message
Dim config As uds_msgconfig = New uds_msgconfig()
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = 0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_OBD_FUNCTIONAL
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_FUNCTIONAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

result = UDSApi.MsgAlloc_2013(request, config, 4)
If Not UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Error occurred while allocating request message.", "Error")
Else

    ' [...] fill data (functional message is limited to 1 CAN frame)

    ' The message is sent using the PCAN-USB
    result = UDSApi.Write_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request)
    If UDSApi.StatusIsOk_2013(result) Then

        ' Wait for the transmit confirmation
        result = UDSApi.WaitForSingleMessage_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request,
            True, 10, 100, confirmation)
        If UDSApi.StatusIsOk_2013(result) And confirmation.msg.Msgdata_any_Copy.netstatus =
            cantp_netstatus.PCANTP_NETSTATUS_OK Then
            MessageBox.Show("Message was transmitted.", "Success")

            ' wait for the responses
            result = UDSApi.WaitForFunctionalResponses_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
                request, 10, 100, True, response_array_length, response_array, count)
            If count > 0 Then
                MessageBox.Show("Responses were received", "Success")
                For i As UInt32 = 0 To count - 1

```

```

        UDSApi.MsgFree_2013(response_array(i))
    Next
Else
    MessageBox.Show("No response was received", "Error")
End If
UDSApi.MsgFree_2013(confirmation)
Else
    ' An error occurred
    MessageBox.Show("Error occurred while waiting for transmit confirmation.", "Error")
End If
UDSApi.MsgFree_2013(request)
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If
End If
End If

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    confirmation: uds_msg;
    response_array_length: UInt32;
    response_array: array [0 .. 4] of uds_msg;
    count: UInt32;
    config: uds_msgconfig;
    i: UInt32;
begin
    response_array_length := 5;
    FillChar(request, sizeof(request), 0);
    FillChar(confirmation, sizeof(confirmation), 0);
    FillChar(response_array, sizeof(uds_msg) * response_array_length, 0);

    // prepare an 11bit CAN ID, functionally addressed UDS message
    FillChar(config, sizeof(config), 0);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := 0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_OBD_FUNCTIONAL);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_FUNCTIONAL;
    config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

    result := TUDSApi.MsgAlloc_2013(request, config, 4);
    if not TUDSApi.StatusIsOk_2013(result) then
    begin
        MessageBox(0, 'Error occurred while allocating request message.',
            'Error', MB_OK);
    end
    else
    begin

        // [...] fill data (functional message is limited to 1 CAN frame)

        // The message is sent using the PCAN-USB
        result := TUDSApi.Write_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, @request);
        if TUDSApi.StatusIsOk_2013(result) then

```

```

begin

    // Wait for the transmit confirmation
    result := TUDSApi.WaitForSingleMessage_2013
        (cantp_handle.PCANTP_HANDLE_USBBUS1, @request, true, 10, 100,
        confirmation);
    if TUDSApi.StatusIsOk_2013(result) and
        (confirmation.msg.msgdata_any.netstatus = cantp_netstatus.
        PCANTP_NETSTATUS_OK) then
    begin
        MessageBox(0, 'Message was transmitted.', 'Success', MB_OK);
        // wait for the responses
        result := TUDSApi.WaitForFunctionalResponses_2013
            (cantp_handle.PCANTP_HANDLE_USBBUS1, @request, 10, 100, true,
            response_array_length, @response_array, count);
        if count > 0 then
        begin
            MessageBox(0, 'Responses were received', 'Success', MB_OK);
            for i := 0 to count - 1 do
            begin
                TUDSApi.MsgFree_2013(response_array[i]);
            end;
        end
        else
        begin
            MessageBox(0, 'No response was received', 'Error', MB_OK);
        end;
        TUDSApi.MsgFree_2013(confirmation);
    end
    else
    begin
        // An error occurred
        MessageBox(0, 'Error occurred while waiting for transmit confirmation.',
            'Error', MB_OK);
    end;
    TUDSApi.MsgFree_2013(request);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;
end;
end;
end;

```

See also: `uds_msg` on page 21, `WaitForSingleMessage_2013` on page 235, `WaitForService_2013` on page 248, `WaitForServiceFunctional_2013` on page 252.

Plain function version: `UDS_WaitForFunctionalResponses_2013` on page 654.

3.7.42 WaitForService_2013

Handles the communication workflow for a UDS service expecting a single response. The method waits for a transmit confirmation then for a message response. Even if the `PUDS_MSGTYPE_FLAG_NO_POSITIVE_RESPONSE` flag is set, the method will still wait.

Syntax

Pascal OO

```
class function WaitForService_2013(
  channel: cantp_handle;
  msg_request: Puds_msg;
  var out_msg_response: uds_msg;
  out_msg_request_confirmation: Puds_msg
): uds_status; overload;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_WaitForService_2013")]
public static extern uds_status WaitForService_2013(
  [MarshalAs(UnmanagedType.U4)]
  cantp_handle channel,
  [In] ref uds_msg msg_request,
  out uds_msg out_msg_response,
  out uds_msg out_msg_request_confirmation);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_WaitForService_2013")]
static uds_status WaitForService_2013(
  [MarshalAs(UnmanagedType.U4)]
  cantp_handle channel,
  [In] uds_msg %msg_request,
  uds_msg %out_msg_response,
  uds_msg %out_msg_request_confirmation);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_WaitForService_2013")>
Public Shared Function WaitForService_2013(
  <MarshalAs(UnmanagedType.U4)>
  ByVal channel As cantp_handle,
  ByRef msg_request As uds_msg,
  ByRef out_msg_response As uds_msg,
  ByRef out_msg_request_confirmation As uds_msg) As uds_status
End Function
```

Parameters

| Parameter | Description |
|------------------------------|--|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| msg_request | A <code>uds_msg</code> containing the PUDS request message that was previously sent (see <code>uds_msg</code> on page 21). |
| out_msg_response | A <code>uds_msg</code> buffer to store the PUDS response message (see <code>uds_msg</code> on page 21). |
| out_msg_request_confirmation | A <code>uds_msg</code> buffer to store the PUDS request confirmation message also known as loopback message (see <code>uds_msg</code> on page 21). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---|--|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of initialized channels of the calling application. |
| <code>PUDS_STATUS_SERVICE_TIMEOUT_CONFIRMATION</code> | Timeout while waiting for request confirmation (request message loopback). |
| <code>PUDS_STATUS_SERVICE_TIMEOUT_RESPONSE</code> | Timeout while waiting for response message. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is invalid. |
| <code>PUDS_STATUS_NO_MESSAGE</code> | Indicates that no matching message was received in the given time. |
| <code>PUDS_STATUS_NETWORK_ERROR</code> | A network error occurred. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

The `WaitForService_2013` method is a utility method that calls other PCAN-UDS 2.x API methods to simplify UDS communication workflow


1. The method gets the defined timeouts (global API request and response timeouts and timeouts defined by the current session information with the ECU),
2. Waits for the confirmation of request's transmission,
3. On success, waits for the response.
4. If a negative response code is received stating that the ECU requires extended timing (`PUDS_NRC_EXTENDED_TIMING`, 0x78), the method switches to the enhanced timeout (enhanced P2CAN server max timeout, see `uds_sessioninfo` on page 22) and waits for another response.
5. Fill the output message with response data.

Even if the `PUDS_MSGTYPE_FLAG_NO_POSITIVE_RESPONSE` flag is set in the PUDS request, the method will still wait for an eventual Negative Response. If no error message is received the method will return `PUDS_STATUS_NO_MESSAGE`, although in this case it must not be considered as an error. Moreover, if a negative response code `PUDS_NRC_EXTENDED_TIMING` is received the `PUDS_MSGTYPE_FLAG_NO_POSITIVE_RESPONSE` flag is ignored as stated in ISO-14229-1.

The parameters of `WaitFor*_2013` methods have a new order. They do not keep the order of the previous version.

Example

The following example shows the use of the method `WaitForService_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted (service `ECUReset`), and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel and the mapping were already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
```

```

config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ECUReset message
result = UDSApi.SvcECUReset_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out request,
    UDSApi.uds_svc_param_er.PUDS_SVC_PARAM_ER_SR);
if (UDSApi.StatusIsOk_2013(result))
{
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
    if (UDSApi.StatusIsOk_2013(result))
    {
        MessageBox.Show("Response was received", "Success");
        UDSApi.MsgFree_2013(ref response);
        UDSApi.MsgFree_2013(ref request_confirmation);
    }
    else
    {
        MessageBox.Show("An error occurred", "Error");
    }
    UDSApi.MsgFree_2013(ref request);
}
else
{
    MessageBox.Show("An error occurred, while sending the request.", "Error");
}
}

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response {};
uds_msgconfig config = {};

config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ECUReset message
result = UDSApi::SvcECUReset_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_er::PUDS_SVC_PARAM_ER_SR);
if (UDSApi::StatusIsOk_2013(result))
{
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
    if (UDSApi::StatusIsOk_2013(result))
    {
        MessageBox::Show("Response was received", "Success");
        UDSApi::MsgFree_2013(response);
        UDSApi::MsgFree_2013(request_confirmation);
    }
    else
    {
        MessageBox::Show("An error occurred", "Error");
    }
}

```

```

    }
    UDSApi::MsgFree_2013(request);
}
else
{
    MessageBox::Show("An error occurred, while sending the request.", "Error");
}

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical ECUReset message
result = UDSApi.SvcECUReset_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi.uds_svc_param_er.PUDS_SVC_PARAM_ER_SR)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
    If UDSApi.StatusIsOk_2013(result) Then
        MessageBox.Show("Response was received", "Success")
        UDSApi.MsgFree_2013(response)
        UDSApi.MsgFree_2013(request_confirmation)
    Else
        MessageBox.Show("An error occurred", "Error")
    End If
    UDSApi.MsgFree_2013(request)
Else
    MessageBox.Show("An error occurred, while sending the request.", "Error")
End If

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;

```

```

config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ECUReset message
result := TUDSApi.SvcECUReset_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, uds_svc_param_er.PUDS_SVC_PARAM_ER_SR);
if TUDSApi.StatusIsOk_2013(result) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
    if TUDSApi.StatusIsOk_2013(result) then
    begin
        MessageBox(0, 'Response was received', 'Success', MB_OK);
        TUDSApi.MsgFree_2013(response);
        TUDSApi.MsgFree_2013(request_confirmation);
    end
    else
    begin
        MessageBox(0, 'An error occurred', 'Error', MB_OK);
    end;
    TUDSApi.MsgFree_2013(request);
end
else
begin
    MessageBox(0, 'An error occurred, while sending the request.',
        'Error', MB_OK);
end;
end;
end;

```

See also: [WaitForServiceFunctional_2013](#) on page 252.

Plain function version: [UDS_WaitForService_2013](#) on page 656.

3.7.43 waitForServiceFunctional_2013

Handles the communication workflow for a UDS service requested with functional addressing, i.e. multiple responses can be expected. The method waits for a transmit confirmation then for responses.

Syntax

Pascal OO

```

class function WaitForServiceFunctional_2013(
    channel: cantp_handle;
    msg_request: Puds_msg;
    max_msg_count: UInt32;
    wait_until_timeout: Boolean;
    out_msg_responses: Puds_msg;
    var out_msg_count: UInt32;
    out_msg_request_confirmation: Puds_msg
): uds_status; overload;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_WaitForServiceFunctional_2013")]
public static extern uds_status WaitForServiceFunctional_2013(
    [MarshalAs(UnmanagedType.U4)]

```

```

cantp_handle channel,
ref uds_msg msg_request,
UInt32 max_msg_count,
[MarshalAs(UnmanagedType.U1)]
bool wait_until_timeout,
[MarshalAs(UnmanagedType.LPArray, SizeParamIndex=2)]
[Out] uds_msg[] out_msg_responses,
out UInt32 out_msg_count,
out uds_msg out_msg_request_confirmation);

```

C++ / CLR

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_WaitForServiceFunctional_2013")]
static uds_status WaitForServiceFunctional_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msg %msg_request,
    UInt32 max_msg_count,
    [MarshalAs(UnmanagedType.U1)]
    bool wait_until_timeout,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 2)]
    [Out] array<uds_msg> ^out_msg_responses,
    UInt32 %out_msg_count,
    uds_msg %out_msg_request_confirmation);

```

Visual Basic

```

<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_WaitForServiceFunctional_2013")>
Public Shared Function WaitForServiceFunctional_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByRef msg_request As uds_msg,
    ByVal max_msg_count As UInt32,
    <MarshalAs(UnmanagedType.U1)>
    ByVal wait_until_timeout As Boolean,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=2)>
    <Out> ByVal out_msg_responses As uds_msg(),
    ByRef out_msg_count As UInt32) As uds_status,
    ByRef out_msg_request_confirmation As uds_msg
End Function

```

Parameters

| Parameter | Description |
|------------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| msg_request | A uds_msg containing the PUDS request message that was previously sent (see uds_msg on page 21). |
| max_msg_count | Size of the responses buffer array (maximum messages that can be received). |
| wait_until_timeout | If false, the method is interrupted if out_msg_count reaches max_msg_count. |
| out_msg_responses | A uds_msg buffer array to store the PUDS response message (see uds_msg on page 21). |
| out_msg_count | Output, number of read messages. |
| out_msg_request_confirmation | A uds_msg buffer to store the PUDS request confirmation message also known as loopback message (see uds_msg on page 21). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|--|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of initialized channels of the calling application. |
| PUDS_STATUS_OVERFLOW | Output responses buffer is too small. |
| PUDS_STATUS_SERVICE_TX_ERROR | An error occurred while transmitting the request. |
| PUDS_STATUS_SERVICE_TIMEOUT_CONFIRMATION | Timeout while waiting for request confirmation (request message loopback). |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is invalid. |
| PUDS_STATUS_NO_MESSAGE | Indicates that no matching message was received in the given time. |
| PUDS_STATUS_NETWORK_ERROR | A network error occurred. |
| PUDS_STATUS_SERVICE_RX_OVERFLOW | Service received more messages than input buffer expected. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks


The `WaitForServiceFunctional_2013` method is a utility function that calls other PCAN-UDS 2.x API methods to simplify UDS communication workflow when requests involve functional addressing:

1. The method gets the defined timeouts (global API request and response timeouts and timeouts defined by the current session information with the ECU),
2. Waits for the confirmation of request's transmission,
3. On success, it waits for the responses.
4. The method fills the output messages array with received responses.

The parameters of `WaitFor*_2013` methods have a new order. They do not keep the order of the previous version.

Example

The following example shows the use of the method `WaitForServiceFunctional_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS functional service request is transmitted (service `ECUReset`), and the `WaitForServiceFunctional_2013` method is called to get the responses. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
UInt32 response_array_length = 5;
uds_msg[] response_array = new uds_msg[5];
UInt32 count = 0;
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_OBD_FUNCTIONAL;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_FUNCTIONAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ECUReset message

```

```

result = UDSApi.SvcECUReset_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out request,
    UDSApi.uds_svc_param_er.PUDS_SVC_PARAM_ER_SR);
if (UDSApi.StatusIsOk_2013(result))
{
    result = UDSApi.WaitForServiceFunctional_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref
        request, response_array_length, true, response_array, out count,
        out request_confirmation);
    if (UDSApi.StatusIsOk_2013(result))
    {
        if (count > 0)
        {
            MessageBox.Show("Responses were received", "Success");
            for (UInt32 i = 0; i < count; i++)
            {
                UDSApi.MsgFree_2013(ref response_array[i]);
            }
        }
        else
        {
            MessageBox.Show("No response was received", "Error");
        }
        UDSApi.MsgFree_2013(ref request_confirmation);
    }
    else
    {
        MessageBox.Show("An error occurred", "Error");
    }
    UDSApi.MsgFree_2013(ref request);
}
else
{
    MessageBox.Show("An error occurred, while sending the request.", "Error");
}
}

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
UInt32 response_array_length = 5;
array<uds_msg>^ response_array = gcnew array<uds_msg>(5);
UInt32 count = 0;
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_OBD_FUNCTIONAL;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_FUNCTIONAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ECUReset message
result = UDSApi::SvcECUReset_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_er::PUDS_SVC_PARAM_ER_SR);
if (UDSApi::StatusIsOk_2013(result))
{
    result = UDSApi::WaitForServiceFunctional_2013(PCANTP_HANDLE_USBBUS1, request,
        response_array_length, true, response_array, count, request_confirmation);
    if (UDSApi::StatusIsOk_2013(result))
    {

```



```

    {
        if (count > 0)
        {
            MessageBox::Show("Responses were received", "Success");
            for (int i = 0; i < count; i++)
            {
                UDSApi::MsgFree_2013(response_array[i]);
            }
        }
        else
        {
            MessageBox::Show("No response was received", "Error");
        }
        UDSApi::MsgFree_2013(request_confirmation);
    }
    else
    {
        MessageBox::Show("An error occurred", "Error");
    }
    UDSApi::MsgFree_2013(request);
}
else
{
    MessageBox::Show("An error occurred, while sending the request.", "Error");
}
}

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response_array_length As UInt32 = 5
Dim response_array(5) As uds_msg
Dim count As UInt32 = 0
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_OBD_FUNCTIONAL
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_FUNCTIONAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical ECUReset message
result = UDSApi.SvcECUReset_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi.uds_svc_param_er.PUDS_SVC_PARAM_ER_SR)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForServiceFunctional_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request,
        response_array_length, True, response_array, count, request_confirmation)
    If UDSApi.StatusIsOk_2013(result) Then
        If count > 0 Then
            MessageBox.Show("Responses were received", "Success")
            For i As UInt32 = 0 To count - 1
                UDSApi.MsgFree_2013(response_array(i))
            Next
        Else
            MessageBox.Show("No response was received", "Error")
        End If
        UDSApi.MsgFree_2013(request_confirmation)
    End If
End If

```



```

Else
    MessageBox.Show("An error occurred", "Error")
End If
UDSApi.MsgFree_2013(request)
Else
    MessageBox.Show("An error occurred, while sending the request.", "Error")
End If

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response_array_length: UInt32;
    response_array: array [0 .. 4] of uds_msg;
    count: UInt32;
    config: uds_msgconfig;
    i: UInt32;
begin
    response_array_length := 5;
    count := 0;
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response_array, sizeof(uds_msg) * response_array_length, 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_OBD_FUNCTIONAL);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_FUNCTIONAL;
    config.typem := uds_msgtype.PUDS_MSGTYPE_USDT;

    // Sends a physical ECUReset message
    result := TUDSApi.SvcECUReset_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
        request, uds_svc_param_er.PUDS_SVC_PARAM_ER_SR);
    if TUDSApi.StatusIsOk_2013(result) then
        begin
            result := TUDSApi.WaitForServiceFunctional_2013
                (cantp_handle.PCANTP_HANDLE_USBBUS1, @request, response_array_length, true,
                @response_array, count, @request_confirmation);
            if TUDSApi.StatusIsOk_2013(result) then
                begin
                    if count > 0 then
                        begin
                            MessageBox(0, 'Responses were received', 'Success', MB_OK);
                            for i := 0 to count - 1 do
                                begin
                                    TUDSApi.MsgFree_2013(response_array[i]);
                                end;
                            end
                        else
                            begin

```

```

        MessageBox(0, 'No response was received', 'Error', MB_OK);
    end;
    TUDSApi.MsgFree_2013(request_confirmation);
end
else
begin
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;
    TUDSApi.MsgFree_2013(request);
end
else
begin
    MessageBox(0, 'An error occurred, while sending the request.',
        'Error', MB_OK);
end;
end;

```

See also: [WaitForService_2013](#) on page 248.

Plain function Version: [UDS_WaitForServiceFunctional_2013](#) on page 659.

3.7.44 SvcDiagnosticSessionControl_2013

Writes a UDS request according to the DiagnosticSessionControl service's specifications. The DiagnosticSessionControl service is used to enable different diagnostic sessions in the server.

Syntax

Pascal OO

```

class function SvcDiagnosticSessionControl_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    session_type: uds_svc_param_dsc
): uds_status;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcDiagnosticSessionControl_2013")]
public static extern uds_status SvcDiagnosticSessionControl_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    [MarshalAs(UnmanagedType.U1)]
    uds_svc_param_dsc session_type);

```

C++ / CLR

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcDiagnosticSessionControl_2013")]
static uds_status SvcDiagnosticSessionControl_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType.U1)]
    uds_svc_param_dsc session_type);

```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcDiagnosticSessionControl_2013")>
Public Shared Function SvcDiagnosticSessionControl_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.U1)>
    ByVal session_type As uds_svc_param_dsc) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| session_type | Subfunction parameter: type of the session (see uds_svc_param_dsc on page 71). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |


Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

If this service is called with the `PUDS_MSGTYPE_FLAG_NO_POSITIVE_RESPONSE` parameter set to ignore responses, the API will automatically change the current session to the new one. Else the session information will be updated when the response is received (only if `WaitForService_2013` or `WaitForServiceFunctional_2013` is used).

Example

The following example shows the use of the service method `SvcDiagnosticSessionControl_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical DiagnosticSessionControl message
result = UDSApi.SvcDiagnosticSessionControl_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    out request, UDSApi.uds_svc_param_dsc.PUDS_SVC_PARAM_DSC_DS);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical DiagnosticSessionControl message
result = UDSApi::SvcDiagnosticSessionControl_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_dsc::PUDS_SVC_PARAM_DSC_DS);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else

```

```
// An error occurred
MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);
```

Visual Basic

```
Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical DiagnosticSessionControl message
result = UDSApi.SvcDiagnosticSessionControl_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, UDSApi.uds_svc_param_dsc.PUDS_SVC_PARAM_DSC_DS)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)
```

Pascal OO

```
var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
```

```

config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr := $0;
config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical DiagnosticSessionControl message
result := TUDSApi.SvcDiagnosticSessionControl_2013
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    uds_svc_param_dsc.PUDS_SVC_PARAM_DSC_DS);
if TUDSApi.StatusIsOk_2013(result) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: `WaitForService_2013` on page 248.

Plain function version: `UDS_SvcDiagnosticSessionControl_2013` on page 661.

3.7.45 SvcECUReset_2013

Writes a UDS request according to the ECUReset service's specifications. The ECUReset service is used by the client to request a server (ECU) reset.

Syntax

Pascal OO

```

class function SvcECUReset_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    reset_type: uds_svc_param_er
): uds_status;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcECUReset_2013")]
public static extern uds_status SvcECUReset_2013(
    [MarshalAs(UnmanagedType.U4)]

```

```

cantp_handle channel,
uds_msgconfig request_config,
out uds_msg out_msg_request,
[MarshalAs(UnmanagedType.U1)]
uds_svc_param_er reset_type);

```

C++ / CLR

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcECUReset_2013")]
static uds_status SvcECUReset_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType.U1)]
    uds_svc_param_er reset_type);

```

Visual Basic

```

<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcECUReset_2013")>
Public Shared Function SvcECUReset_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.U1)>
    ByVal reset_type As uds_svc_param_er) As uds_status
End Function

```

Parameters

| Parameter | Description |
|-----------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| reset_type | Subfunction parameter: type of reset (see uds_svc_param_er on page 72). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with Reset_2013 (see Reset_2013 on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see MsgFree_2013 on page 214). |


Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

After receiving a positive response to UDS request `ECUReset` with the help of function `UDS_WaitForService_2013`, the API will revert the session information of that ECU to the default diagnostic session.

Example

The following example shows the use of the service method `SvcECUReset_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ECUReset message
result = UDSApi.SvcECUReset_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out request,
    UDSApi.uds_svc_param_er.PUDS_SVC_PARAM_ER_SR);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
```



```

config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ECUReset message
result = UDSApi::SvcECUReset_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_er::PUDS_SVC_PARAM_ER_SR);
if (UDSApI::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApI::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApI::MsgFree_2013(request);
UDSApI::MsgFree_2013(response);
UDSApI::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical ECUReset message
result = UDSApi.SvcECUReset_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi.uds_svc_param_er.PUDS_SVC_PARAM_ER_SR)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApI.MsgFree_2013(request)
UDSApI.MsgFree_2013(response)
UDSApI.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;

```

```

response: uds_msg;
config: uds_msgconfig;
begin
  FillChar(request, sizeof(request), 0);
  FillChar(request_confirmation, sizeof(request_confirmation), 0);
  FillChar(response, sizeof(response), 0);

  // Set request message configuration
  FillChar(config, sizeof(config), 0);
  config.can_id :=
    UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
  config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
  config.nai.extension_addr := $0;
  config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
  config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
  config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
  config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
  config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

  // Sends a physical ECUReset message
  result := TUDSApi.SvcECUReset_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, uds_svc_param_er.PUDS_SVC_PARAM_ER_SR);
  if TUDSApi.StatusIsOk_2013(result) then
  begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
      @request, response, @request_confirmation);
  end;
  if TUDSApi.StatusIsOk_2013(result) then
  begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
  end
  else
  begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
  end;

  // Free structures
  TUDSApi.MsgFree_2013(request);
  TUDSApi.MsgFree_2013(response);
  TUDSApi.MsgFree_2013(request_confirmation);
end;

```



See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcECUReset_2013](#) on page 663.

3.7.46 SvcSecurityAccess_2013

Writes a UDS request according to the SecurityAccess service's specifications. SecurityAccess service provides a mean to access data and/or diagnostic services which have restricted access for security, emissions, or safety reasons.

Overloads

| | Method | Description |
|---|--|---|
|  | SvcSecurityAccess_2013(cantp_handle, uds_msgconfig, uds_msg, Byte) | Writes to the transmit queue a request for UDS service SecurityAccess. Without optional data. |
|  | SvcSecurityAccess_2013(cantp_handle, uds_msgconfig, uds_msg, Byte, Byte[], UInt32) | Writes to the transmit queue a request for UDS service SecurityAccess. |

Plain function version: `UDS_SvcSecurityAccess_2013` on page 664.

3.7.47 SvcSecurityAccess_2013(cantp_handle, uds_msgconfig, uds_msg, Byte)

Writes a UDS request according to the SecurityAccess service's specifications, without optional data. SecurityAccess service provides a mean to access data and/or diagnostic services which have restricted access for security, emissions, or safety reasons.

Syntax

Pascal OO

```
class function SvcSecurityAccess_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    security_access_type: Byte
): uds_status; overload;
```

C#

```
public static uds_status SvcSecurityAccess_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    Byte security_access_type);
```

C++ / CLR

```
static uds_status SvcSecurityAccess_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    Byte security_access_type);
```

Visual Basic

```
Public Shared Function SvcSecurityAccess_2013(
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal security_access_type As Byte) As uds_status
End Function
```

Parameters

| Parameter | Description |
|----------------------|--|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |
| security_access_type | Subfunction parameter: type of SecurityAccess (see SecurityAccess Type Definitions on page 765). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcSecurityAccess_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical SecurityAccess message
result = UDSApi.SvcSecurityAccess_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out request,
    UDSApi.PUDS_SVC_PARAM_SA_RSD_3);
```

```

if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical SecurityAccess message
result = UDSApi::SvcSecurityAccess_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::PUDS_SVC_PARAM_SA_RSD_3);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT

```

```

config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical SecurityAccess message
result = UDSApI.SvcSecurityAccess_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    UDSApI.PUDS_SVC_PARAM_SA_RSD_3)
If UDSApI.StatusIsOk_2013(result) Then
    result = UDSApI.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApI.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApI.MsgFree_2013(request)
UDSApI.MsgFree_2013(response)
UDSApI.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.type := uds_msgtype.PUDS_MSGTYPE_USDT;

    // Sends a physical SecurityAccess message
    result := TUDSApI.SvcSecurityAccess_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        config, request, TUDSApI.PUDS_SVC_PARAM_SA_RSD_3);
    if TUDSApI.StatusIsOk_2013(result) then
        begin
            result := TUDSApI.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
                @request, response, @request_confirmation);
        end;
    if TUDSApI.StatusIsOk_2013(result) then
        begin
            MessageBox(0, 'Response was received', 'Success', MB_OK);
        end;
    end;
end;

```

```

end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248, [SecurityAccess Type Definitions](#) on page 765.

Plain function version: [UDS_SvcSecurityAccess_2013](#) on page 664.

3.7.48 `svcSecurityAccess_2013(cantp_handle, uds_msgconfig, uds_msg, Byte, Byte[], UInt32)`

Writes a UDS request according to the SecurityAccess service's specifications. SecurityAccess service provides a mean to access data and/or diagnostic services which have restricted access for security, emissions, or safety reasons.

Syntax

Pascal OO

```

class function SvcSecurityAccess_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    security_access_type: Byte;
    security_access_data: PByte;
    security_access_data_size: UInt32
): uds_status; overload;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcSecurityAccess_2013")]
public static extern uds_status SvcSecurityAccess_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    Byte security_access_type,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 5)]
    Byte[] security_access_data,
    UInt32 security_access_data_size);

```

C++ / CLR

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcSecurityAccess_2013")]
static uds_status SvcSecurityAccess_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    Byte security_access_type,

```

```
[MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 5)]
array<Byte> ^ security_access_data,
UInt32 security_access_data_size);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcSecurityAccess_2013")>
Public Shared Function SvcSecurityAccess_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal security_access_type As Byte,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=5)>
    ByVal security_access_data As Byte(),
    ByVal security_access_data_size As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|---------------------------|---|
| Channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| security_access_type | Subfunction parameter: type of SecurityAccess (see SecurityAccess Type Definitions on page 765) |
| security_access_data | If Requesting Seed, buffer is the optional data to transmit to a server/ECU (like identification). If Sending Key, data holds the value generated by the security algorithm corresponding to a specific "seed" value. |
| security_access_data_size | Size in bytes of the buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is not valid. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | The mapping given in the message configuration is unknown. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with Reset_2013 (see Reset_2013 on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_OVERFLOW | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data length. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see MsgFree_2013 on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcSecurityAccess_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical SecurityAccess message
Byte[] data = { 0x42 };
result = UDSApi.SvcSecurityAccess_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out request,
    UDSApi.PUDS_SVC_PARAM_SA_RSD_3, data, 1);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;
```

```
// Sends a physical SecurityAccess message
array<Byte>^ data = { 0x42 };
result = UDSApi::SvcSecurityAccess_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::PUDS_SVC_PARAM_SA_RSD_3, data, 1);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);
```

Visual Basic

```
Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical SecurityAccess message
Dim data As Byte() = {&H42}
result = UDSApi.SvcSecurityAccess_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi.PUDS_SVC_PARAM_SA_RSD_3, data, 1)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)
```

Pascal OO

```
var
    result: uds_status;
    data: array [0 .. 0] of Byte;
    request: uds_msg;
```

```

request_confirmation: uds_msg;
response: uds_msg;
config: uds_msgconfig;
begin
  FillChar(request, sizeof(request), 0);
  FillChar(request_confirmation, sizeof(request_confirmation), 0);
  FillChar(response, sizeof(response), 0);

  // Set request message configuration
  FillChar(config, sizeof(config), 0);
  config.can_id :=
    UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
  config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
  config.nai.extension_addr := $0;
  config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
  config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
  config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
  config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
  config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

  // Sends a physical SecurityAccess message
  data[0] := $42;
  result := TUDSApi.SvcSecurityAccess_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, request, TUDSApi.PUDS_SVC_PARAM_SA_RSD_3, @data, 1);
  if TUDSApi.StatusIsOk_2013(result) then
  begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
      @request, response, @request_confirmation);
  end;
  if TUDSApi.StatusIsOk_2013(result) then
  begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
  end
  else
  begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
  end;

  // Free structures
  TUDSApi.MsgFree_2013(request);
  TUDSApi.MsgFree_2013(response);
  TUDSApi.MsgFree_2013(request_confirmation);
end;
```



See also: [WaitForService_2013](#) on page 248, [SecurityAccess Type Definitions](#) on page 765.

Plain function version: [UDS_SvcSecurityAccess_2013](#) on page 664.

3.7.49 SvcCommunicationControl_2013

Writes a UDS request according to the CommunicationControl service's specifications. CommunicationControl service's purpose is to switch on/off the transmission and/or the reception of certain messages of (a) server(s)/ECU(s).

Overloads

| | Method | Description |
|---|--|--|
|  | SvcCommunicationControl_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_cc, Byte) | Writes to the transmit queue a request for UDS service CommunicationControl. |
|  | SvcCommunicationControl_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_cc, Byte, UInt16) | Writes to the transmit queue a request for UDS service CommunicationControl with node identification number optional data. |

Plain function version: `UDS_SvcCommunicationControl_2013` on page 666.

3.7.50 SvcCommunicationControl_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_cc, Byte)

Writes a UDS request according to the CommunicationControl service's specifications. CommunicationControl service's purpose is to switch on/off the transmission and/or the reception of certain messages of (a) server(s)/ECU(s).

Syntax

Pascal OO

```
class function SvcCommunicationControl_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    control_type: uds_svc_param_cc;
    communication_type: Byte
): uds_status; overload;
```

C#

```
public static uds_status SvcCommunicationControl_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    uds_svc_param_cc control_type,
    Byte communication_type);
```

C++ / CLR

```
static uds_status SvcCommunicationControl_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    uds_svc_param_cc control_type,
    Byte communication_type);
```

Visual Basic

```
Public Shared Function SvcCommunicationControl_2013(
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
```

```

ByVal control_type As uds_svc_param_cc,
ByVal communication_type As Byte) As uds_status
End Function

```

Parameters

| Parameter | Description |
|--------------------|--|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |
| control_type | Subfunction parameter: type of <code>CommunicationControl</code> (see <code>uds_svc_param_cc</code> on page 74). |
| communication_type | A bit-code value to reference the kind of communication to be controlled (see <code>CommunicationControl Communication Type Definitions</code> on page 766). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcCommunicationControl_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

```

```
// Sends a physical CommunicationControl message
result = UDSApi.SvcCommunicationControl_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, UDSApi.uds_svc_param_cc.PUDS_SVC_PARAM_CC_ERXTX,
    UDSApi.PUDS_SVC_PARAM_CC_FLAG_APPL | UDSApi.PUDS_SVC_PARAM_CC_FLAG_NWM |
    UDSApi.PUDS_SVC_PARAM_CC_FLAG_DENWRIRO);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical CommunicationControl message
result = UDSApi::SvcCommunicationControl_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_cc::PUDS_SVC_PARAM_CC_ERXTX,
    UDSApi::PUDS_SVC_PARAM_CC_FLAG_APPL | UDSApi::PUDS_SVC_PARAM_CC_FLAG_NWM |
    UDSApi::PUDS_SVC_PARAM_CC_FLAG_DENWRIRO);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);
```

Visual Basic

```
Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
```

```

Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical CommunicationControl message
result = UDSApi.SvcCommunicationControl_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, UDSApi.uds_svc_param_cc.PUDS_SVC_PARAM_CC_ERXTX,
    UDSApi.PUDS_SVC_PARAM_CC_FLAG_APPL Or UDSApi.PUDS_SVC_PARAM_CC_FLAG_NWM Or
    UDSApi.PUDS_SVC_PARAM_CC_FLAG_DENWRIRO)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.type := uds_msgtype.PUDS_MSGTYPE_USDT;

```



```
// Sends a physical CommunicationControl message
result := TUDSApi.SvcCommunicationControl_2013
  (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
   uds_svc_param_cc.PUDS_SVC_PARAM_CC_ERCTX,
   TUDSApi.PUDS_SVC_PARAM_CC_FLAG_APPL Or TUDSApi.PUDS_SVC_PARAM_CC_FLAG_NWM Or
   TUDSApi.PUDS_SVC_PARAM_CC_FLAG_DENWRIRO);
if TUDSApi.StatusIsOk_2013(result) then
begin
  result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
  MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
  // An error occurred
  MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;
```

See also: [WaitForService_2013](#) on page 248, [CommunicationControl](#) Communication Type Definitions on page 766.

Plain function version: [UDS_SvcCommunicationControl_2013](#) on page 666.

3.7.51 [SvcCommunicationControl_2013](#)(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_cc, Byte, UInt16)

Writes a UDS request according to the CommunicationControl service's specifications. CommunicationControl service's purpose is to switch on/off the transmission and/or the reception of certain messages of (a) server(s)/ECU(s). This overloaded function only works with [PUDS_SVC_PARAM_CC_ERXTXWEAI](#) or [PUDS_SVC_PARAM_CC_ERXTXWEAI](#) control type (see [uds_svc_param_cc](#) on page 74).

Syntax

Pascal OO

```
class function SvcCommunicationControl_2013(
  channel: cantp_handle;
  request_config: uds_msgconfig;
  var out_msg_request: uds_msg;
  control_type: uds_svc_param_cc;
  communication_type: Byte;
  node_identification_number: UInt16
): uds_status; overload;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcCommunicationControl_2013")]
public static extern uds_status SvcCommunicationControl_2013(
  [MarshalAs(UnmanagedType.U4)]
  cantp_handle channel,
```



```
uds_msgconfig request_config,
out uds_msg out_msg_request,
[MarshalAs(UnmanagedType.U1)]
uds_svc_param_cc control_type,
Byte communication_type,
UInt16 node_identification_number);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcCommunicationControl_2013")]
static uds_status SvcCommunicationControl_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType.U1)]
    uds_svc_param_cc control_type,
    Byte communication_type,
    UInt16 node_identification_number);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcCommunicationControl_2013")>
Public Shared Function SvcCommunicationControl_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.U1)>
    ByVal control_type As uds_svc_param_cc,
    ByVal communication_type As Byte,
    ByVal node_identification_number As UInt16) As uds_status
End Function
```

Parameters

| Parameter | Description |
|----------------------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| control_type | Subfunction parameter: type of CommunicationControl (see uds_svc_param_cc on page 74). |
| communication_type | A bit-code value to reference the kind of communication to be controlled (see CommunicationControl Communication Type Definitions on page 766). |
| node_identification_number | A two bytes value, identify a node on a sub-network, only used with PUDS_SVC_PARAM_CC_ERXDTXWEAI or PUDS_SVC_PARAM_CC_ERXTXWEAI control type). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is not valid. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcCommunicationControl_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical CommunicationControl message
result = UDSApi.SvcCommunicationControl_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, UDSApi.uds_svc_param_cc.PUDS_SVC_PARAM_CC_ERXDTXWEAI, 0x1, 0x000A);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical CommunicationControl message
result = UDSApi::SvcCommunicationControl_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_cc::PUDS_SVC_PARAM_CC_ERXDTXWEAI, 0x1, 0x000A);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical CommunicationControl message
result = UDSApi.SvcCommunicationControl_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, UDSApi.uds_svc_param_cc.PUDS_SVC_PARAM_CC_ERXDTXWEAI, &H1, &HA)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")

```

```

Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

    // Sends a physical CommunicationControl message
    result := TUDSApi.SvcCommunicationControl_2013
        (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
        uds_svc_param_cc.PUDS_SVC_PARAM_CC_ERXDTXWEAI, $1, $000A);
    if TUDSApi.StatusIsOk_2013(result) then
        begin
            result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
            @request, response, @request_confirmation);
        end;
    if TUDSApi.StatusIsOk_2013(result) then
        begin
            MessageBox(0, 'Response was received', 'Success', MB_OK);
        end
    else
        begin
            // An error occurred
            MessageBox(0, 'An error occurred', 'Error', MB_OK);
        end;

    // Free structures
    TUDSApi.MsgFree_2013(request);

```

```
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;
```



See also: [WaitForService_2013](#) on page 248, [CommunicationControl](#) Communication Type Definitions on page 766.

Plain function version: [UDS_SvcCommunicationControl_2013](#) on page 666.

3.7.52 SvcTesterPresent_2013

Writes a UDS request according to the TesterPresent service's specifications. TesterPresent service indicates to the server(s)/ECU(s) that a client is still connected to the vehicle and that certain diagnostic services and/or communications that have been previously activated are to remain active.

Overloads

| | Method | Description |
|---|---|---|
|  | SvcTesterPresent_2013(cantp_handle, uds_msgconfig, uds_msg) | Writes to the transmit queue a request for UDS service TesterPresent (with zero subfunction parameter). |
|  | SvcTesterPresent_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_tp) | Writes to the transmit queue a request for UDS service TesterPresent. |

Plain function version: [UDS_SvcTesterPresent_2013](#) on page 668.

3.7.53 SvcTesterPresent_2013(cantp_handle, uds_msgconfig, uds_msg)

Writes a UDS request according to the TesterPresent service's specifications (with zero subfunction parameter). TesterPresent service indicates to a server(s)/ECU(s) that a client is still connected to the vehicle and that certain diagnostic services and/or communications that have been previously activated are to remain active.

Syntax

Pascal OO

```
class function SvcTesterPresent_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg
): uds_status; overload;
```

C#

```
public static uds_status SvcTesterPresent_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request);
```

C++ / CLR

```
static uds_status SvcTesterPresent_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request);
```

Visual Basic

```
Public Shared Function SvcTesterPresent_2013(
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcTesterPresent_2013` on the channel `PCANTP_HANDLE_USBUSB1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
```

```

config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;

// Sends a physical TesterPresent message with no positive response
config.type = uds_msgtype.PUDS_MSGTYPE_USDT |
    uds_msgtype.PUDS_MSGTYPE_FLAG_NO_POSITIVE_RESPONSE;
result = UDSApi.SvcTesterPresent_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out request);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Error");
else if (UDSApi.StatusIsOk_2013(result, uds_status.PUDS_STATUS_NO_MESSAGE))
    MessageBox.Show("No error response", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;

// Sends a physical TesterPresent message with no positive response
config.type = PUDS_MSGTYPE_USDT | PUDS_MSGTYPE_FLAG_NO_POSITIVE_RESPONSE;
result = UDSApi::SvcTesterPresent_2013(PCANTP_HANDLE_USBBUS1, config, request);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Error");
else if (UDSApi::StatusIsOk_2013(result, PUDS_STATUS_NO_MESSAGE))
    MessageBox::Show("No error response", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()

```



```

Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL

' Sends a physical TesterPresent message with no positive response
config.type = uds_msgtype.PUDS_MSGTYPE_USDT Or
    uds_msgtype.PUDS_MSGTYPE_FLAG_NO_POSITIVE_RESPONSE
result = UDSApi.SvcTesterPresent_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Error")
ElseIf UDSApi.StatusIsOk_2013(result, uds_status.PUDS_STATUS_NO_MESSAGE) Then
    MessageBox.Show("No error response", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;

```



```
// Sends a physical TesterPresent message with no positive response
config.type := uds_msgtype(Byte(uds_msgtype.PUDS_MSGTYPE_USDT) Or
    Byte(uds_msgtype.PUDS_MSGTYPE_FLAG_NO_POSITIVE_RESPONSE));
result := TUDSApi.SvcTesterPresent_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, request);
if TUDSApi.StatusIsOk_2013(result) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Error', MB_OK);
end
else if TUDSApi.StatusIsOk_2013(result, uds_status.PUDS_STATUS_NO_MESSAGE) then
begin
    MessageBox(0, 'No error response', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;
```

See also: [WaitForService_2013](#) on page 248, [uds_svc_param_tp](#) on page 75.

Plain function version: [UDS_SvcTesterPresent_2013](#) on page 668.

3.7.54 SvcTesterPresent_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_tp)

Writes a UDS request according to the TesterPresent service's specifications. TesterPresent service indicates to the server(s)/ECU(s) that a client is still connected to the vehicle and that certain diagnostic services and/or communications that have been previously activated are to remain active.

Syntax

Pascal OO

```
class function SvcTesterPresent_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    testerpresent_type: uds_svc_param_tp
): uds_status; overload;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcTesterPresent_2013")]
public static extern uds_status SvcTesterPresent_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    [MarshalAs(UnmanagedType.U1)]
    uds_svc_param_tp testerpresent_type);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcTesterPresent_2013")]
static uds_status SvcTesterPresent_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType::U1)]
    uds_svc_param_tp testerpresent_type);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcTesterPresent_2013")>
Public Shared Function SvcTesterPresent_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.U1)>
    ByVal testerpresent_type As uds_svc_param_tp) As uds_status
End Function
```

Parameters

| Parameter | Description |
|--------------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| testerpresent_type | Tester present type, no Subfunction parameter by default (see uds_svc_param_tp on page 75). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcTesterPresent_2013` on the channel `PCANTP_HANDLE_USBUSB1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical TesterPresent message with no positive response
result = UDSApi.SvcTesterPresent_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out request,
    UDSApi.uds_svc_param_tp.PUDS_SVC_PARAM_TP_ZSUBF);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Error");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical TesterPresent message with no positive response
result = UDSApi::SvcTesterPresent_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_tp::PUDS_SVC_PARAM_TP_ZSUBF);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Error");
else

```

```

    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical TesterPresent message with no positive response
result = UDSApi.SvcTesterPresent_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi.uds_svc_param_tp.PUDS_SVC_PARAM_TP_ZSUBF)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Error")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);

```

```

config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr := $0;
config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical TesterPresent message with no positive response
result := TUDSApi.SvcTesterPresent_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, request, uds_svc_param_tp.PUDS_SVC_PARAM_TP_ZSUBF);
if TUDSApi.StatusIsOk_2013(result) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Error', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248, [uds_svc_param_tp](#) on page 75.

Plain function version: [UDS_SvcTesterPresent_2013](#) on page 668.

3.7.55 SvcSecuredDataTransmission_2013

Writes a UDS request according to the SecuredDataTransmission service's specifications(ISO-14229-1:2013). SecuredDataTransmission service's purpose is to transmit data that are protected against attacks from third parties, which could endanger data security.

Syntax

Pascal OO

```

class function SvcSecuredDataTransmission_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    security_data_request_record: PByte;
    security_data_request_record_size: UInt32
): uds_status;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcSecuredDataTransmission_2013")]
public static extern uds_status SvcSecuredDataTransmission_2013(

```

```
[MarshalAs(UnmanagedType.U4)]
cantp_handle channel,
uds_msgconfig request_config,
out uds_msg out_msg_request,
[MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 4)]
Byte[] security_data_request_record,
UInt32 security_data_request_record_size);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcSecuredDataTransmission_2013")]
static uds_status SvcSecuredDataTransmission_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 4)]
    array<Byte> ^security_data_request_record,
    UInt32 security_data_request_record_size);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcSecuredDataTransmission_2013")>
Public Shared Function SvcSecuredDataTransmission_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=4)>
    ByVal security_data_request_record As Byte(),
    ByVal security_data_request_record_size As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| security_data_request_record | Buffer containing the data as processed by the Security Sub-Layer (See ISO-15764). |
| security_data_request_record_size | Size in bytes of the buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is not valid. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | The mapping given in the message configuration is unknown. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with Reset_2013 (see Reset_2013 on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_OVERFLOW | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data length. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see MsgFree_2013 on page 214). |


Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see [MsgFree_2013](#) on page 214).

When using SecuredDataTransmission service, user may need to construct a [security_data_request_record](#) that is equal to another UDS service request definition. That is why [UDS_Svc*](#) functions can be called with [PUDS_ONLY_PREPARE_REQUEST](#) as channel identifier (instead of using [cantp_handle](#)): it prepares the [uds_msg](#) structure without sending it.

Example

The following example shows the use of the service method [SvcSecuredDataTransmission_2013](#) on the channel [PCANTP_HANDLE_USBBUS1](#). A UDS physical service request is transmitted, and the [WaitForService_2013](#) method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

Byte[] buffer = { 0xF0, 0xA1, 0xB2, 0xC3 };

// Sends a physical SecuredDataTransmission message
result = UDSApi.SvcSecuredDataTransmission_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, buffer, 4);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
```



```

uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

array<Byte>^ buffer = { 0xF0, 0xA1, 0xB2, 0xC3 };

// Sends a physical SecuredDataTransmission message
result = UDSApi::SvcSecuredDataTransmission_2013(PCANTP_HANDLE_USBBUS1, config, request,
    buffer, 4);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

Dim buffer As Byte() = {&HF0, &HA1, &HB2, &HC3}

' Sends a physical SecuredDataTransmission message
result = UDSApi.SvcSecuredDataTransmission_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, buffer, 4)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred

```



```

    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    buffer: array [0 .. 3] of Byte;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

    buffer[0] := $F0;
    buffer[1] := $A1;
    buffer[2] := $B2;
    buffer[3] := $C3;

    // Sends a physical SecuredDataTransmission message
    result := TUDSApi.SvcSecuredDataTransmission_2013
        (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request, @buffer, 4);
    if TUDSApi.StatusIsOk_2013(result) then
    begin
        result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
            @request, response, @request_confirmation);
    end;
    if TUDSApi.StatusIsOk_2013(result) then
    begin
        MessageBox(0, 'Response was received', 'Success', MB_OK);
    end
    else
    begin
        // An error occurred
        MessageBox(0, 'An error occurred', 'Error', MB_OK);
    end;

    // Free structures

```

```
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;
```

See also: `WaitForService_2013` on page 248.

Plain function version: `UDS_SvcSecuredDataTransmission_2013` on page 670.

3.7.56 SvcSecuredDataTransmission_2020

Writes a UDS request according to the SecuredDataTransmission service's specifications (ISO-14229-1:2020). SecuredDataTransmission service's purpose is to transmit data that are protected against attacks from third parties, which could endanger data security.

Syntax

Pascal OO

```
class function SvcSecuredDataTransmission_2020(
  channel: cantp_handle;
  request_config: uds_msgconfig;
  var out_msg_request: uds_msg;
  administrative_parameter: UInt16;
  signature_encryption_calculation: Byte;
  anti_replay_counter: UInt16;
  internal_service_identifier: Byte;
  service_specific_parameters: PByte;
  service_specific_parameters_size: UInt32;
  signature_mac: PByte;
  signature_size: UInt16
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcSecuredDataTransmission_2020")]
public static extern uds_status SvcSecuredDataTransmission_2020(
  cantp_handle channel,
  uds_msgconfig request_config,
  out uds_msg out_msg_request,
  UInt16 administrative_parameter,
  Byte signature_encryption_calculation,
  UInt16 anti_replay_counter,
  Byte internal_service_identifier,
  [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 8)]
  Byte[] service_specific_parameters,
  UInt32 service_specific_parameters_size,
  [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 10)]
  Byte[] signature_mac,
  UInt16 signature_size);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcSecuredDataTransmission_2020")]
static uds_status SvcSecuredDataTransmission_2020(
  [MarshalAs(UnmanagedType::U4)]
  cantp_handle channel,
  uds_msgconfig request_config,
  uds_msg %out_msg_request,
  UInt16 administrative_parameter,
  Byte signature_encryption_calculation,
```

```

UInt16 anti_replay_counter,
Byte internal_service_identifier,
[MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 8)]
array<Byte> ^service_specific_parameters,
UInt32 service_specific_parameters_size,
[MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 10)]
array<Byte> ^signature_mac,
UInt16 signature_size);

```

Visual Basic

```

<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcSecuredDataTransmission_2020")>
Public Shared Function SvcSecuredDataTransmission_2020(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal administrative_parameter As UInt16,
    ByVal signature_encryption_calculation As Byte,
    ByVal anti_replay_counter As UInt16,
    ByVal internal_service_identifier As Byte,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=8)>
    ByVal service_specific_parameters As Byte(),
    ByVal service_specific_parameters_size As UInt32,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=10)>
    ByVal signature_mac As Byte(),
    ByVal signature_size As UInt16) As uds_status
End Function

```

Parameters

| Parameter | Description |
|----------------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |
| administrative_parameter | Security features used in the message (see SecuredDataTransmission Administrative Parameter Flags Definitions on page 767) |
| signature_encryption_calculation | Signature or encryption algorithm identifier. |
| anti_replay_counter | Anti-replay counter value. |
| internal_service_identifier | Internal message service request identifier (see uds_service on page 53). |
| service_specific_parameters | Buffer that contains internal message service request data. |
| service_specific_parameters_size | Internal message service request data size (in bytes). |
| signature_mac | Buffer that contains signature used to verify the message. |
| signature_size | Size in bytes of the signature. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_OVERFLOW</code> | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data size. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |


Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

When using `SecuredDataTransmission` service, user may need to construct a `security_data_request_record` that is equal to another UDS service request definition. That is why `UDS_Svc*` functions can be called with `PUDS_ONLY_PREPARE_REQUEST` as channel identifier (instead of using `cantp_handle`): it prepares the `uds_msg` structure without sending it.

Example

The following example shows the use of the service method `SvcSecuredDataTransmission_2020` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();
UInt16 administrative_parameter;
Byte signature_encryption_calculation;
UInt16 anti_replay_counter;
Byte internal_service_identifier;
Byte[] service_specific_parameters = new Byte[4];
UInt32 service_specific_parameters_size;
Byte[] signature_mac = new Byte[6];
UInt16 signature_size;

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
```

```

config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical SecuredDataTransmission message
administrative_parameter = UDSApi.PUDS_SVC_PARAM_APAR_REQUEST_MSG_FLAG |
    UDSApi.PUDS_SVC_PARAM_APAR_REQUEST_RESPONSE_SIGNATURE_FLAG |
    UDSApi.PUDS_SVC_PARAM_APAR_SIGNED_MSG_FLAG;
signature_encryption_calculation = 0x0;
anti_replay_counter = 0x0124;
internal_service_identifier = 0x2E;
service_specific_parameters[0] = 0xF1;
service_specific_parameters[1] = 0x23;
service_specific_parameters[2] = 0xAA;
service_specific_parameters[3] = 0x55;
service_specific_parameters_size = 4;
signature_mac[0] = 0xDB;
signature_mac[1] = 0xD1;
signature_mac[2] = 0x0E;
signature_mac[3] = 0xDC;
signature_mac[4] = 0x55;
signature_mac[5] = 0xAA;
signature_size = 0x0006;
result = UDSApi.SvcSecuredDataTransmission_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, administrative_parameter, signature_encryption_calculation, anti_replay_counter,
    internal_service_identifier, service_specific_parameters, service_specific_parameters_size,
    signature_mac, signature_size);
if (UDSApI.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApI.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApI.MsgFree_2013(ref request);
UDSApI.MsgFree_2013(ref response);
UDSApI.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};
UInt16 administrative_parameter;
Byte signature_encryption_calculation;
UInt16 anti_replay_counter;
Byte internal_service_identifier;
array<Byte>^ service_specific_parameters = gcnew array<Byte>(4);
UInt32 service_specific_parameters_size;
array<Byte>^ signature_mac = gcnew array<Byte>(6);
UInt16 signature_size;

// Set request message configuration
config.can_id = (UInt16)uds_can_id::PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype::PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol::PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address::PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;

```

```

config.nai.target_addr = (UInt16)uds_address::PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing::PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype::PUDS_MSGTYPE_USDT;

// Sends a physical SecuredDataTransmission message
administrative_parameter = UDSApi::PUDS_SVC_PARAM_APAR_REQUEST_MSG_FLAG |
    UDSApi::PUDS_SVC_PARAM_APAR_REQUEST_RESPONSE_SIGNATURE_FLAG |
    UDSApi::PUDS_SVC_PARAM_APAR_SIGNED_MSG_FLAG;
signature_encryption_calculation = 0x0;
anti_replay_counter = 0x0124;
internal_service_identifier = 0x2E;
service_specific_parameters[0] = 0xF1;
service_specific_parameters[1] = 0x23;
service_specific_parameters[2] = 0xAA;
service_specific_parameters[3] = 0x55;
service_specific_parameters_size = 4;
signature_mac[0] = 0xDB;
signature_mac[1] = 0xD1;
signature_mac[2] = 0x0E;
signature_mac[3] = 0xDC;
signature_mac[4] = 0x55;
signature_mac[5] = 0xAA;
signature_size = 0x0006;
result = UDSApi::SvcSecuredDataTransmission_2020(cantp_handle::PCANTP_HANDLE_USBBUS1, config,
    request, administrative_parameter, signature_encryption_calculation,
    anti_replay_counter, internal_service_identifier, service_specific_parameters,
    service_specific_parameters_size, signature_mac, signature_size);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(cantp_handle::PCANTP_HANDLE_USBBUS1, request,
        response, request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()
Dim administrative_parameter As UInt16
Dim signature_encryption_calculation As Byte
Dim anti_replay_counter As UInt16
Dim internal_service_identifier As Byte
Dim service_specific_parameters() As Byte = New Byte(4) {}
Dim service_specific_parameters_size As UInt32
Dim signature_mac() As Byte = New Byte(6) {}
Dim signature_size As UInt16

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT

```

```

config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical SecuredDataTransmission message
administrative_parameter = UDSApi.PUDS_SVC_PARAM_APAR_REQUEST_MSG_FLAG Or
    UDSApi.PUDS_SVC_PARAM_APAR_REQUEST_RESPONSE_SIGNATURE_FLAG Or
    UDSApi.PUDS_SVC_PARAM_APAR_SIGNED_MSG_FLAG
signature_encryption_calculation = &H0
anti_replay_counter = &H124
internal_service_identifier = &H2E
service_specific_parameters(0) = &HF1
service_specific_parameters(1) = &H23
service_specific_parameters(2) = &HAA
service_specific_parameters(3) = &H55
service_specific_parameters_size = 4
signature_mac(0) = &HDB
signature_mac(1) = &HD1
signature_mac(2) = &HE
signature_mac(3) = &HDC
signature_mac(4) = &H55
signature_mac(5) = &HAA
signature_size = &H6
result = UDSApi.SvcSecuredDataTransmission_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, administrative_parameter, signature_encryption_calculation, anti_replay_counter,
    internal_service_identifier, service_specific_parameters, service_specific_parameters_size,
    signature_mac, signature_size)
If (UDSApI.StatusIsOk_2013(result)) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If (UDSApI.StatusIsOk_2013(result)) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApI.MsgFree_2013(request)
UDSApI.MsgFree_2013(response)
UDSApI.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    administrative_parameter: UInt16;
    signature_encryption_calculation: Byte;
    anti_replay_counter: UInt16;
    internal_service_identifier: Byte;
    service_specific_parameters: array [0 .. 3] of Byte;
    service_specific_parameters_size: UInt32;
    signature_mac: array [0 .. 5] of Byte;
    signature_size: UInt16;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);

```



```

FillChar(response, sizeof(response), 0);

// Set request message configuration
FillChar(config, sizeof(config), 0);
config.can_id :=
    UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr := $0;
config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical SecuredDataTransmission message
administrative_parameter := TUDSApi.PUDS_SVC_PARAM_APAR_REQUEST_MSG_FLAG or
    TUDSApi.PUDS_SVC_PARAM_APAR_REQUEST_RESPONSE_SIGNATURE_FLAG or
    TUDSApi.PUDS_SVC_PARAM_APAR_SIGNED_MSG_FLAG;
signature_encryption_calculation := $0;
anti_replay_counter := $0124;
internal_service_identifier := $2E;
service_specific_parameters[0] := $F1;
service_specific_parameters[1] := $23;
service_specific_parameters[2] := $AA;
service_specific_parameters[3] := $55;
service_specific_parameters_size := 4;
signature_mac[0] := $DB;
signature_mac[1] := $D1;
signature_mac[2] := $0E;
signature_mac[3] := $DC;
signature_mac[4] := $55;
signature_mac[5] := $AA;
signature_size := $0006;
result := TUDSApi.SvcSecuredDataTransmission_2020(PCANTP_HANDLE_USBBUS1,
    config, &request, administrative_parameter,
    signature_encryption_calculation, anti_replay_counter,
    internal_service_identifier, PByte(@service_specific_parameters),
    service_specific_parameters_size, PByte(@signature_mac), signature_size);
if (TUDSApi.StatusIsOk_2013(result)) then
    result := TUDSApi.WaitForService_2013(PCANTP_HANDLE_USBBUS1, @request,
        &response, @request_confirmation);
if (TUDSApi.StatusIsOk_2013(result)) then
    MessageBox(0, 'Response was received', 'Success', MB_OK)
else
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);

// Free structures
TUDSApi.MsgFree_2013(&request);
TUDSApi.MsgFree_2013(&response);
TUDSApi.MsgFree_2013(&request_confirmation);
end;

```



See also: SecuredDataTransmission Administrative Parameter Flags Definitions on page 767,
[WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcSecuredDataTransmission_2020](#) on page 672.

3.7.57 SvcControlDTCSetting_2013

Writes a UDS request according to the ControlDTCSetting service's specifications. ControlDTCSetting service shall be used by a client to stop or resume the setting of diagnostic trouble codes (DTCs) in the server(s)/ECU(s).

Overloads

| | Method | Description |
|---|--|---|
|  | SvcControlDTCSetting_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_cdtcs) | Writes to the transmit queue a request for UDS service ControlDTCSetting, without DTC settings control options. |
|  | SvcControlDTCSetting_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_cdtcs, Byte[], UInt32) | Writes to the transmit queue a request for UDS service ControlDTCSetting. |

Plain function version: [UDS_SvcControlDTCSetting_2013](#) on page 675.

3.7.58 SvcControlDTCSetting_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_cdtcs)

Writes a UDS request according to the ControlDTCSetting service's specifications, without DTC setting control options. ControlDTCSetting service shall be used by a client to stop or resume the setting of diagnostic trouble codes (DTCs) in the server(s).

Syntax

Pascal OO

```
class function SvcControlDTCSetting_2013(
  channel: cantp_handle;
  request_config: uds_msgconfig;
  var out_msg_request: uds_msg;
  dtc_setting_type: uds_svc_param_cdtcs
): uds_status; overload;
```

C#

```
public static uds_status SvcControlDTCSetting_2013(
  cantp_handle channel,
  uds_msgconfig request_config,
  out uds_msg out_msg_request,
  uds_svc_param_cdtcs dtc_setting_type);
```

C++ / CLR

```
static uds_status SvcControlDTCSetting_2013(
  cantp_handle channel,
  uds_msgconfig request_config,
  uds_msg %out_msg_request,
  uds_svc_param_cdtcs dtc_setting_type);
```

Visual Basic

```
Public Shared Function SvcControlDTCSetting_2013(
  ByVal channel As cantp_handle,
  ByVal request_config As uds_msgconfig,
  ByRef out_msg_request As uds_msg,
  ByVal dtc_setting_type As uds_svc_param_cdtcs) As uds_status
End Function
```

Parameters

| Parameter | Description |
|------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |
| dtc_setting_type | Subfunction parameter (see <code>uds_svc_param_cdtcs</code> on page 76). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcControlDTCSetting_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ControlDTCSetting message
result = UDSApi.SvcControlDTCSetting_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, UDSApi.uds_svc_param_cdtcs.PUDS_SVC_PARAM_CDTCS_OFF);
```

```

if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ControlDTCSetting message
result = UDSApi::SvcControlDTCSetting_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_cdtcs::PUDS_SVC_PARAM_CDTCS_OFF);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT

```

```

config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical ControlDTCSetting message
result = UDSApi.SvcControlDTCSetting_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi.uds_svc_param_cdtcs.PUDS_SVC_PARAM_CDTCS_OFF)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.type := uds_msgtype.PUDS_MSGTYPE_USDT;

    // Sends a physical ControlDTCSetting message
    result := TUDSApi.SvcControlDTCSetting_2013
        (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
        uds_svc_param_cdtcs.PUDS_SVC_PARAM_CDTCS_OFF);
    if TUDSApi.StatusIsOk_2013(result) then
    begin
        result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
            @request, response, @request_confirmation);
    end;
    if TUDSApi.StatusIsOk_2013(result) then
    begin

```

```

    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcControlDTCSetting_2013](#) on page 675.

3.7.59 `svcControlDTCSetting_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_cdtcs, Byte[], UInt32)`

Writes UDS request according to the ControlDTCSetting service's specifications. ControlDTCSetting service shall be used by a client to stop or resume the setting of diagnostic trouble codes (DTCs) in the server(s).

Syntax

Pascal OO

```

class function SvcControlDTCSetting_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    dtc_setting_type: uds_svc_param_cdtcs;
    dtc_setting_control_option_record: PByte;
    dtc_setting_control_option_record_size: UInt32
): uds_status; overload;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcControlDTCSetting_2013")]
public static extern uds_status SvcControlDTCSetting_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    [MarshalAs(UnmanagedType.U1)]
    uds_svc_param_cdtcs dtc_setting_type,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 5)]
    Byte[] dtc_setting_control_option_record,
    UInt32 dtc_setting_control_option_record_size);

```

C++ / CLR

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcControlDTCSetting_2013")]
static uds_status SvcControlDTCSetting_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType.U1)]
    uds_svc_param_cdtcs dtc_setting_type,

```

```
[MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 5)]
array<Byte> ^dtc_setting_control_option_record,
UInt32 dtc_setting_control_option_record_size);
```

Visual Basic

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcControlDTCSetting_2013")]
static uds_status SvcControlDTCSetting_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType::U1)]
    uds_svc_param_cdtcs dtc_setting_type,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 5)]
    array<Byte> ^dtc_setting_control_option_record,
    UInt32 dtc_setting_control_option_record_size);
```

Parameters

| Parameter | Description |
|--|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |
| dtc_setting_type | Subfunction parameter (see <code>uds_svc_param_cdtcs</code> on page 76). |
| dtc_setting_control_option_record | This parameter record is user-optional and transmits data to a server (ECU) when controlling the DTC setting. It can contain a list of DTCs to be turned on or off. |
| dtc_setting_control_option_record_size | Size in bytes of the buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_OVERFLOW</code> | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data length. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcControlDTCSetting_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

Byte[] buffer = { 0xF0, 0xA1, 0xB2, 0xC3 };

// Sends a physical ControlDTCSetting message
result = UDSApi.SvcControlDTCSetting_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, UDSApi.uds_svc_param_cdtcs.PUDS_SVC_PARAM_CDTCS_OFF, buffer, 4);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

array<Byte>^ buffer = { 0xF0, 0xA1, 0xB2, 0xC3 };

// Sends a physical ControlDTCSetting message
result = UDSApi::SvcControlDTCSetting_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_cdtcs::PUDS_SVC_PARAM_CDTCS_OFF, buffer, 4);
```



```

if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

Dim buffer As Byte() = {&HF0, &HA1, &HB2, &HC3}

' Sends a physical ControlDTCSetting message
result = UDSApi.SvcControlDTCSetting_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi.uds_svc_param_cdtcs.PUDS_SVC_PARAM_CDTCS_OFF, buffer, 4)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    buffer: array [0 .. 3] of Byte;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;

```



```

begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

    buffer[0] := $F0;
    buffer[1] := $A1;
    buffer[2] := $B2;
    buffer[3] := $C3;

    // Sends a physical ControlDTCSetting message
    result := TUDSApi.SvcControlDTCSetting_2013
        (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
         uds_svc_param_cdtcs.PUDS_SVC_PARAM_CDTCS_OFF, @buffer, 4);
    if TUDSApi.StatusIsOk_2013(result) then
        begin
            result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
            @request, response, @request_confirmation);
        end;
    if TUDSApi.StatusIsOk_2013(result) then
        begin
            MessageBox(0, 'Response was received', 'Success', MB_OK);
        end
    else
        begin
            // An error occurred
            MessageBox(0, 'An error occurred', 'Error', MB_OK);
        end;

    // Free structures
    TUDSApi.MsgFree_2013(request);
    TUDSApi.MsgFree_2013(response);
    TUDSApi.MsgFree_2013(request_confirmation);
end;

```




See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcControlDTCSetting_2013](#) on page 675.

3.7.60 SvcResponseOnEvent_2013

Writes a UDS request according to the ResponseOnEvent service's specifications. The ResponseOnEvent service requests a server (ECU) to start or stop the transmission of responses on a specified event.

Overloads

| | Method | Description |
|---|--|---|
|  | SvcResponseOnEvent_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_roe, bool, byte) | Writes to the transmit queue a request for UDS service ResponseOnEvent, without service to respond to record and event type record. |
|  | SvcResponseOnEvent_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_roe, bool, byte, byte[], UInt32) | Writes to the transmit queue a request for UDS service ResponseOnEvent. |
|  | SvcResponseOnEvent_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_roe, bool, byte, byte[], UInt32, byte[], UInt32) | Writes to the transmit queue a request for UDS service ResponseOnEvent. |

Plain function version: `UDS_SvcResponseOnEvent_2013` on page 677.

3.7.61 SvcResponseOnEvent_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_roe, bool, byte)

Writes a UDS request according to the ResponseOnEvent service's specifications, without service to respond to record and event type record. The ResponseOnEvent service requests a server (ECU) to start or stop the transmission of responses on a specified event.

Syntax

Pascal OO

```
class function SvcResponseOnEvent_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    event_type: uds_svc_param_roe;
    store_event: Boolean;
    event_window_time: Byte
): uds_status; overload;
```

C#

```
public static uds_status SvcResponseOnEvent_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    uds_svc_param_roe event_type,
    bool store_event,
    Byte event_window_time);
```

C++ / CLR

```
static uds_status SvcResponseOnEvent_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    uds_svc_param_roe event_type,
    bool store_event,
    Byte event_window_time);
```

Visual Basic

```
Public Shared Function SvcResponseOnEvent_2013(
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal event_type As uds_svc_param_roe,
    ByVal store_event As Boolean,
    ByVal event_window_time As Byte) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| event_type | Subfunction parameter: event type (see uds_svc_param_roe on page 77). |
| store_event | Storage State (true to store event, false to do not store event). |
| event_window_time | Specify a window for the event logic to be active in the server/ECU (see ResponseOnEvent Service Definitions on page 766). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with Reset_2013 (see Reset_2013 on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see MsgFree_2013 on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcResponseOnEvent_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();
```

```
// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ResponseOnEvent message
result = UDSApi.SvcResponseOnEvent_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, UDSApi.uds_svc_param_roe.PUDS_SVC_PARAM_ROE_ONDTCS, false, 0x08);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ResponseOnEvent message
result = UDSApi::SvcResponseOnEvent_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_roe::PUDS_SVC_PARAM_ROE_ONDTCS, false, 0x08);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);
```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical ResponseOnEvent message
result = UDSApi.SvcResponseOnEvent_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi.uds_svc_param_roe.PUDS_SVC_PARAM_ROE_ONDTCS, False, &H8)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);

```

```

config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ResponseOnEvent message
result := TUDSApi.SvcResponseOnEvent_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, request, uds_svc_param_roe.PUDS_SVC_PARAM_ROE_ONDTCS, false, $08);
if TUDSApi.StatusIsOk_2013(result) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248, [ResponseOnEvent Service Definitions](#) on page 766.

Plain function version: [UDS_SvcResponseOnEvent_2013](#) on page 677.

3.7.62 `SvcResponseOnEvent_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_roe, bool, byte, byte[], UInt32)`

Writes a UDS request according to the ResponseOnEvent service's specifications, without service to respond to record. The ResponseOnEvent service requests a server (ECU) to start or stop the transmission of responses on a specified event.

Syntax

Pascal OO

```

class function SvcResponseOnEvent_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    event_type: uds_svc_param_roe;
    store_event: Boolean;
    event_window_time: Byte;
    event_type_record: PByte;
    event_type_record_size: UInt32
): uds_status; overload;

```

C#

```

public static uds_status SvcResponseOnEvent_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    uds_svc_param_roe event_type,
    bool store_event,
    Byte event_window_time,

```

```
Byte[] event_type_record,
UInt32 event_type_record_size);
```

C++ / CLR

```
static uds_status SvcResponseOnEvent_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    uds_svc_param_roe event_type,
    bool store_event,
    Byte event_window_time,
    array<Byte> ^event_type_record,
    UInt32 event_type_record_size);
```

Visual Basic

```
Public Shared Function SvcResponseOnEvent_2013(
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal event_type As uds_svc_param_roe,
    ByVal store_event As Boolean,
    ByVal event_window_time As Byte,
    ByVal event_type_record As Byte(),
    ByVal event_type_record_size As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| event_type | Subfunction parameter: event type (see uds_svc_param_roe on page 77). |
| store_event | Storage State (true to store event, false to do not store event). |
| event_window_time | Specify a window for the event logic to be active in the server/ECU (see ResponseOnEvent Service Definitions on page 766). |
| event_type_record | Additional parameters for the specified event type. |
| event_type_record_size | Size in bytes of the event type Record (see ResponseOnEvent Service Definitions on page 766). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_OVERFLOW</code> | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data length. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcResponseOnEvent_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ResponseOnEvent message
Byte[] event_type_record = { 0x08, (Byte)uds_service.PUDS_SERVICE_SI_ReadDTCInformation };
result = UDSApi.SvcResponseOnEvent_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, UDSApi.uds_svc_param_roe.PUDS_SVC_PARAM_ROE_ONDTCS, false, 0x08,
    event_type_record, 2);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");

```



```

else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ResponseOnEvent message
array<Byte>^ event_type_record = { 0x08, uds_service::PUDS_SERVICE_SI_ReadDTCInformation };
result = UDSApi::SvcResponseOnEvent_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_roe::PUDS_SVC_PARAM_ROE_ONDTCS, false, 0x08, event_type_record,
    2);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL

```

```

config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical ResponseOnEvent message
Dim event_type_record As Byte() = {&H8, uds_service.PUDS_SERVICE_SI_ReadDTCInformation}
result = UDSApI.SvcResponseOnEvent_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    UDSApI.uds_svc_param_roe.PUDS_SVC_PARAM_ROE_ONDTCS, False, &H8, event_type_record, 2)
If UDSApI.StatusIsOk_2013(result) Then
    result = UDSApI.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApI.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApI.MsgFree_2013(request)
UDSApI.MsgFree_2013(response)
UDSApI.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    event_type_record: array [0 .. 1] of Byte;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.typepem := uds_msgtype.PUDS_MSGTYPE_USDT;

    // Sends a physical ResponseOnEvent message
    event_type_record[0] :=
        Byte(uds_svc_param_rdtci.PUDS_SVC_PARAM_RDTCI_RNODTCBSM);
    event_type_record[1] := $01;
    result := TUDSApI.SvcResponseOnEvent_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        config, request, uds_svc_param_roe.PUDS_SVC_PARAM_ROE_ONDTCS, false, $08,
        @event_type_record, 2);
    if TUDSApI.StatusIsOk_2013(result) then
        begin
            result := TUDSApI.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,

```

```

    @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248, [ResponseOnEvent](#) Service Definitions on page 766.

Plain function version: [UDS_SvcResponseOnEvent_2013](#) on page 677.

3.7.63 SvcResponseOnEvent_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_roe, bool, byte, byte[], UInt32, byte[], UInt32)

Writes a UDS request according to the ResponseOnEvent service's specifications. The ResponseOnEvent service requests a server (ECU) to start or stop the transmission of responses on a specified event.

Syntax

Pascal OO

```

class function SvcResponseOnEvent_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    event_type: uds_svc_param_roe;
    store_event: Boolean;
    event_window_time: Byte;
    event_type_record: PByte;
    event_type_record_size: UInt32;
    service_to_respond_to_record: PByte;
    service_to_respond_to_record_size: UInt32
): uds_status; overload;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcResponseOnEvent_2013")]
public static extern uds_status SvcResponseOnEvent_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    [MarshalAs(UnmanagedType.U1)]
    uds_svc_param_roe event_type,
    [MarshalAs(UnmanagedType.I1)]
    bool store_event,
    Byte event_window_time,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 7)]
    Byte[] event_type_record,
    UInt32 event_type_record_size,

```

```
[MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 9)]
Byte[] service_to_respond_to_record,
UInt32 service_to_respond_to_record_size);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcResponseOnEvent_2013")]
static uds_status SvcResponseOnEvent_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType::U1)]
    uds_svc_param_roe event_type,
    [MarshalAs(UnmanagedType::I1)]
    bool store_event,
    Byte event_window_time,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 7)]
    array<Byte> ^event_type_record,
    UInt32 event_type_record_size,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 9)]
    array<Byte> ^service_to_respond_to_record,
    UInt32 service_to_respond_to_record_size);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcResponseOnEvent_2013")>
Public Shared Function SvcResponseOnEvent_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.U1)>
    ByVal event_type As uds_svc_param_roe,
    <MarshalAs(UnmanagedType.I1)>
    ByVal store_event As Boolean,
    ByVal event_window_time As Byte,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=7)>
    ByVal event_type_record As Byte(),
    ByVal event_type_record_size As UInt32,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=9)>
    ByVal service_to_respond_to_record As Byte(),
    ByVal service_to_respond_to_record_size As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|------------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| event_type | Subfunction parameter: event type (see uds_svc_param_roe on page 77). |
| store_event | Storage State (true to store event, false to do not store event). |
| event_window_time | Specify a window for the event logic to be active in the server/ECU (see ResponseOnEvent Service Definitions on page 766). |
| event_type_record | Additional parameters for the specified event type. |
| event_type_record_size | Size in bytes of the event type Record (see ResponseOnEvent Service Definitions on page 766). |
| service_to_respond_to_record | Service parameters, with first byte as service identifier (see uds_svc_param_roe_recommended_service_id on page 78). |

| Parameter | Description |
|-----------------------------------|--|
| service_to_respond_to_record_size | Size in bytes of the service to respond to record. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_OVERFLOW</code> | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data length. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcResponseOnEvent_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.



Note: It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ResponseOnEvent message
Byte[] event_type_record = { (Byte)UDSApi.uds_svc_param_rdtci.PUDS_SVC_PARAM_RDTCI_RNODTCBSM,
0x01 };
Byte[] service_to_respond_to_record = { 0x08,
(Byte)uds_service.PUDS_SERVICE_SI_ReadDTCInformation };
result = UDSApi.SvcResponseOnEvent_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
request, UDSApi.uds_svc_param_roe.PUDS_SVC_PARAM_ROE_ONDTCS, false, 0x08,
```

```

        event_type_record, 2, service_to_respond_to_record, 2);
if (UDSApi.StatusIsOk_2013(result))
result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
    response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ResponseOnEvent message
array<Byte>^ event_type_record =
    {(Byte)UDSApi::uds_svc_param_rdtci::PUDS_SVC_PARAM_RDTCI_RNODTCBSM, 0x01};
array<Byte>^ service_to_respond_to_record = { 0x08,
uds_service::PUDS_SERVICE_SI_ReadDTCInformation };
result = UDSApi::SvcResponseOnEvent_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_roe::PUDS_SVC_PARAM_ROE_ONDTCS, false, 0x08, event_type_record, 2,
    service_to_respond_to_record, 2);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

```

```

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical ResponseOnEvent message
Dim event_type_record As Byte() = {UDSApi.uds_svc_param_rdtci.PUDS_SVC_PARAM_RDTCI_RNODTCBSM,
    &H1}
Dim service_to_respond_to_record As Byte() = {&H8,
    uds_service.PUDS_SERVICE_SI_ReadDTCInformation}
result = UDSApi.SvcResponseOnEvent_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi.uds_svc_param_roe.PUDS_SVC_PARAM_ROE_ONDTCS, False, &H8, event_type_record, 2,
    service_to_respond_to_record, 2)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    event_type_record: array [0 .. 1] of Byte;
    service_to_respond_to_record: array [0 .. 1] of Byte;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);

```



```

config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ResponseOnEvent message
event_type_record[0] := Byte(uds_svc_param_rdtci.PUDS_SVC_PARAM_RDTCI_RNODTCBSM);
event_type_record[1] := $01;
service_to_respond_to_record[0] := $08;
service_to_respond_to_record[1] := Byte(uds_service.PUDS_SERVICE_SI_ReadDTCInformation);
result := TUDSApi.SvcResponseOnEvent_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, request, uds_svc_param_roe.PUDS_SVC_PARAM_ROE_ONDTCS, false, $08,
    @event_type_record, 2, @service_to_respond_to_record, 2);
if TUDSApi.StatusIsOk_2013(result) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```



See also: [WaitForService_2013](#) on page 248, [ResponseOnEvent Service Definitions](#) on page 766.

Plain function version: [UDS_SvcResponseOnEvent_2013](#) on page 677.

3.7.64 SvcLinkControl_2013

Writes a UDS request according to the LinkControl service's specifications. The LinkControl service is used to control the communication link baud rate between the client and the server(s)/ECU(s) for the exchange of diagnostic data.

Overloads

| | Method | Description |
|---|---|---|
|  | SvcLinkControl_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_lc, uds_svc_param_lc_baudrate_identifier) | Writes to the transmit queue a request for UDS service LinkControl (without PUDS_SVC_PARAM_LC_VBTWSBR parameter). |
|  | SvcLinkControl_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_lc, uds_svc_param_lc_baudrate_identifier, UInt32) | Writes to the transmit queue a request for UDS service LinkControl with PUDS_SVC_PARAM_LC_VBTWSBR parameter and link baud rate parameter. |

Plain function version: [UDS_SvcLinkControl_2013](#) on page 679.

3.7.65 SvcLinkControl_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_lc, uds_svc_param_lc_baudrate_identifier)

Writes a UDS request according to the LinkControl service's specifications (without `PUDS_SVC_PARAM_LC_VBTWSBR` parameter). The LinkControl service is used to control the communication link baud rate between the client and the server(s)/ECU(s) for the exchange of diagnostic data.

Syntax

Pascal OO

```
class function SvcLinkControl_2013(
  channel: cantp_handle;
  request_config: uds_msgconfig;
  var out_msg_request: uds_msg;
  link_control_type: uds_svc_param_lc;
  baudrate_identifier: uds_svc_param_lc_baudrate_identifier
): uds_status; overload;
```

C#

```
public static uds_status SvcLinkControl_2013(
  cantp_handle channel,
  uds_msgconfig request_config,
  out uds_msg out_msg_request,
  uds_svc_param_lc link_control_type,
  uds_svc_param_lc_baudrate_identifier baudrate_identifier);
```

C++ / CLR

```
static uds_status SvcLinkControl_2013(
  cantp_handle channel,
  uds_msgconfig request_config,
  uds_msg %out_msg_request,
  uds_svc_param_lc link_control_type,
  uds_svc_param_lc_baudrate_identifier baudrate_identifier);
```

Visual Basic

```
Public Shared Function SvcLinkControl_2013(
  ByVal channel As cantp_handle,
  ByVal request_config As uds_msgconfig,
  ByRef out_msg_request As uds_msg,
  ByVal link_control_type As uds_svc_param_lc,
  ByVal baudrate_identifier As uds_svc_param_lc_baudrate_identifier) As uds_status
End Function
```

Parameters

| Parameter | Description |
|---------------------|--|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |
| link_control_type | Subfunction parameter: Link Control type (see <code>uds_svc_param_lc</code> on page 79). |
| baudrate_identifier | Defined baud rate identifier (see <code>uds_svc_param_lc_baudrate_identifier</code> on page 80, except <code>PUDS_SVC_PARAM_LC_VBTWSBR</code>). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcLinkControl_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical LinkControl message (Verify Fixed Baudrate)
result = UDSApi.SvcLinkControl_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out request,
    UDSApi.uds_svc_param_lc.PUDS_SVC_PARAM_LC_VBTWSBR,
    UDSApi.uds_svc_param_lc_baudrate_identifier.PUDS_SVC_PARAM_LC_BAUDRATE_CAN_250K);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

```

```
// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical LinkControl message (Verify Fixed Baudrate)
result = UDSApi::SvcLinkControl_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_lc::PUDS_SVC_PARAM_LC_VBTWSBR,
    UDSApi::uds_svc_param_lc_baudrate_identifier::PUDS_SVC_PARAM_LC_BAUDRATE_CAN_250K);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);
```

Visual Basic

```
Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical LinkControl message (Verify Fixed Baudrate)
result = UDSApi.SvcLinkControl_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi.uds_svc_param_lc.PUDS_SVC_PARAM_LC_VBTWSBR,
```

```

UDSApi.uds_svc_param_lc_baudrate_identifier.PUDS_SVC_PARAM_LC_BAUDRATE_CAN_250K)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

    // Sends a physical LinkControl message (Verify Fixed Baudrate)
    result := TUDSApi.SvcLinkControl_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        config, request, uds_svc_param_lc.PUDS_SVC_PARAM_LC_VBTWSBR,
        uds_svc_param_lc_baudrate_identifier.PUDS_SVC_PARAM_LC_BAUDRATE_CAN_250K);
    if TUDSApi.StatusIsOk_2013(result) then
    begin
        result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
            @request, response, @request_confirmation);
    end;
    if TUDSApi.StatusIsOk_2013(result) then
    begin
        MessageBox(0, 'Response was received', 'Success', MB_OK);
    end
    else
    begin
        // An error occurred
    end
end

```

```

    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcLinkControl_2013](#) on page 679.

3.7.66 SvcLinkControl_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_lc, uds_svc_param_lc_baudrate_identifier, UInt32)

Writes a UDS request according to the LinkControl service's specifications (with [PUDS_SVC_PARAM_LC_VBTWSBR](#) parameter and link baud rate parameter). The LinkControl service is used to control the communication link baud rate between the client and the server(s)/ECU(s) for the exchange of diagnostic data.

Syntax

Pascal OO

```

class function SvcLinkControl_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    link_control_type: uds_svc_param_lc;
    baudrate_identifier: uds_svc_param_lc_baudrate_identifier;
    link_baudrate: UInt32
): uds_status; overload;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcLinkControl_2013")]
public static extern uds_status SvcLinkControl_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    [MarshalAs(UnmanagedType.U1)]
    uds_svc_param_lc link_control_type,
    [MarshalAs(UnmanagedType.U1)]
    uds_svc_param_lc_baudrate_identifier baudrate_identifier,
    UInt32 link_baudrate);

```

C++ / CLR

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcLinkControl_2013")]
static uds_status SvcLinkControl_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType.U1)]
    uds_svc_param_lc link_control_type,

```

```
[MarshalAs(UnmanagedType::U1)]
uds_svc_param_lc_baudrate_identifier baudrate_identifier,
UInt32 link_baudrate);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcLinkControl_2013")>
Public Shared Function SvcLinkControl_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.U1)>
    ByVal link_control_type As uds_svc_param_lc,
    <MarshalAs(UnmanagedType.U1)>
    ByVal baudrate_identifier As uds_svc_param_lc_baudrate_identifier,
    ByVal link_baudrate As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|---------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |
| link_control_type | Subfunction parameter: Link Control type (see <code>uds_svc_param_lc</code> on page 79). |
| baudrate_identifier | Defined baud rate identifier (see <code>uds_svc_param_lc_baudrate_identifier</code> on page 80). |
| link_baudrate | Used only with <code>PUDS_SVC_PARAM_LC_VBTWSBR</code> parameter: a three-byte value baud rate (baud rate High, Middle and Low Bytes). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcLinkControl_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical LinkControl message (Verify Fixed Baudrate)
result = UDSApi.SvcLinkControl_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out request,
    UDSApi.uds_svc_param_lc.PUDS_SVC_PARAM_LC_VBTWSBR, 0, 500000);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical LinkControl message (Verify Fixed Baudrate)
result = UDSApi::SvcLinkControl_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_lc::PUDS_SVC_PARAM_LC_VBTWSBR,
    (UDSApi::uds_svc_param_lc_baudrate_identifier) 0, 500000);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");

```



```

else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical LinkControl message (Verify Fixed Baudrate)
result = UDSApi.SvcLinkControl_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi.uds_svc_param_lc.PUDS_SVC_PARAM_LC_VBTWSBR, 0, 500000)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=

```



```

    UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr := $0;
config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical LinkControl message (Verify Fixed Baudrate)
result := TUDSApi.SvcLinkControl_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, request, uds_svc_param_lc.PUDS_SVC_PARAM_LC_VBTWSBR,
    uds_svc_param_lc_baudrate_identifier(0), 500000);
if TUDSApi.StatusIsOk_2013(result) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcLinkControl_2013](#) on page 679.

3.7.67 SvcReadDataByIdentifier_2013

Writes a UDS request according to the ReadDataByIdentifier service's specifications. The ReadDataByIdentifier service allows the client to request data record values from the server (ECU) identified by one or more data identifiers.

Syntax

Pascal OO

```

class function SvcReadDataByIdentifier_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    data_identifier: Puds_svc_param_di;
    data_identifier_length: UInt32
): uds_status;

```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDataByIdentifier_2013")]
public static extern uds_status SvcReadDataByIdentifier_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    [MarshalAs(UnmanagedType.LPArray, ArraySubType = UnmanagedType.U2, SizeParamIndex = 4)]
    uds_svc_param_di[] data_identifier,
    UInt32 data_identifier_length);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDataByIdentifier_2013")]
static uds_status SvcReadDataByIdentifier_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType::LPArray, ArraySubType = UnmanagedType::U2, SizeParamIndex = 4)]
    array<uds_svc_param_di> ^data_identifier,
    UInt32 data_identifier_length);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcReadDataByIdentifier_2013")>
Public Shared Function SvcReadDataByIdentifier_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.LPArray, ArraySubType:=UnmanagedType.U2, SizeParamIndex:=4)>
    ByVal data_identifier As uds_svc_param_di(),
    ByVal data_identifier_length As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| data_identifier | Buffer containing a list of two-byte data identifiers (see uds_svc_param_di on page 82). |
| data_identifier_length | Number of elements in the buffer (size in UInt16 of the buffer). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_OVERFLOW</code> | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data length. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcReadDataByIdentifier_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ReadDataByIdentifier message
UDSApi.uds_svc_param_di[] buffer = { UDSApi.uds_svc_param_di.PUDS_SVC_PARAM_DI_ADSDID,
    UDSApi.uds_svc_param_di.PUDS_SVC_PARAM_DI_ECUMDDID };
result = UDSApi.SvcReadDataByIdentifier_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, buffer, 2);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");

```

```

else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDataByIdentifier message
array<UDSApi::uds_svc_param_di>^ buffer = { UDSApi::uds_svc_param_di::PUDS_SVC_PARAM_DI_ADSDID,
    UDSApi::uds_svc_param_di::PUDS_SVC_PARAM_DI_ECUMDDID };
result = UDSApi::SvcReadDataByIdentifier_2013(PCANTP_HANDLE_USBBUS1, config, request, buffer,
    2);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL

```

```

config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical ReadDataByIdentifier message
Dim buffer As UDSApi.uds_svc_param_di() = {UDSApi.uds_svc_param_di.PUDS_SVC_PARAM_DI_ADSDID,
UDSApi.uds_svc_param_di.PUDS_SVC_PARAM_DI_ECUMDDID}
result = UDSApi.SvcReadDataByIdentifier_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
request, buffer, 2)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
    request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    buffer: array [0 .. 1] of uds_svc_param_di;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.typem := uds_msgtype.PUDS_MSGTYPE_USDT;

    // Sends a physical ReadDataByIdentifier message
    buffer[0] := uds_svc_param_di.PUDS_SVC_PARAM_DI_ADSDID;
    buffer[1] := uds_svc_param_di.PUDS_SVC_PARAM_DI_ECUMDDID;
    result := TUDSApi.SvcReadDataByIdentifier_2013
        (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request, @buffer, 2);
    if TUDSApi.StatusIsOk_2013(result) then
        begin
            result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
                @request, response, @request_confirmation);
        end
    end

```

```

end;
if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248, [uds_svc_param_di](#) on page 82.

Plain function version: [UDS_SvcReadDataByIdentifier_2013](#) on page 681.

3.7.68 SvcReadMemoryByAddress_2013

Writes a UDS request according to the ReadMemoryByAddress service's specifications. The ReadMemoryByAddress service allows the client to request memory data from the server (ECU) via a provided starting address and to specify the size of memory to be read.

Syntax

Pascal OO

```

class function SvcReadMemoryByAddress_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    memory_address_buffer: PByte;
    memory_address_size: Byte;
    memory_size_buffer: PByte;
    memory_size_size: Byte
): uds_status;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadMemoryByAddress_2013")]
public static extern uds_status SvcReadMemoryByAddress_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 4)]
    byte[] memory_address_buffer,
    byte memory_address_size,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 6)]
    byte[] memory_size_buffer,
    byte memory_size_size);

```

C++ / CLR

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadMemoryByAddress_2013")]
static uds_status SvcReadMemoryByAddress_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,

```

```

uds_msgconfig request_config,
uds_msg %out_msg_request,
[MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 4)]
array<Byte> ^memory_address_buffer,
Byte memory_address_size,
[MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 6)]
array<Byte> ^memory_size_buffer,
Byte memory_size_size);

```

Visual Basic

```

<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcReadMemoryByAddress_2013")>
Public Shared Function SvcReadMemoryByAddress_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=4)>
    ByVal memory_address_buffer As Byte(),
    ByVal memory_address_size As Byte,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=6)>
    ByVal memory_size_buffer As Byte(),
    ByVal memory_size_size As Byte) As uds_status
End Function

```

Parameters

| Parameter | Description |
|-----------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| memory_address_buffer | Starting address of server (ECU) memory from which data is to be retrieved. |
| memory_address_size | Size in bytes of the memory address buffer (max.: 0xF). |
| memory_size_buffer | Number of bytes to be read starting at the address specified by memory address. |
| memory_size_size | Size in bytes of the memory size buffer (max.: 0xF). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is not valid. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | The mapping given in the message configuration is unknown. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with Reset_2013 (see Reset_2013 on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see MsgFree_2013 on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcReadMemoryByAddress_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
Byte[] memory_address_buffer = new Byte[10];
Byte[] memory_size_buffer = new Byte[10];
Byte memory_address_size = 10;
Byte memory_size_size = 3;
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Fill data
for (int i = 0; i < memory_address_size; i++)
{
    memory_address_buffer[i] = (Byte)(Convert.ToByte('A') + i);
    memory_size_buffer[i] = (Byte)(Convert.ToByte('1') + i);
}

// Sends a physical ReadMemoryByAddress message
result = UDSApi.SvcReadMemoryByAddress_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, memory_address_buffer, memory_address_size, memory_size_buffer, memory_size_size);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
array<Byte>^ memory_address_buffer = gcnew array<Byte>(10);
array<Byte>^ memory_size_buffer = gcnew array<Byte>(10);
```



```

Byte memory_address_size = 10;
Byte memory_size_size = 3;
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Fill data
for (int i = 0; i < memory_address_size; i++)
{
    memory_address_buffer[i] = 'A' + i;
    memory_size_buffer[i] = '1' + i;
}
// Sends a physical ReadMemoryByAddress message
result = UDSApi::SvcReadMemoryByAddress_2013(PCANTP_HANDLE_USBBUS1, config, request,
    memory_address_buffer, memory_address_size, memory_size_buffer, memory_size_size);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim memory_address_buffer(10) As Byte
Dim memory_size_buffer(10) As Byte
Dim memory_address_size As Byte = 10
Dim memory_size_size As Byte = 3
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Fill data
Dim addr_start As Char = "A"
Dim size_start As Char = "1"

```

```

For i As UInt32 = 0 To memory_address_size - 1
    memory_address_buffer(i) = Convert.ToByte(addr_start) + i
    memory_size_buffer(i) = Convert.ToByte(size_start) + i
Next
' Sends a physical ReadMemoryByAddress message
result = UDSApi.SvcReadMemoryByAddress_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, memory_address_buffer, memory_address_size, memory_size_buffer, memory_size_size)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    memory_address_buffer: array [0 .. 9] of Byte;
    memory_size_buffer: array [0 .. 9] of Byte;
    memory_address_size: Byte;
    memory_size_size: Byte;
    i: UInt32;
begin
    memory_address_size := 10;
    memory_size_size := 3;
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

    // Fill data
    for i := 0 to memory_address_size - 1 do
        begin

```

```

    memory_address_buffer[i] := $41 + i;
    memory_size_buffer[i] := $31 + i;
end;
// Sends a physical ReadMemoryByAddress message
result := TUDSApi.SvcReadMemoryByAddress_2013
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request, @memory_address_buffer,
    memory_address_size, @memory_size_buffer, memory_size_size);
if TUDSApi.StatusIsOk_2013(result) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcReadMemoryByAddress_2013](#) on page 683.

3.7.69 SvcReadScalingDataByIdentifier_2013

Writes a UDS request according to the ReadScalingDataByIdentifier service's specifications. The ReadScalingDataByIdentifier service allows the client to request scaling data record information from the server (ECU) identified by a data identifier.

Syntax

Pascal OO

```

class function SvcReadScalingDataByIdentifier_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    data_identifier: uds_svc_param_di
): uds_status;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadScalingDataByIdentifier_2013")]
public static extern uds_status SvcReadScalingDataByIdentifier_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    [MarshalAs(UnmanagedType.U2)]
    uds_svc_param_di data_identifier);

```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadScalingDataByIdentifier_2013")]
static uds_status SvcReadScalingDataByIdentifier_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType::U2)]
    uds_svc_param_di data_identifier);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcReadScalingDataByIdentifier_2013")>
Public Shared Function SvcReadScalingDataByIdentifier_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.U2)>
    ByVal data_identifier As uds_svc_param_di) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| data_identifier | A two-byte Data Identifier (see uds_svc_param_di on page 82). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcReadScalingDataByIdentifier_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ReadScalingDataByIdentifier message
result = UDSApi.SvcReadScalingDataByIdentifier_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    out request, UDSApi.uds_svc_param_di.PUDS_SVC_PARAM_DI_BSFPDID);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadScalingDataByIdentifier message
result = UDSApi::SvcReadScalingDataByIdentifier_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_di::PUDS_SVC_PARAM_DI_BSFPDID);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
```

```

        MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical ReadScalingDataByIdentifier message
result = UDSApi.SvcReadScalingDataByIdentifier_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, UDSApi.uds_svc_param_di.PUDS_SVC_PARAM_DI_BSFPDID)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);

```

```

config.can_id :=
    UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr := $0;
config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ReadScalingDataByIdentifier message
result := TUDSApi.SvcReadScalingDataByIdentifier_2013
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    uds_svc_param_di.PUDS_SVC_PARAM_DI_BSFPDID);
if TUDSApi.StatusIsOk_2013(result) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: `WaitForService_2013` on page 248, `uds_svc_param_di` on page 82.

Plain function version: `UDS_SvcReadScalingDataByIdentifier_2013` on page 685.

3.7.70 SvcReadDataByPeriodicIdentifier_2013

Writes a UDS request according to the `ReadDataByPeriodicIdentifier` service's specifications. The `ReadDataByPeriodicIdentifier` service allows the client to request the periodic transmission of data record values from the server (ECU) identified by one or more periodic data identifier.

Syntax

Pascal OO

```

class function SvcReadDataByPeriodicIdentifier_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    transmission_mode: uds_svc_param_rdbpi;
    periodic_data_identifier: PByte;
    periodic_data_identifier_size: UInt32
): uds_status;

```


C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDataByPeriodicIdentifier_2013")]
public static extern uds_status SvcReadDataByPeriodicIdentifier_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    [MarshalAs(UnmanagedType.U1)]
    uds_svc_param_rdbpi transmission_mode,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 5)]
    byte[] periodic_data_identifier,
    UInt32 periodic_data_identifier_size);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDataByPeriodicIdentifier_2013")]
static uds_status SvcReadDataByPeriodicIdentifier_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType::U1)]
    uds_svc_param_rdbpi transmission_mode,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 5)]
    array<Byte> ^periodic_data_identifier,
    UInt32 periodic_data_identifier_size);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcReadDataByPeriodicIdentifier_2013")>
Public Shared Function SvcReadDataByPeriodicIdentifier_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.U1)>
    ByVal transmission_mode As uds_svc_param_rdbpi,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=5)>
    ByVal periodic_data_identifier As Byte(),
    ByVal periodic_data_identifier_size As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-------------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 25). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| transmission_mode | Transmission rate mode (see uds_svc_param_rdbpi on page 86). |
| periodic_data_identifier | Buffer containing a list of periodic data identifiers (see uds_svc_param_di on page 82). |
| periodic_data_identifier_size | Number of elements in the buffer (size in bytes of the buffer). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is not valid. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | The mapping given in the message configuration is unknown. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_OVERFLOW | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data length. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcReadDataByPeriodicIdentifier_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
Byte[] periodic_data_identifier = new Byte[10];
UInt16 periodic_data_identifier_size = 10;
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Fill Data
for (int i = 0; i < periodic_data_identifier_size; i++)
{
    periodic_data_identifier[i] = (Byte)(Convert.ToByte('A') + i);
}

// Sends a physical ReadDataByPeriodicIdentifier message
result = UDSApi.SvcReadDataByPeriodicIdentifier_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, out request, UDSApi.uds_svc_param_rdbpi.PUDS_SVC_PARAM_RDBPI_SAMR,
    periodic_data_identifier, periodic_data_identifier_size);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);

```

```

if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
array<Byte>^ periodic_data_identifier = gcnew array<Byte>(10);
UInt16 periodic_data_identifier_size = 10;
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Fill Data
for (int i = 0; i < periodic_data_identifier_size; i++)
{
    periodic_data_identifier[i] = 'A' + i;
}

// Sends a physical ReadDataByPeriodicIdentifier message
result = UDSApi::SvcReadDataByPeriodicIdentifier_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_rdbpi::PUDS_SVC_PARAM_RDBPI_SAMR, periodic_data_identifier,
    periodic_data_identifier_size);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim periodic_data_identifier(10) As Byte

```

```

Dim periodic_data_identifier_size As UInt16 = 10
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Fill Data
Dim start_value As Char = "A"
For i As UInt32 = 0 To periodic_data_identifier_size - 1
    periodic_data_identifier(i) = Convert.ToByte(start_value) + i
Next

' Sends a physical ReadDataByPeriodicIdentifier message
result = UDSApi.SvcReadDataByPeriodicIdentifier_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, request, UDSApi.uds_svc_param_rdbpi.PUDS_SVC_PARAM_RDBPI_SAMR,
    periodic_data_identifier, periodic_data_identifier_size)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    periodic_data_identifier: array [0 .. 9] of Byte;
    periodic_data_identifier_size: UInt16;
    i: UInt32;
begin
    periodic_data_identifier_size := 10;
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;

```

```

config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Fill Data
for i := 0 to periodic_data_identifier_size - 1 do
begin
    periodic_data_identifier[i] := $41 + i;
end;

// Sends a physical ReadDataByPeriodicIdentifier message
result := TUDSApi.SvcReadDataByPeriodicIdentifier_2013
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    uds_svc_param_rdbpi.PUDS_SVC_PARAM_RDBPI_SAMR, @periodic_data_identifier,
    periodic_data_identifier_size);
if TUDSApi.StatusIsOk_2013(result) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248, [uds_svc_param_di](#) on page 82.

Plain function version: [UDS_SvcReadDataByPeriodicIdentifier_2013](#) on page 686.

3.7.71 SvcDynamicallyDefineDataIdentifierDBID_2013

Writes a UDS request according to the DynamicallyDefineDataIdentifier service's specifications. The DynamicallyDefineDataIdentifier service allows the client to dynamically define in a server (ECU) a data identifier that can be read via the ReadDataByIdentifier service later. The define by identifier subfunction specifies that the definition of the dynamic data identifier shall occur via a data identifier reference.

Syntax

Pascal OO

```

class function SvcDynamicallyDefineDataIdentifierDBID_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    dynamically_defined_data_identifier: uds_svc_param_di;
    source_data_identifier: PUInt16;

```

```
memory_size: PByte;
position_in_source_data_record: PByte;
number_of_elements: UInt32
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcDynamicallyDefineDataIdentifierDBID_2013")]
public static extern uds_status SvcDynamicallyDefineDataIdentifierDBID_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    [MarshalAs(UnmanagedType.U2)]
    uds_svc_param_di dynamically_defined_data_identifier,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 7)]
    UInt16[] source_data_identifier,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 7)]
    Byte[] memory_size,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 7)]
    Byte[] position_in_source_data_record,
    UInt32 number_of_elements);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcDynamicallyDefineDataIdentifierDBID_2013")]
static uds_status SvcDynamicallyDefineDataIdentifierDBID_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType::U2)]
    uds_svc_param_di dynamically_defined_data_identifier,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 7)]
    array<UInt16> ^source_data_identifier,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 7)]
    array<Byte> ^memory_size,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 7)]
    array<Byte> ^position_in_source_data_record,
    UInt32 number_of_elements);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcDynamicallyDefineDataIdentifierDBID_2013")>
Public Shared Function SvcDynamicallyDefineDataIdentifierDBID_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.U2)>
    ByVal dynamically_defined_data_identifier As uds_svc_param_di,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=7)>
    ByVal source_data_identifier As UInt16(),
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=7)>
    ByVal memory_size As Byte(),
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=7)>
    ByVal position_in_source_data_record As Byte(),
    ByVal number_of_elements As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-------------------------------------|---|
| channel | The handle of a PUDS channel (see <code>canntp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |
| dynamically_defined_data_identifier | A two-byte data identifier (see <code>uds_svc_param_di</code> on page 82). |
| source_data_identifier | Buffer containing the sources of information to be included into the dynamic data record. |
| memory_size | Buffer containing the total numbers of bytes from the source data record address. |
| position_in_source_data_record | Buffer containing the starting byte positions of the excerpt of the source data record. |
| number_of_elements | Number of <code>source_data_identifier</code> / <code>position_in_source_data_record</code> / <code>memory_size</code> triplet. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_OVERFLOW</code> | The total buffer length is too big. The resulting UDS message data size is bigger than the maximum data length. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcDynamicallyDefineDataIdentifierDBID_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
UInt16[] source_data_identifier = new UInt16[10];
Byte[] memory_size = new Byte[10];
Byte[] position_in_source_data_record = new Byte[10];
UInt16 number_of_elements = 10;
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = canntp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
```

```

config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Fill Data
for (int i = 0; i < number_of_elements; i++)
{
    source_data_identifier[i] = (Byte)((((0xF0 + i) << 8) + ('A' + i)));
    memory_size[i] = (Byte)(i + 1);
    position_in_source_data_record[i] = (Byte)(100 + i);
}

// Sends a physical DynamicallyDefineDataIdentifierDBID message
result = UDSApi.SvcDynamicallyDefineDataIdentifierDBID_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, out request, UDSApi.uds_svc_param_di.PUDS_SVC_PARAM_DI_CDDID,
    source_data_identifier, memory_size, position_in_source_data_record, number_of_elements);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
array<UInt16>^ source_data_identifier = gcnew array<UInt16>(10);
array<Byte>^ memory_size = gcnew array<Byte>(10);
array<Byte>^ position_in_source_data_record = gcnew array<Byte>(10);
UInt16 number_of_elements = 10;
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Fill Data
for (int i = 0; i < number_of_elements; i++)
{
    source_data_identifier[i] = (((0xF0 + i) << 8) + ('A' + i));
    memory_size[i] = (i + 1);
    position_in_source_data_record[i] = (100 + i);
}

```



```
// Sends a physical DynamicallyDefineDataIdentifierDBID message
result = UDSApi::SvcDynamicallyDefineDataIdentifierDBID_2013(PCANTP_HANDLE_USBBUS1, config,
    request, UDSApi::uds_svc_param_di::PUDS_SVC_PARAM_DI_CDDID, source_data_identifier,
    memory_size, position_in_source_data_record, number_of_elements);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);
```

Visual Basic

```
Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim source_data_identifier(10) As UInt16
Dim memory_size(10) As Byte
Dim position_in_source_data_record(10) As Byte
Dim number_of_elements As UInt16 = 10
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Fill Data
Dim start_value As Char = "A"
For i As UInt32 = 0 To number_of_elements - 1
    source_data_identifier(i) = (((&HF0 + i) << 8) + (Convert.ToByte(start_value) + i))
    memory_size(i) = i + 1
    position_in_source_data_record(i) = 100 + i
Next

' Sends a physical DynamicallyDefineDataIdentifierDBID message
result = UDSApi.SvcDynamicallyDefineDataIdentifierDBID_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, request, UDSApi.uds_svc_param_di.PUDS_SVC_PARAM_DI_CDDID, source_data_identifier,
    memory_size, position_in_source_data_record, number_of_elements)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
```


End If

```
' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)
```

Pascal OO

```
var
  result: uds_status;
  request: uds_msg;
  request_confirmation: uds_msg;
  response: uds_msg;
  config: uds_msgconfig;
  memory_size: array [0 .. 9] of Byte;
  position_in_source_data_record: array [0 .. 9] of Byte;
  source_data_identifier: array [0 .. 9] of UInt16;
  number_of_elements: UInt16;
  i: UInt32;
begin
  number_of_elements := 10;
  FillChar(request, sizeof(request), 0);
  FillChar(request_confirmation, sizeof(request_confirmation), 0);
  FillChar(response, sizeof(response), 0);

  // Set request message configuration
  FillChar(config, sizeof(config), 0);
  config.can_id :=
    UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
  config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
  config.nai.extension_addr := $0;
  config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
  config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
  config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
  config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
  config.typem := uds_msgtype.PUDS_MSGTYPE_USDT;

  // Fill Data
  for i := 0 to number_of_elements - 1 do
  begin
    source_data_identifier[i] := ((($F0 + i) Shl 8) + ($41 + i));
    memory_size[i] := (i + 1);
    position_in_source_data_record[i] := (100 + i);
  end;

  // Sends a physical DynamicallyDefineDataIdentifierDBID message
  result := TUDSApi.SvcDynamicallyDefineDataIdentifierDBID_2013
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
     uds_svc_param_di.PUDS_SVC_PARAM_DI_CDDID, @source_data_identifier, @memory_size,
     @position_in_source_data_record, number_of_elements);
  if TUDSApi.StatusIsOk_2013(result) then
  begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
     @request, response, @request_confirmation);
  end;
  if TUDSApi.StatusIsOk_2013(result) then
  begin
```

```

    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: `WaitForService_2013` on page 248.

Plain function version: `UDS_SvcDynamicallyDefineDataIdentifierDBID_2013` on page 688.

3.7.72 `svcDynamicallyDefineDataIdentifierDBMA_2013`

Writes a UDS request according to the `DynamicallyDefineDataIdentifier` service's specifications. The `DynamicallyDefineDataIdentifier` service allows the client to dynamically define in a server (ECU) a data identifier that can be read via the `ReadDataByIdentifier` service later. Defined by the memory address subfunction, this specifies that the definition of the dynamic data identifier shall occur via an address reference.

Syntax

Pascal OO

```

class function SvcDynamicallyDefineDataIdentifierDBMA_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    dynamically_defined_data_identifier: uds_svc_param_di;
    memory_address_size: Byte;
    memory_size_size: Byte;
    memory_address_buffer: PByte;
    memory_size_buffer: PByte;
    number_of_elements: UInt32
): uds_status;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcDynamicallyDefineDataIdentifierDBMA_2013")]
public static extern uds_status SvcDynamicallyDefineDataIdentifierDBMA_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    [MarshalAs(UnmanagedType.U2)]
    uds_svc_param_di dynamically_defined_data_identifier,
    Byte memory_address_size,
    Byte memory_size_size,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 8)]
    Byte[] memory_address_buffer,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 8)]
    Byte[] Memory_size_buffer,
    UInt32 number_of_elements);

```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcDynamicallyDefineDataIdentifierDBMA_2013")]
static uds_status SvcDynamicallyDefineDataIdentifierDBMA_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType::U2)]
    uds_svc_param_di dynamically_defined_data_identifier,
    Byte memory_address_size,
    Byte memory_size_size,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 8)]
    array<Byte> ^memory_address_buffer,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 8)]
    array<Byte> ^Memory_size_buffer,
    UInt32 number_of_elements);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcDynamicallyDefineDataIdentifierDBMA_2013")>
Public Shared Function SvcDynamicallyDefineDataIdentifierDBMA_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.U2)>
    ByVal dynamically_defined_data_identifier As uds_svc_param_di,
    ByVal memory_address_size As Byte,
    ByVal memory_size_size As Byte,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=8)>
    ByVal memory_address_buffer As Byte(),
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=8)>
    ByVal Memory_size_buffer As Byte(),
    ByVal number_of_elements As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-------------------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| dynamically_defined_data_identifier | A two-byte Data Identifier (see uds_svc_param_di on page 82). |
| memory_address_size | Size in bytes of the memory address items in the memory address buffer (max.: 0xF). |
| memory_size_size | Size in bytes of the memory size items in the memory size buffer (max.: 0xF). |
| memory_address_buffer | Buffer containing the memory address buffers must be an array of 'buffers length' entries which contains 'memory address length' bytes (size is 'buffers length * memory address length' bytes). |
| Memory_size_buffer | Buffer containing the memory size buffers must be an array of 'buffers length' entries which contains 'memory size length' bytes (size is 'buffers length * memory size length' bytes). |
| number_of_elements | Number of memory address/memory_size couple. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---------------------------------|--|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is not valid. |


| | |
|---|---|
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | The mapping given in the message configuration is unknown. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with Reset_2013 (see Reset_2013 on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_OVERFLOW | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data length. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see MsgFree_2013 on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see [MsgFree_2013](#) on page 214).

Example

The following example shows the use of the service method [SvcDynamicallyDefineDataIdentifierDBMA_2013](#) on the channel [PCANTP_HANDLE_USBBUS1](#). A UDS physical service request is transmitted, and the [WaitForService_2013](#) method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
UInt16 number_of_elements = 3;
Byte[] memory_address_buffer = new Byte[15];
Byte[] memory_size_buffer = new Byte[9];
Byte memory_address_size = 5;
Byte memory_size_size = 3;
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Fill data
for (int j = 0; j < number_of_elements; j++)
{
    for (int i = 0; i < memory_address_size; i++)
    {
        memory_address_buffer[memory_address_size * j + i] = (Byte)((10 * j) + i + 1);
    }
    for (int i = 0; i < memory_size_size; i++)
    {
        memory_size_buffer[memory_size_size * j + i] = (Byte)(100 + (10 * j) + i + 1);
    }
}

```

```
// Sends a physical DynamicallyDefineDataIdentifierDBMA message
result = UDSApi.SvcDynamicallyDefineDataIdentifierDBMA_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, out request, UDSApi.uds_svc_param_di.PUDS_SVC_PARAM_DI_CESWNDID,
    memory_address_size, memory_size_size, memory_address_buffer, memory_size_buffer,
    number_of_elements);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
UInt16 number_of_elements = 3;
array<Byte>^ memory_address_buffer = gcnew array<Byte>(15);
array<Byte>^ memory_size_buffer = gcnew array<Byte>(9);
Byte memory_address_size = 5;
Byte memory_size_size = 3;
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Fill data
for (int j = 0; j < number_of_elements; j++)
{
    for (int i = 0; i < memory_address_size; i++)
    {
        memory_address_buffer[memory_address_size * j + i] = ((10 * j) + i + 1);
    }
    for (int i = 0; i < memory_size_size; i++)
    {
        memory_size_buffer[memory_size_size * j + i] = (100 + (10 * j) + i + 1);
    }
}

// Sends a physical DynamicallyDefineDataIdentifierDBMA message
result = UDSApi::SvcDynamicallyDefineDataIdentifierDBMA_2013(PCANTP_HANDLE_USBBUS1, config,
    request, UDSApi::uds_svc_param_di::PUDS_SVC_PARAM_DI_CESWNDID, memory_address_size,
    memory_size_size, memory_address_buffer, memory_size_buffer, number_of_elements);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
```

```

if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim number_of_elements As UInt16 = 3
Dim memory_address_buffer(15) As Byte
Dim memory_size_buffer(9) As Byte
Dim memory_address_size As Byte = 5
Dim memory_size_size As Byte = 3
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Fill data
For j As UInt32 = 0 To number_of_elements - 1
    For i As UInt32 = 0 To memory_address_size - 1
        memory_address_buffer(memory_address_size * j + i) = ((10 * j) + i + 1)
    Next
    For i As UInt32 = 0 To memory_size_size - 1
        memory_size_buffer(memory_size_size * j + i) = (100 + (10 * j) + i + 1)
    Next
Next

' Sends a physical DynamicallyDefinedDataIdentifierDBMA message
result = UDSApi.SvcDynamicallyDefinedDataIdentifierDBMA_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, request, UDSApi.uds_svc_param_di.PUDS_SVC_PARAM_DI_CESWNDID, memory_address_size,
    memory_size_size, memory_address_buffer, memory_size_buffer, number_of_elements)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures

```

```
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)
```

Pascal OO

```
var
  result: uds_status;
  request: uds_msg;
  request_confirmation: uds_msg;
  response: uds_msg;
  config: uds_msgconfig;
  memory_address_buffer: array [0 .. 14] of Byte;
  memory_size_buffer: array [0 .. 8] of Byte;
  number_of_elements: UInt16;
  memory_address_size: Byte;
  memory_size_size: Byte;
  i: UInt32;
  j: UInt32;
begin
  number_of_elements := 3;
  memory_address_size := 5;
  memory_size_size := 3;
  FillChar(request, sizeof(request), 0);
  FillChar(request_confirmation, sizeof(request_confirmation), 0);
  FillChar(response, sizeof(response), 0);

  // Set request message configuration
  FillChar(config, sizeof(config), 0);
  config.can_id :=
    UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
  config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
  config.nai.extension_addr := $0;
  config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
  config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
  config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
  config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
  config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

  // Fill data
  for j := 0 to number_of_elements - 1 do
  begin
    for i := 0 to memory_address_size - 1 do
    begin
      memory_address_buffer[memory_address_size * j + i] := ((10 * j) + i + 1);
    end;
    for i := 0 to memory_size_size - 1 do
    begin
      memory_size_buffer[memory_size_size * j + i] := (100 + (10 * j) + i + 1);
    end;
  end;

  // Sends a physical DynamicallyDefineDataIdentifierDBMA message
  result := TUDSApi.SvcDynamicallyDefineDataIdentifierDBMA_2013
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
     uds_svc_param_di.PUDS_SVC_PARAM_DI_CESWNDID, memory_address_size, memory_size_size,
     @memory_address_buffer, @memory_size_buffer, number_of_elements);
  if TUDSApi.StatusIsOk_2013(result) then
```

```

begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248, [uds_svc_param_di](#) on page 82.

Plain function version: [UDS_SvcDynamicallyDefineDataIdentifierDBMA_2013](#) on page 690.

3.7.73 svcDynamicallyDefineDataIdentifierCDDDI_2013

Writes a UDS request according to the Dynamically Defined Data Identifier service's specifications. The Clear Dynamically Defined Data Identifier subfunction shall be used to clear the specified dynamic data identifier.

Syntax

Pascal OO

```

class function SvcDynamicallyDefineDataIdentifierCDDDI_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    dynamically_defined_data_identifier: uds_svc_param_di
): uds_status;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcDynamicallyDefineDataIdentifierCDDDI_2013")]
public static extern uds_status SvcDynamicallyDefineDataIdentifierCDDDI_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    [MarshalAs(UnmanagedType.U2)]
    uds_svc_param_di dynamically_defined_data_identifier);

```

C++ / CLR

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcDynamicallyDefineDataIdentifierCDDDI_2013")]
static uds_status SvcDynamicallyDefineDataIdentifierCDDDI_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,

```



```
uds_msg %out_msg_request,
[MarshalAs(UnmanagedType::U2)]
uds_svc_param_di dynamically_defined_data_identifier);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcDynamicallyDefineDataIdentifierCDDDI_2013")>
Public Shared Function SvcDynamicallyDefineDataIdentifierCDDDI_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.U2)>
    ByVal dynamically_defined_data_identifier As uds_svc_param_di) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-------------------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| dynamically_defined_data_identifier | A two-byte data identifier (see uds_svc_param_di on page 82). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcDynamicallyDefineDataIdentifierCDDDI_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
```

```

uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical DynamicallyDefineDataIdentifierCDDDI message
result =
    UDSApi.SvcDynamicallyDefineDataIdentifierCDDDI_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        config, out request, UDSApi.uds_svc_param_di.PUDS_SVC_PARAM_DI_CESWNDID);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical DynamicallyDefineDataIdentifierCDDDI message
result = UDSApi::SvcDynamicallyDefineDataIdentifierCDDDI_2013(PCANTP_HANDLE_USBBUS1, config,
    request, UDSApi::uds_svc_param_di::PUDS_SVC_PARAM_DI_CESWNDID);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures

```

```
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);
```

Visual Basic

```
Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical DynamicallyDefineDataIdentifierCDDDI message
result =
    UDSApi.SvcDynamicallyDefineDataIdentifierCDDDI_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        config, request, UDSApi.uds_svc_param_di.PUDS_SVC_PARAM_DI_CESWNDID)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)
```

Pascal OO

```
var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
```

```

config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical DynamicallyDefineDataIdentifierCDDDI message
result := TUDSApi.SvcDynamicallyDefineDataIdentifierCDDDI_2013
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    uds_svc_param_di.PUDS_SVC_PARAM_DI_CESWNDID);
if TUDSApi.StatusIsOk_2013(result) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248, [uds_svc_param_di](#) on page 82.

Plain function version: [UDS_SvcDynamicallyDefineDataIdentifierCDDDI_2013](#) on page 693.

3.7.74 SvcDynamicallyDefineDataIdentifierClearAllDDDI_2013

Writes a UDS request according to the Clear all Dynamically Defined Data Identifier service's specifications. The Clear Dynamically Defined Data Identifier subfunction (without data identifier parameter) shall be used to clear all the specified dynamic data identifier in the server.

Syntax

Pascal OO

```

class function SvcDynamicallyDefineDataIdentifierClearAllDDDI_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg
): uds_status;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint =
    "UDS_SvcDynamicallyDefineDataIdentifierClearAllDDDI_2013")]
public static extern uds_status SvcDynamicallyDefineDataIdentifierClearAllDDDI_2013(
    [MarshalAs(UnmanagedType.U4)]

```

```
cantp_handle channel,
uds_msgconfig request_config,
out uds_msg out_msg_request);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint =
"UDS_SvcDynamicallyDefineDataIdentifierClearAllDDDI_2013")]
static uds_status SvcDynamicallyDefineDataIdentifierClearAllDDDI_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll",
EntryPoint:="UDS_SvcDynamicallyDefineDataIdentifierClearAllDDDI_2013")>
Public Shared Function SvcDynamicallyDefineDataIdentifierClearAllDDDI_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcDynamicallyDefineDataIdentifierClearAllDDDI_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the

`WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical DynamicallyDefineDataIdentifierClearAllDDDI message
result =
    UDSApi.SvcDynamicallyDefineDataIdentifierClearAllDDDI_2013(
        cantp_handle.PCANTP_HANDLE_USBBUS1, config, out request);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical DynamicallyDefineDataIdentifierClearAllDDDI message
result = UDSApi::SvcDynamicallyDefineDataIdentifierClearAllDDDI_2013(PCANTP_HANDLE_USBBUS1,
```

```

        config, request);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical DynamicallyDefineDataIdentifierClearAllDDDI message
result =
    UDSApi.SvcDynamicallyDefineDataIdentifierClearAllDDDI_2013(
        cantp_handle.PCANTP_HANDLE_USBBUS1, config, request)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;

```



```

begin
  FillChar(request, sizeof(request), 0);
  FillChar(request_confirmation, sizeof(request_confirmation), 0);
  FillChar(response, sizeof(response), 0);

  // Set request message configuration
  FillChar(config, sizeof(config), 0);
  config.can_id :=
    UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
  config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
  config.nai.extension_addr := $0;
  config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
  config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
  config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
  config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
  config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

  // Sends a physical DynamicallyDefineDataIdentifierClearAllDDDI message
  result := TUDSApi.SvcDynamicallyDefineDataIdentifierClearAllDDDI_2013
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request);
  if TUDSApi.StatusIsOk_2013(result) then
    begin
      result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
    end;
  if TUDSApi.StatusIsOk_2013(result) then
    begin
      MessageBox(0, 'Response was received', 'Success', MB_OK);
    end
  else
    begin
      // An error occurred
      MessageBox(0, 'An error occurred', 'Error', MB_OK);
    end;

  // Free structures
  TUDSApi.MsgFree_2013(request);
  TUDSApi.MsgFree_2013(response);
  TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcDynamicallyDefineDataIdentifierClearAllDDDI_2013](#) on page 694.

3.7.75 SvcWriteDataByIdentifier_2013

Writes a UDS request according to the WriteDataByIdentifier service's specifications. The WriteDataByIdentifier service allows the client to write information into the server (ECU) at an internal location specified by the provided data identifier.

Syntax

Pascal OO

```

class function SvcWriteDataByIdentifier_2013(
  channel: cantp_handle;
  request_config: uds_msgconfig;

```



```
var out_msg_request: uds_msg;
data_identifier: uds_svc_param_di;
data_record: PByte;
data_record_size: UInt32
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcWriteDataByIdentifier_2013")]
public static extern uds_status SvcWriteDataByIdentifier_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    [MarshalAs(UnmanagedType.U2)]
    uds_svc_param_di data_identifier,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 5)]
    Byte[] data_record,
    UInt32 data_record_size);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcWriteDataByIdentifier_2013")]
static uds_status SvcWriteDataByIdentifier_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType::U2)]
    uds_svc_param_di data_identifier,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 5)]
    array<Byte> ^data_record,
    UInt32 data_record_size);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcWriteDataByIdentifier_2013")>
Public Shared Function SvcWriteDataByIdentifier_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.U2)>
    ByVal data_identifier As uds_svc_param_di,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=5)>
    ByVal data_record As Byte(),
    ByVal data_record_size As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| data_identifier | A two-byte data identifier (see uds_svc_param_di on page 82). |
| data_record | Buffer containing the data to write. |
| data_record_size | Size in bytes of the buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_OVERFLOW</code> | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data length. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcWriteDataByIdentifier_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
Byte[] data_record = new Byte[10];
UInt16 data_record_size = 10;
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Fill Data
for (int i = 0; i < data_record_size; i++)
{
    data_record[i] = (Byte)(Convert.ToByte('A') + i);
}

// Sends a physical WriteDataByIdentifier message
result = UDSApi.SvcWriteDataByIdentifier_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out

```

```

    request, UDSApi.uds_svc_param_di.PUDS_SVC_PARAM_DI_ASFPDID, data_record, data_record_size);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
array<Byte>^ data_record = gcnew array<Byte>(10);
UInt16 data_record_size = 10;
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Fill Data
for (int i = 0; i < data_record_size; i++)
{
    data_record[i] = 'A' + i;
}

// Sends a physical WriteDataByIdentifier message
result = UDSApi::SvcWriteDataByIdentifier_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_di::PUDS_SVC_PARAM_DI_ASFPDID, data_record, data_record_size);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()

```

```

Dim response As uds_msg = New uds_msg()
Dim data_record(10) As Byte
Dim data_record_size As UInt16 = 10
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Fill Data
Dim start_value As Char = "A"
For i As UInt32 = 0 To data_record_size - 1
    data_record(i) = Convert.ToByte(start_value) + i
Next

' Sends a physical WriteDataByIdentifier message
result = UDSApi.SvcWriteDataByIdentifier_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, UDSApi.uds_svc_param_di.PUDS_SVC_PARAM_DI_ASFPDID, data_record, data_record_size)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    data_record: array [0 .. 9] of Byte;
    data_record_size: UInt16;
    i: UInt32;
begin
    data_record_size := 10;
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;

```

```

config.nai.extension_addr := $0;
config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Fill Data
for i := 0 to data_record_size - 1 do
begin
    data_record[i] := ($41 + i);
end;

// Sends a physical WriteDataByIdentifier message
result := TUDSApi.SvcWriteDataByIdentifier_2013
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    uds_svc_param_di.PUDS_SVC_PARAM_DI_ASFPDID, @data_record, data_record_size);
if TUDSApi.StatusIsOk_2013(result) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248, [uds_svc_param_di](#) on page 82.

Plain function version: [UDS_SvcWriteDataByIdentifier_2013](#) on page 696.

3.7.76 SvcWriteMemoryByAddress_2013

Writes a UDS request according to the WriteMemoryByAddress service's specifications. The WriteMemoryByAddress service allows the client to write information into the server (ECU) at one or more contiguous memory locations.

Syntax

Pascal OO

```

class function SvcWriteMemoryByAddress_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    memory_address_buffer: PByte;

```

```
memory_address_size: Byte;
memory_size_buffer: PByte;
memory_size_size: Byte;
data_record: PByte;
data_record_size: UInt32
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcWriteMemoryByAddress_2013")]
public static extern uds_status SvcWriteMemoryByAddress_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 4)]
    Byte[] memory_address_buffer,
    Byte memory_address_size,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 6)]
    Byte[] memory_size_buffer,
    Byte memory_size_size,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 8)]
    Byte[] data_record,
    UInt32 data_record_size);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcWriteMemoryByAddress_2013")]
static uds_status SvcWriteMemoryByAddress_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 4)]
    array<Byte> ^memory_address_buffer,
    Byte memory_address_size,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 6)]
    array<Byte> ^memory_size_buffer,
    Byte memory_size_size,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 8)]
    array<Byte> ^data_record,
    UInt32 data_record_size);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcWriteMemoryByAddress_2013")>
Public Shared Function SvcWriteMemoryByAddress_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=4)>
    ByVal memory_address_buffer As Byte(),
    ByVal memory_address_size As Byte,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=6)>
    ByVal memory_size_buffer As Byte(),
    ByVal memory_size_size As Byte,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=8)>
    ByVal data_record As Byte(),
    ByVal data_record_size As UInt32) As uds_status
```

End Function

Parameters

| Parameter | Description |
|-----------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |
| memory_address_buffer | Starting address buffer of server (ECU) memory to which data is to be written. |
| memory_address_size | Size in bytes of the memory address buffer (max.: 0xF). |
| memory_size_buffer | Number of bytes to be written starting at the address specified by memory address. |
| memory_size_size | Size in bytes of the memory size buffer (max.: 0xF). |
| data_record | Buffer containing the data to write. |
| data_record_size | Size in bytes of the buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_OVERFLOW</code> | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data length. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcWriteMemoryByAddress_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
Byte[] data_record = new Byte[50];
Byte[] memory_address_buffer = new Byte[50];
Byte[] memory_size_buffer = new Byte[50];
UInt16 data_record_size = 50;
Byte memory_address_size = 5;
```



```

Byte memory_size_size = 3;
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Fill Data
for (int i = 0; i < data_record_size; i++)
{
    data_record[i] = (Byte)(i + 1);
    memory_address_buffer[i] = (Byte)(Convert.ToByte('A') + i);
    memory_size_buffer[i] = (Byte)(10 + i);
}

// Sends a physical WriteMemoryByAddress message
result = UDSApi.SvcWriteMemoryByAddress_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, memory_address_buffer, memory_address_size, memory_size_buffer, memory_size_size,
    data_record, data_record_size);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
array<Byte>^ data_record = gcnew array<Byte>(50);
array<Byte>^ memory_address_buffer = gcnew array<Byte>(50);
array<Byte>^ memory_size_buffer = gcnew array<Byte>(50);
UInt16 data_record_size = 50;
Byte memory_address_size = 5;
Byte memory_size_size = 3;
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

```



```
// Fill Data
for (int i = 0; i < data_record_size; i++)
{
    data_record[i] = (i + 1);
    memory_address_buffer[i] = 'A' + i;
    memory_size_buffer[i] = (10 + i);
}

// Sends a physical WriteMemoryByAddress message
result = UDSApi::SvcWriteMemoryByAddress_2013(PCANTP_HANDLE_USBBUS1, config, request,
    memory_address_buffer, memory_address_size, memory_size_buffer, memory_size_size,
    data_record, data_record_size);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
    request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);
```

Visual Basic

```
Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim data_record(50) As Byte
Dim memory_address_buffer(50) As Byte
Dim memory_size_buffer(50) As Byte
Dim data_record_size As UInt16 = 50
Dim memory_address_size As Byte = 5
Dim memory_size_size As Byte = 3
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Fill Data
Dim start_value As Char = "A"
For i As UInt32 = 0 To data_record_size - 1
    data_record(i) = i + 1
    memory_address_buffer(i) = Convert.ToByte(start_value) + i
    memory_size_buffer(i) = 10 + i
Next

' Sends a physical WriteMemoryByAddress message
result = UDSApi.SvcWriteMemoryByAddress_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, memory_address_buffer, memory_address_size, memory_size_buffer, memory_size_size,
```

```

    data_record, data_record_size)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApI.MsgFree_2013(request)
UDSApI.MsgFree_2013(response)
UDSApI.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    data_record: array [0 .. 49] of Byte;
    memory_address_buffer: array [0 .. 49] of Byte;
    memory_size_buffer: array [0 .. 49] of Byte;
    data_record_size: UInt16;
    memory_address_size: Byte;
    memory_size_size: Byte;
    i: UInt32;
begin
    data_record_size := 50;
    memory_address_size := 5;
    memory_size_size := 3;
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

    // Fill Data
    for i := 0 to data_record_size - 1 do
    begin
        data_record[i] := (i + 1);
        memory_address_buffer[i] := ($41 + i);
        memory_size_buffer[i] := (10 + i);
    end
end

```

```

end;

// Sends a physical WriteMemoryByAddress message
result := TUDSApi.SvcWriteMemoryByAddress_2013
  (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request, @memory_address_buffer,
   memory_address_size, @memory_size_buffer, memory_size_size, @data_record,
   data_record_size);
if TUDSApi.StatusIsOk_2013(result) then
begin
  result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
  MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
  // An error occurred
  MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcWriteMemoryByAddress_2013](#) on page 698.

3.7.77 svcClearDiagnosticInformation_2013

Writes a UDS request according to the ClearDiagnosticInformation service's specifications. The ClearDiagnosticInformation service is used by the client to clear diagnostic information in one server's (ECU) or multiple servers' (ECUs) memory.

Syntax

Pascal OO

```

class function SvcClearDiagnosticInformation_2013(
  channel: cantp_handle;
  request_config: uds_msgconfig;
  var out_msg_request: uds_msg;
  group_of_dtc: UInt32
): uds_status;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcClearDiagnosticInformation_2013")]
public static extern uds_status SvcClearDiagnosticInformation_2013(
  [MarshalAs(UnmanagedType.U4)]
  cantp_handle channel,
  uds_msgconfig request_config,
  out uds_msg out_msg_request,
  UInt32 group_of_dtc);

```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcClearDiagnosticInformation_2013")]
static uds_status SvcClearDiagnosticInformation_2013(
[MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    UInt32 group_of_dtc);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcClearDiagnosticInformation_2013")>
Public Shared Function SvcClearDiagnosticInformation_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal group_of_dtc As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------------|--|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |
| group_of_dtc | A three-byte value indicating the group of DTCs (e.g. powertrain, body, chassis) or the particular DTC to be cleared (see <code>ClearDiagnosticInformation Group of DTC Definitions</code> on page 767). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcClearDiagnosticInformation_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ClearDiagnosticInformation message
result = UDSApi.SvcClearDiagnosticInformation_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    out request, 0xF1A2B3);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ClearDiagnosticInformation message
result = UDSApi::SvcClearDiagnosticInformation_2013(PCANTP_HANDLE_USBBUS1, config, request,
    0xF1A2B3);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else

```

```

        // An error occurred
        MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical ClearDiagnosticInformation message
result = UDSApi.SvcClearDiagnosticInformation_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, &HF1A2B3)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;

```

```

config.nai.extension_addr := $0;
config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ClearDiagnosticInformation message
result := TUDSApi.SvcClearDiagnosticInformation_2013
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request, $F1A2B3);
if TUDSApi.StatusIsOk_2013(result) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248, [ClearDiagnosticInformation](#) Group of DTC Definitions on page 767, [SvcClearDiagnosticInformation_2020](#) on page 391.

Plain function version: [UDS_SvcClearDiagnosticInformation_2013](#) on page 700.

3.7.78 svcClearDiagnosticInformation_2020

Writes a UDS request according to the ClearDiagnosticInformation service's specifications with memory selection parameter (ISO-14229-1:2020). The ClearDiagnosticInformation service is used by the client to clear diagnostic information in one server's or multiple servers' memory.

Syntax

Pascal OO

```

class function SvcClearDiagnosticInformation_2020(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    group_of_dtc: UInt32;
    memory_selection: Byte
): uds_status;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcClearDiagnosticInformation_2020")]

```

```
public static extern uds_status SvcClearDiagnosticInformation_2020(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    UInt32 group_of_dtc,
    Byte memory_selection);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcClearDiagnosticInformation_2013")]
static uds_status SvcClearDiagnosticInformation_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    UInt32 group_of_dtc);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcClearDiagnosticInformation_2020")>
Public Shared Function SvcClearDiagnosticInformation_2020(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal group_of_dtc As UInt32,
    ByVal memory_selection As Byte) As uds_status
End Function
```

Parameters

| Parameter | Description |
|------------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| group_of_dtc | A three-byte value indicating the group of DTCs (e.g. powertrain, body, chassis) or the particular DTC to be cleared (see ClearDiagnosticInformation Group of DTC Definitions on page 767). |
| memory_selection | User defined DTC memory. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with Reset_2013 (see Reset_2013 on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see MsgFree_2013 on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see [MsgFree_2013](#) on page 214).

Example

The following example shows the use of the service method [SvcClearDiagnosticInformation_2020](#) on the channel [PCANTP_HANDLE_USBBUS1](#). A UDS physical service request is transmitted, and the [WaitForService_2013](#) method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ClearDiagnosticInformation message
result = UDSApi.SvcClearDiagnosticInformation_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    out request, 0xF1A2B3, 0x42);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
```

```

config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ClearDiagnosticInformation message
result = UDSApi::SvcClearDiagnosticInformation_2020(PCANTP_HANDLE_USBBUS1, config, request,
    0xF1A2B3, 0x42);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical ClearDiagnosticInformation message
result = UDSApi.SvcClearDiagnosticInformation_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, &HF1A2B3, &H42)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
  result: uds_status;
  request: uds_msg;
  request_confirmation: uds_msg;
  response: uds_msg;
  config: uds_msgconfig;
begin
  FillChar(request, sizeof(request), 0);
  FillChar(request_confirmation, sizeof(request_confirmation), 0);
  FillChar(response, sizeof(response), 0);

  // Set request message configuration
  FillChar(config, sizeof(config), 0);
  config.can_id :=
    UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
  config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
  config.nai.extension_addr := $0;
  config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
  config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
  config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
  config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
  config.typem := uds_msgtype.PUDS_MSGTYPE_USDT;

  // Sends a physical ClearDiagnosticInformation message
  result := TUDSApi.SvcClearDiagnosticInformation_2020
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request, $F1A2B3, $42);
  if TUDSApi.StatusIsOk_2013(result) then
  begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
      @request, response, @request_confirmation);
  end;
  if TUDSApi.StatusIsOk_2013(result) then
  begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
  end
  else
  begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
  end;

  // Free structures
  TUDSApi.MsgFree_2013(request);
  TUDSApi.MsgFree_2013(response);
  TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248, [ClearDiagnosticInformation](#) Group of DTC Definitions on page 767, [SvcClearDiagnosticInformation_2013](#) on page 387.

Plain function version: [UDS_SvcClearDiagnosticInformation_2020](#) on page 702.

3.7.79 SvcReadDTCInformation_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information.

Syntax

Pascal OO

```
class function SvcReadDTCInformation_2013(
  channel: cantp_handle;
  request_config: uds_msgconfig;
  var out_msg_request: uds_msg;
  PUDS_SVC_PARAM_RDTCI_Type: uds_svc_param_rdtci;
  dtc_status_mask: Byte
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformation_2013")]
public static extern uds_status SvcReadDTCInformation_2013(
  [MarshalAs(UnmanagedType.U4)]
  cantp_handle channel,
  uds_msgconfig request_config,
  out uds_msg out_msg_request,
  [MarshalAs(UnmanagedType.U1)]
  uds_svc_param_rdtci PUDS_SVC_PARAM_RDTCI_Type,
  Byte dtc_status_mask);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformation_2013")]
static uds_status SvcReadDTCInformation_2013(
  [MarshalAs(UnmanagedType::U4)]
  cantp_handle channel,
  uds_msgconfig request_config,
  uds_msg %out_msg_request,
  [MarshalAs(UnmanagedType::U1)]
  uds_svc_param_rdtci PUDS_SVC_PARAM_RDTCI_Type,
  Byte dtc_status_mask);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcReadDTCInformation_2013")>
Public Shared Function SvcReadDTCInformation_2013(
  <MarshalAs(UnmanagedType.U4)>
  ByVal channel As cantp_handle,
  ByVal request_config As uds_msgconfig,
  ByRef out_msg_request As uds_msg,
  <MarshalAs(UnmanagedType.U1)>
  ByVal PUDS_SVC_PARAM_RDTCI_Type As uds_svc_param_rdtci,
  ByVal dtc_status_mask As Byte) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |

| Parameter | Description |
|------------------------------|---|
| PUDS_SVC_PARAM_RDTCL_Type | Subfunction parameter, ReadDTCInformation type, use one of the following: PUDS_SVC_PARAM_RDTCL_RNODTCBSM, PUDS_SVC_PARAM_RDTCL_RDTCSM, PUDS_SVC_PARAM_RDTCL_RMMDTCBSM, PUDS_SVC_PARAM_RDTCL_RNOMMDTCBSM, PUDS_SVC_PARAM_RDTCL_RNOOBDTCBSM, PUDS_SVC_PARAM_RDTCL_ROBDTCBSM. See also <code>uds_svc_param_rdtcl</code> on page 88. |
| <code>dtc_status_mask</code> | Contains eight DTC status bits. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

- This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).
- Only the following subfunctions are allowed:
 - `reportNumberOfDTCByStatusMask`,
 - `reportDTCByStatusMask`,
 - `reportMirrorMemoryDTCByStatusMask`,
 - `reportNumberOfMirrorMemoryDTCByStatusMask`,
 - `reportNumberOfEmissionsRelatedOBDDTCByStatusMask`,
 - `reportEmissionsRelatedOBDDTCByStatusMask`.

Example

The following example shows the use of the service method `SvcReadDTCInformation_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
```

```

config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformation message
result = UDSApi.SvcReadDTCInformation_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, UDSApi.uds_svc_param_rdtci.PUDS_SVC_PARAM_RDTCI_RDTCBSM, 0xF0);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformation message
result = UDSApi::SvcReadDTCInformation_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_rdtci::PUDS_SVC_PARAM_RDTCI_RDTCBSM, 0xF0);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical ReadDTCInformation message
result = UDSApi.SvcReadDTCInformation_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi.uds_svc_param_rdtci.PUDS_SVC_PARAM_RDTCI_RDTCBSM, &HF0)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=

```



```

cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformation message
result := TUDSApi.SvcReadDTCInformation_2013
  (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
   uds_svc_param_rdtci.PUDS_SVC_PARAM_RDTCI_RDTCSBDM, $F0);
if TUDSApi.StatusIsOk_2013(result) then
begin
  result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
  MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
  // An error occurred
  MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248, [uds_svc_param_rdtci](#) on page 88.

Plain function version: [UDS_SvcReadDTCInformation_2013](#) on page 704.

3.7.80 SvcReadDTCInformationRDTCSBDM_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. The subfunction reportDTCSnapshotRecordByDTCNumber ([PUDS_SVC_PARAM_RDTCI_RDTCSBDM](#)) is implicit.

Syntax

Pascal OO

```

class function SvcReadDTCInformationRDTCSBDM_2013(
  channel: cantp_handle;
  request_config: uds_msgconfig;
  var out_msg_request: uds_msg;
  dtc_mask: UInt32;
  dtc_snapshot_record_number: Byte
): uds_status;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationRDTCSBDM_2013")]
public static extern uds_status SvcReadDTCInformationRDTCSBDM_2013(
  [MarshalAs(UnmanagedType.U4)]
  cantp_handle channel,
  uds_msgconfig request_config,
  out uds_msg out_msg_request,
  UInt32 dtc_mask,
  Byte dtc_snapshot_record_number);

```


C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationRDTCSBDTC_2013")]
static uds_status SvcReadDTCInformationRDTCSBDTC_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    UInt32 dtc_mask,
    Byte dtc_snapshot_record_number);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcReadDTCInformationRDTCSBDTC_2013")>
Public Shared Function SvcReadDTCInformationRDTCSBDTC_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal dtc_mask As UInt32,
    ByVal dtc_snapshot_record_number As Byte) As uds_status
End Function
```

Parameters

| Parameter | Description |
|----------------------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| dtc_mask | A unique identification number (three-byte value) for a specific diagnostic trouble code (DTC). |
| dtc_snapshot_record_number | The number of the specific DTCSnapshot data records. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with Reset_2013 (see Reset_2013 on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see MsgFree_2013 on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcReadDTCInformationRDTCSBDTC_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRDTCSBDTC message
result = UDSApi.SvcReadDTCInformationRDTCSBDTC_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, out request, 0x00A1B2B3, 0x12);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRDTCSBDTC message
```

```

result = UDSApi::SvcReadDTCInformationRDTCSBDTC_2013(PCANTP_HANDLE_USBBUS1, config, request,
    0x00A1B2B3, 0x12);
if (UDSApI::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApI::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApI::MsgFree_2013(request);
UDSApI::MsgFree_2013(response);
UDSApI::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical ReadDTCInformationRDTCSBDTC message
result = UDSApi.SvcReadDTCInformationRDTCSBDTC_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, request, &HA1B2B3, &H12)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApI.MsgFree_2013(request)
UDSApI.MsgFree_2013(response)
UDSApI.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin

```

```

FillChar(request, sizeof(request), 0);
FillChar(request_confirmation, sizeof(request_confirmation), 0);
FillChar(response, sizeof(response), 0);

// Set request message configuration
FillChar(config, sizeof(config), 0);
config.can_id :=
    UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr := $0;
config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRDTCSBDTC message
result := TUDSApi.SvcReadDTCInformationRDTCSBDTC_2013
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request, $00A1B2B3, $12);
if TUDSApi.StatusIsOk_2013(result) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcReadDTCInformationRDTCSBDTC_2013](#) on page 706.

3.7.81 SvcReadDTCInformationRDTCSBRN_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. The subfunction reportDTCSnapshotByRecordNumber ([PUDS_SVC_PARAM_RDTCI_RDTCSBRN](#)) is implicit.

Syntax

Pascal OO

```

class function SvcReadDTCInformationRDTCSBRN_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;

```

```
    dtc_snapshot_record_number: Byte
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationRDTCSBRN_2013")]
public static extern uds_status SvcReadDTCInformationRDTCSBRN_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    Byte dtc_snapshot_record_number);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationRDTCSBRN_2013")]
static uds_status SvcReadDTCInformationRDTCSBRN_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    Byte dtc_snapshot_record_number);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcReadDTCInformationRDTCSBRN_2013")>
Public Shared Function SvcReadDTCInformationRDTCSBRN_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal dtc_snapshot_record_number As Byte) As uds_status
End Function
```

Parameters

| Parameter | Description |
|----------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| dtc_snapshot_record_number | The number of the specific DTCSnapshot data records. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with Reset_2013 (see Reset_2013 on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see MsgFree_2013 on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see [MsgFree_2013](#) on page 214).

Example

The following example shows the use of the service method [SvcReadDTCInformationRDTCSBRN_2013](#) on the channel [PCANTP_HANDLE_USBBUS1](#). A UDS physical service request is transmitted, and the [WaitForService_2013](#) method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRDTCSBRN message
result = UDSApi.SvcReadDTCInformationRDTCSBRN_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    out request, 0x12);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
```

```

config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRDTCSB RN message
result = UDSApi::SvcReadDTCInformationRDTCSB RN_2013(PCANTP_HANDLE_USBBUS1, config, request,
    0x12);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical ReadDTCInformationRDTCSB RN message
result = UDSApi.SvcReadDTCInformationRDTCSB RN_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, &H12)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```
var
```



```

result: uds_status;
request: uds_msg;
request_confirmation: uds_msg;
response: uds_msg;
config: uds_msgconfig;
begin
  FillChar(request, sizeof(request), 0);
  FillChar(request_confirmation, sizeof(request_confirmation), 0);
  FillChar(response, sizeof(response), 0);

  // Set request message configuration
  FillChar(config, sizeof(config), 0);
  config.can_id :=
    UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
  config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
  config.nai.extension_addr := $0;
  config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
  config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
  config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
  config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
  config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

  // Sends a physical ReadDTCInformationRDTCSBRN message
  result := TUDSApi.SvcReadDTCInformationRDTCSBRN_2013
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request, $12);
  if TUDSApi.StatusIsOk_2013(result) then
  begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
      @request, response, @request_confirmation);
  end;
  if TUDSApi.StatusIsOk_2013(result) then
  begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
  end
  else
  begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
  end;

  // Free structures
  TUDSApi.MsgFree_2013(request);
  TUDSApi.MsgFree_2013(response);
  TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcReadDTCInformationRDTCSBRN_2013](#) on page 707.

3.7.82 SvcReadDTCInformationReportExtended_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. Only

reportDTCExtendedDataRecordByDTCNumber and
reportMirrorMemoryDTCExtendedDataRecordByDTCNumber subfunctions are allowed.

Syntax

Pascal OO

```
class function SvcReadDTCInformationReportExtended_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    PUDS_SVC_PARAM_RDTCI_Type: uds_svc_param_rdtci;
    dtc_mask: UInt32;
    dtc_extended_data_record_number: Byte
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationReportExtended_2013")]
public static extern uds_status SvcReadDTCInformationReportExtended_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    [MarshalAs(UnmanagedType.U1)]
    uds_svc_param_rdtci PUDS_SVC_PARAM_RDTCI_Type,
    UInt32 dtc_mask,
    Byte dtc_extended_data_record_number);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationReportExtended_2013")]
static uds_status SvcReadDTCInformationReportExtended_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType::U1)]
    uds_svc_param_rdtci PUDS_SVC_PARAM_RDTCI_Type,
    UInt32 dtc_mask,
    Byte dtc_extended_data_record_number);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcReadDTCInformationReportExtended_2013")>
Public Shared Function SvcReadDTCInformationReportExtended_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.U1)>
    ByVal PUDS_SVC_PARAM_RDTCI_Type As uds_svc_param_rdtci,
    ByVal dtc_mask As UInt32,
    ByVal dtc_extended_data_record_number As Byte) As uds_status
End Function
```

Parameters

| Parameter | Description |
|---------------------------------|--|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |
| PUDS_SVC_PARAM_RDTCL_Type | Subfunction parameter, ReadDTCInformation type, use one of the following: PUDS_SVC_PARAM_RDTCL_RDTCEDRBDN, PUDS_SVC_PARAM_RDTCL_RMMDEDRBDN. See also <code>uds_svc_param_rdtci</code> on page 88. |
| dtc_mask | A unique identification number (three-byte value) for a specific diagnostic trouble code. |
| dtc_extended_data_record_number | The number of the specific DTCExtendedData record requested. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcReadDTCInformationReportExtended_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
```

```

config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationReportExtended message
result = UDSApi.SvcReadDTCInformationReportExtended_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, out request, UDSApi.uds_svc_param_rdtci.PUDS_SVC_PARAM_RDTCI_RDTCEDRBDN,
    0x00A1B2B3, 0x12);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationReportExtended message
result = UDSApi::SvcReadDTCInformationReportExtended_2013(PCANTP_HANDLE_USBBUS1, config,
    request, UDSApi::uds_svc_param_rdtci::PUDS_SVC_PARAM_RDTCI_RDTCEDRBDN, 0x00A1B2B3,
    0x12);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()

```

```

Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical ReadDTCInformationReportExtended message
result = UDSApi.SvcReadDTCInformationReportExtended_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, request, UDSApi.uds_svc_param_rdtci.PUDS_SVC_PARAM_RDTCI_RDTCEDRBDN, &HA1B2B3,
    &H12)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.type := uds_msgtype.PUDS_MSGTYPE_USDT;

    // Sends a physical ReadDTCInformationReportExtended message

```

```

result := TUDSApi.SvcReadDTCInformationReportExtended_2013
  (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
   uds_svc_param_rdtci.PUDS_SVC_PARAM_RDTCI_RDTCEDRBDN, $00A1B2B3, $12);
if TUDSApi.StatusIsOk_2013(result) then
begin
  result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
  MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
  // An error occurred
  MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248, [uds_svc_param_rdtci](#) on page 88.

Plain function version: [UDS_SvcReadDTCInformationReportExtended_2013](#) on page 709.

3.7.83 SvcReadDTCInformationReportSeverity_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. Only reportNumberOfDTCBySeverityMaskRecord and reportDTCSeverityInformation subfunctions are allowed.

Syntax

Pascal OO

```

class function SvcReadDTCInformationReportSeverity_2013(
  channel: cantp_handle;
  request_config: uds_msgconfig;
  var out_msg_request: uds_msg;
  PUDS_SVC_PARAM_RDTCI_Type: uds_svc_param_rdtci;
  dtc_severity_mask: Byte;
  dtc_status_mask: Byte
): uds_status;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationReportSeverity_2013")]
public static extern uds_status SvcReadDTCInformationReportSeverity_2013(
  [MarshalAs(UnmanagedType.U4)]
  cantp_handle channel,
  uds_msgconfig request_config,
  out uds_msg out_msg_request,
  [MarshalAs(UnmanagedType.U1)]
  uds_svc_param_rdtci PUDS_SVC_PARAM_RDTCI_Type,
  byte dtc_severity_mask,
  Byte dtc_status_mask);

```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationReportSeverity_2013")]
static uds_status SvcReadDTCInformationReportSeverity_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType::U1)]
    uds_svc_param_rdtci PUDS_SVC_PARAM_RDTCI_Type,
    Byte dtc_severity_mask,
    Byte dtc_status_mask);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcReadDTCInformationReportSeverity_2013")>
Public Shared Function SvcReadDTCInformationReportSeverity_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.U1)>
    ByVal PUDS_SVC_PARAM_RDTCI_Type As uds_svc_param_rdtci,
    ByVal dtc_severity_mask As Byte,
    ByVal dtc_status_mask As Byte) As uds_status
End Function
```

Parameters

| Parameter | Description |
|---------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| PUDS_SVC_PARAM_RDTCI_Type | Subfunction parameter, ReadDTCInformation type, use one of the following: PUDS_SVC_PARAM_RDTCI_RNODTCBSMR, PUDS_SVC_PARAM_RDTCI_RDTCBSMR See also uds_svc_param_rdtci on page 88. |
| dtc_severity_mask | A mask of eight (8) DTC severity bits (see uds_svc_param_rdtci_dtcsvm on page 91). |
| dtc_status_mask | A mask of eight (8) DTC status bits. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with Reset_2013 (see Reset_2013 on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see MsgFree_2013 on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see [MsgFree_2013](#) on page 214).

Example

The following example shows the use of the service method [SvcReadDTCInformationReportSeverity_2013](#) on the channel [PCANTP_HANDLE_USBBUS1](#). A UDS physical service request is transmitted, and the [WaitForService_2013](#) method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationReportSeverity message
result = UDSApi.SvcReadDTCInformationReportSeverity_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, out request, UDSApi.uds_svc_param_rdtci.PUDS_SVC_PARAM_RDTCI_RNODTCBSMR, 0xF1,
    0x12);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
```



```

config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationReportSeverity message
result = UDSApi::SvcReadDTCInformationReportSeverity_2013(PCANTP_HANDLE_USBBUS1, config,
    request, UDSApi::uds_svc_param_rdtci::PUDS_SVC_PARAM_RDTCI_RNODTCBSMR, 0xF1, 0x12);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical ReadDTCInformationReportSeverity message
result = UDSApi.SvcReadDTCInformationReportSeverity_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, request, UDSApi.uds_svc_param_rdtci.PUDS_SVC_PARAM_RDTCI_RNODTCBSMR, &HF1, &H12)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```


Pascal OO

```

var
  result: uds_status;
  request: uds_msg;
  request_confirmation: uds_msg;
  response: uds_msg;
  config: uds_msgconfig;
begin
  FillChar(request, sizeof(request), 0);
  FillChar(request_confirmation, sizeof(request_confirmation), 0);
  FillChar(response, sizeof(response), 0);

  // Set request message configuration
  FillChar(config, sizeof(config), 0);
  config.can_id :=
    UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
  config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
  config.nai.extension_addr := $0;
  config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
  config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
  config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
  config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
  config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

  // Sends a physical ReadDTCInformationReportSeverity message
  result := TUDSApi.SvcReadDTCInformationReportSeverity_2013
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
     uds_svc_param_rdtci.PUDS_SVC_PARAM_RDTCI_RNODTCBSMR, $F1, $12);
  if TUDSApi.StatusIsOk_2013(result) then
  begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
      @request, response, @request_confirmation);
  end;
  if TUDSApi.StatusIsOk_2013(result) then
  begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
  end
  else
  begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
  end;

  // Free structures
  TUDSApi.MsgFree_2013(request);
  TUDSApi.MsgFree_2013(response);
  TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248, [uds_svc_param_rdtci](#) on page 88, [uds_svc_param_rdtci_dtcsvm](#) on page 91.
Plain function version: [UDS_SvcReadDTCInformationReportSeverity_2013](#) on page 711.

3.7.84 SvcReadDTCInformationRSIODTC_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. The subfunction reportSeverityInformationOfDTC (PUDS_SVC_PARAM_RDTCL_RSIODTC) is implicit.

Syntax

Pascal OO

```
class function SvcReadDTCInformationRSIODTC_2013(
  channel: cantp_handle;
  request_config: uds_msgconfig;
  var out_msg_request: uds_msg;
  dtc_mask: UInt32
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationRSIODTC_2013")]
public static extern uds_status SvcReadDTCInformationRSIODTC_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    UInt32 dtc_mask);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationRSIODTC_2013")]
static uds_status SvcReadDTCInformationRSIODTC_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    UInt32 dtc_mask);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcReadDTCInformationRSIODTC_2013")>
Public Shared Function SvcReadDTCInformationRSIODTC_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal dtc_mask As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| dtc_mask | A unique identification number for a specific diagnostic trouble code. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcReadDTCInformationRSIODTC_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRSIODTC message
result = UDSApi.SvcReadDTCInformationRSIODTC_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    out request, 0xF1A1B2B3);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

```

```
// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRSIODTC message
result = UDSApi::SvcReadDTCInformationRSIODTC_2013(PCANTP_HANDLE_USBBUS1, config, request,
    0xF1A1B2B3);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);
```

Visual Basic

```
Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical ReadDTCInformationRSIODTC message
result = UDSApi.SvcReadDTCInformationRSIODTC_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, &HF1A1B2B3UI)
If UDSApi.StatusIsOk_2013(result) Then
```

```

    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

    // Sends a physical ReadDTCInformationRSIODTC message
    result := TUDSApi.SvcReadDTCInformationRSIODTC_2013
        (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request, $F1A1B2B3);
    if TUDSApi.StatusIsOk_2013(result) then
    begin
        result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
            @request, response, @request_confirmation);
    end;
    if TUDSApi.StatusIsOk_2013(result) then
    begin
        MessageBox(0, 'Response was received', 'Success', MB_OK);
    end
    else
    begin
        // An error occurred
        MessageBox(0, 'An error occurred', 'Error', MB_OK);
    end;

    // Free structures

```

```
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;
```

See also: `WaitForService_2013` on page 248.

Plain function version: `UDS_SvcReadDTCInformationRSIODTC_2013` on page 713.

3.7.85 SvcReadDTCInformationNoParam_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. Only these following subfunctions are allowed:

- reportSupportedDTC
- reportFirstTestFailedDTC
- reportFirstConfirmedDTC
- reportMostRecentTestFailedDTC
- reportMostRecentConfirmedDTC
- reportDTCFaultDetectionCounter
- reportDTCWithPermanentStatus
- reportDTCSnapshotIdentification

Syntax

Pascal OO

```
class function SvcReadDTCInformationNoParam_2013(
  channel: cantp_handle;
  request_config: uds_msgconfig;
  var out_msg_request: uds_msg;
  PUDS_SVC_PARAM_RDTCI_Type: uds_svc_param_rdtci
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationNoParam_2013")]
public static extern uds_status SvcReadDTCInformationNoParam_2013(
  [MarshalAs(UnmanagedType.U4)]
  cantp_handle channel,
  uds_msgconfig request_config,
  out uds_msg out_msg_request,
  [MarshalAs(UnmanagedType.U1)]
  uds_svc_param_rdtci PUDS_SVC_PARAM_RDTCI_Type);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationNoParam_2013")]
static uds_status SvcReadDTCInformationNoParam_2013(
  [MarshalAs(UnmanagedType.U4)]
  cantp_handle channel,
  uds_msgconfig request_config,
  uds_msg %out_msg_request,
  [MarshalAs(UnmanagedType.U1)]
  uds_svc_param_rdtci PUDS_SVC_PARAM_RDTCI_Type);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcReadDTCInformationNoParam_2013")>
Public Shared Function SvcReadDTCInformationNoParam_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.U1)>
    ByVal PUDS_SVC_PARAM_RDTCI_Type As uds_svc_param_rdtci) As uds_status
End Function
```

Parameters

| Parameter | Description |
|---------------------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| PUDS_SVC_PARAM_RDTCI_Type | Subfunction parameter, ReadDTCInformation type, use one of the following: PUDS_SVC_PARAM_RDTCI_RFTDTC, PUDS_SVC_PARAM_RDTCI_RFCDDTC, PUDS_SVC_PARAM_RDTCI_RMRTDTC, PUDS_SVC_PARAM_RDTCI_RMRCDDTC, PUDS_SVC_PARAM_RDTCI_RSUPDTC, PUDS_SVC_PARAM_RDTCI_RDTWCPS, PUDS_SVC_PARAM_RDTCI_RDTCSI. See also uds_svc_param_rdtci on page 88. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with Reset_2013 (see Reset_2013 on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see MsgFree_2013 on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcReadDTCInformationNoParam_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#


```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationNoParam message
result = UDSApi.SvcReadDTCInformationNoParam_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    out request, UDSApi.uds_svc_param_rdtci.PUDS_SVC_PARAM_RDTCI_RSUPDTC);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationNoParam message
result = UDSApi::SvcReadDTCInformationNoParam_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_rdtci::PUDS_SVC_PARAM_RDTCI_RSUPDTC);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred

```



```

        MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical ReadDTCInformationNoParam message
result = UDSApi.SvcReadDTCInformationNoParam_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, UDSApi.uds_svc_param_rdtci.PUDS_SVC_PARAM_RDTCI_RSUPDTC)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);

```

```

config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr := $0;
config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationNoParam message
result := TUDSApi.SvcReadDTCInformationNoParam_2013
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    uds_svc_param_rdtci.PUDS_SVC_PARAM_RDTCI_RSUPDTC);
if TUDSApi.StatusIsOk_2013(result) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248, [uds_svc_param_rdtci](#) on page 88.

Plain function version: [UDS_SvcReadDTCInformationNoParam_2013](#) on page 714.

3.7.86 SvcReadDTCInformationRDTCEDBR_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. The subfunction [reportDTCExtDataRecordByRecordNumber](#) ([PUDS_SVC_PARAM_RDTCI_RDTCEDBR](#)) is implicit.

Syntax

Pascal OO

```

class function SvcReadDTCInformationRDTCEDBR_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    dtc_extended_data_record_number: Byte
): uds_status;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationRDTCEDBR_2013")]

```

```
public static extern uds_status SvcReadDTCInformationRDTCEDBR_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    Byte dtc_extended_data_record_number);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationRDTCEDBR_2013")]
static uds_status SvcReadDTCInformationRDTCEDBR_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    Byte dtc_extended_data_record_number);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcReadDTCInformationRDTCEDBR_2013")>
Public Shared Function SvcReadDTCInformationRDTCEDBR_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal dtc_extended_data_record_number As Byte) As uds_status
End Function
```

Parameters

| Parameter | Description |
|---------------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| dtc_extended_data_record_number | DTC extended data record number. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with Reset_2013 (see Reset_2013 on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see MsgFree_2013 on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcReadDTCInformationRDTCEDBR_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();
Byte dtc_extended_data_record_number = 0x12;

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical SvcReadDTCInformationRDTCEDBR message
result = UDSApi.SvcReadDTCInformationRDTCEDBR_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    out request, dtc_extended_data_record_number);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};
Byte dtc_extended_data_record_number = 0x12;

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;
```

```
// Sends a physical SvcReadDTCInformationRDTCEDBR message
result = UDSApi::SvcReadDTCInformationRDTCEDBR_2013(cantp_handle::PCANTP_HANDLE_USBBUS1,
    config, request, dtc_extended_data_record_number);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(cantp_handle::PCANTP_HANDLE_USBBUS1, request,
        response, request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);
```

Visual Basic

```
Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()
Dim dtc_extended_data_record_number As Byte = &H12

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical SvcReadDTCInformationRDTCEDBR message
result = UDSApi.SvcReadDTCInformationRDTCEDBR_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, dtc_extended_data_record_number)
If (UDSApi.StatusIsOk_2013(result)) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If (UDSApi.StatusIsOk_2013(result)) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)
```

Pascal OO

```
var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
```

```

response: uds_msg;
config: uds_msgconfig;
dtc_extended_data_record_number: Byte;
begin
    dtc_extended_data_record_number := $12;
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.type := uds_msgtype.PUDS_MSGTYPE_USDT;

    // Sends a physical SvcReadDTCInformationRDTCEDBR message
    result := TUDSApi.SvcReadDTCInformationRDTCEDBR_2013(PCANTP_HANDLE_USBBUS1,
        config, request, dtc_extended_data_record_number);
    if (TUDSApi.StatusIsOk_2013(result)) then
    begin
        result := TUDSApi.WaitForService_2013(PCANTP_HANDLE_USBBUS1, @request,
            response, @request_confirmation);
    end;
    if (TUDSApi.StatusIsOk_2013(result)) then
    begin
        MessageBox(0, 'Response was received', 'Success', MB_OK);
    end
    else
    begin
        // An error occurred
        MessageBox(0, 'An error occurred', 'Error', MB_OK);
    end;

    // Free structures
    TUDSApi.MsgFree_2013(request);
    TUDSApi.MsgFree_2013(response);
    TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcReadDTCInformationRDTCEDBR_2013](#) on page 716.

3.7.87 SvcReadDTCInformationRUDMDTCBSM_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. The subfunction reportUserDefMemoryDTCByStatusMask (PUDS_SVC_PARAM_RDTCI_RUDMDTCBSM) is implicit.

Syntax

Pascal OO

```
class function SvcReadDTCInformationRUDMDTCBSM_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    dtc_status_mask: Byte;
    memory_selection: Byte
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationRUDMDTCBSM_2013")]
public static extern uds_status SvcReadDTCInformationRUDMDTCBSM_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    Byte dtc_status_mask,
    Byte memory_selection);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationRUDMDTCBSM_2013")]
static uds_status SvcReadDTCInformationRUDMDTCBSM_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    Byte dtc_status_mask,
    Byte memory_selection);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcReadDTCInformationRUDMDTCBSM_2013")>
Public Shared Function SvcReadDTCInformationRUDMDTCBSM_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal dtc_status_mask As Byte,
    ByVal memory_selection As Byte) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| dtc_status_mask | A mask of eight (8) DTC status bits. |

| Parameter | Description |
|------------------|-------------------|
| memory_selection | Memory selection. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcReadDTCInformationRUDMDTCBSM_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();
Byte dtc_status_mask = 0x12;
Byte memory_selection = 0x34;

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRUDMDTCBSM message
result = UDSApi.SvcReadDTCInformationRUDMDTCBSM_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, out request, dtc_status_mask, memory_selection);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))

```



```

    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};
Byte dtc_status_mask = 0x12;
Byte memory_selection = 0x34;

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRUDMDTCBSM message
result = UDSApi::SvcReadDTCInformationRUDMDTCBSM_2013(cantp_handle::PCANTP_HANDLE_USBBUS1,
    config, request, dtc_status_mask, memory_selection);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(cantp_handle::PCANTP_HANDLE_USBBUS1, request,
        response, request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()
Dim dtc_status_mask As Byte = &H12
Dim memory_selection As Byte = &H34

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL

```

```

config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical ReadDTCInformationRUDMDTCBSM message
result = UDSApi.SvcReadDTCInformationRUDMDTCBSM_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, request, dtc_status_mask, memory_selection)
If (UDSApi.StatusIsOk_2013(result)) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If (UDSApi.StatusIsOk_2013(result)) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    dtc_status_mask: Byte;
    memory_selection: Byte;
begin
    dtc_status_mask := $12;
    memory_selection := $34;
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.type := uds_msgtype.PUDS_MSGTYPE_USDT;

    // Sends a physical ReadDTCInformationRUDMDTCBSM message
    result := TUDSApi.SvcReadDTCInformationRUDMDTCBSM_2013(PCANTP_HANDLE_USBBUS1,
        config, request, dtc_status_mask, memory_selection);
    if (TUDSApi.StatusIsOk_2013(result)) then
        begin

```

```

    result := TUDSApi.WaitForService_2013(PCANTP_HANDLE_USBBUS1, @request,
        response, @request_confirmation);
end;
if (TUDSApi.StatusIsOk_2013(result)) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: `WaitForService_2013` on page 248.

Plain function version: `UDS_SvcReadDTCInformationRUDMDTCSSBDTC_2013` on page 718.

3.7.88 SvcReadDTCInformationRUDMDTCSSBDTC_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. The subfunction `reportUserDefMemoryDTCSnapshotRecordByDTCNumber` (`PUDS_SVC_PARAM_RDTCL_RUDMDTCSSBDTC`) is implicit.

Syntax

Pascal OO

```

class function SvcReadDTCInformationRUDMDTCSSBDTC_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    dtc_mask: UInt32;
    user_def_dtc_snapshot_record_number: Byte;
    memory_selection: Byte
): uds_status;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationRUDMDTCSSBDTC_2013")]
public static extern uds_status SvcReadDTCInformationRUDMDTCSSBDTC_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    UInt32 dtc_mask,
    Byte user_def_dtc_snapshot_record_number,
    Byte memory_selection);

```

C++ / CLR

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationRUDMDTCSSBDTC_2013")]
static uds_status SvcReadDTCInformationRUDMDTCSSBDTC_2013(
    [MarshalAs(UnmanagedType.U4)]

```

```

cantp_handle channel,
uds_msgconfig request_config,
uds_msg %out_msg_request,
UInt32 dtc_mask,
Byte user_def_dtc_snapshot_record_number,
Byte memory_selection);

```

Visual Basic

```

<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcReadDTCInformationRUDMDTCSSBDTC_2013")>
Public Shared Function SvcReadDTCInformationRUDMDTCSSBDTC_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal dtc_mask As UInt32,
    ByVal user_def_dtc_snapshot_record_number As Byte,
    ByVal memory_selection As Byte) As uds_status
End Function

```

Parameters

| Parameter | Description |
|-------------------------------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| dtc_mask | A unique identification number (three-byte value) for a specific diagnostic trouble code. |
| user_def_dtc_snapshot_record_number | User DTC snapshot record number. |
| memory_selection | Memory selection. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with Reset_2013 (see Reset_2013 on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see MsgFree_2013 on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcReadDTCInformationRUDMDTCSSBDTC_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();
UInt32 dtc_mask = 0x12345678;
Byte user_def_dtc_snapshot_record_number = 0x9A;
Byte memory_selection = 0xBC;

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRUDMDTCSSBDTC message
result = UDSApi.SvcReadDTCInformationRUDMDTCSSBDTC_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, out request, dtc_mask, user_def_dtc_snapshot_record_number, memory_selection);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};
UInt32 dtc_mask = 0x12345678;
Byte user_def_dtc_snapshot_record_number = 0x9A;
Byte memory_selection = 0xBC;

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRUDMDTCSSBDTC message
```

```

result = UDSApi::SvcReadDTCInformationRUDMDTCSSBDTC_2013(cantp_handle::PCANTP_HANDLE_USBBUS1,
    config, request, dtc_mask, user_def_dtc_snapshot_record_number, memory_selection);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(cantp_handle::PCANTP_HANDLE_USBBUS1, request,
        response, request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()
Dim dtc_mask As UInt32 = &H12345678
Dim user_def_dtc_snapshot_record_number As Byte = &H9A
Dim memory_selection As Byte = &HBC

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical ReadDTCInformationRUDMDTCSSBDTC message
result = UDSApi.SvcReadDTCInformationRUDMDTCSSBDTC_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, request, dtc_mask, user_def_dtc_snapshot_record_number, memory_selection)
If (UDSApi.StatusIsOk_2013(result)) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If (UDSApi.StatusIsOk_2013(result)) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;

```

```

response: uds_msg;
config: uds_msgconfig;
dtc_mask: UInt32;
user_def_dtc_snapshot_record_number: Byte;
memory_selection: Byte;
begin
    dtc_mask := $12345678;
    user_def_dtc_snapshot_record_number := $9A;
    memory_selection := $BC;
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

    // Sends a physical ReadDTCInformationRUDMDTCSSBDTC message
    result := TUDSApi.SvcReadDTCInformationRUDMDTCSSBDTC_2013
        (PCANTP_HANDLE_USBBUS1, config, request, dtc_mask,
        user_def_dtc_snapshot_record_number, memory_selection);
    if (TUDSApi.StatusIsOk_2013(result)) then
    begin
        result := TUDSApi.WaitForService_2013(PCANTP_HANDLE_USBBUS1, @request,
        response, @request_confirmation);
    end;
    if (TUDSApi.StatusIsOk_2013(result)) then
    begin
        MessageBox(0, 'Response was received', 'Success', MB_OK);
    end
    else
    begin
        // An error occurred
        MessageBox(0, 'An error occurred', 'Error', MB_OK);
    end;

    // Free structures
    TUDSApi.MsgFree_2013(request);
    TUDSApi.MsgFree_2013(response);
    TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcReadDTCInformationRUDMDTCSSBDTC_2013](#) on page 720.

3.7.89 SvcReadDTCInformationRUDMDTCEDRBDN_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. The subfunction reportUserDefMemoryDTCExtDataRecordByDTCNumber (PUDS_SVC_PARAM_RDTCL_RUDMDTCEDRBDN) is implicit.

Syntax

Pascal OO

```
class function SvcReadDTCInformationRUDMDTCEDRBDN_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    dtc_mask: UInt32;
    dtc_extended_data_record_number: Byte;
    memory_selection: Byte
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationRUDMDTCEDRBDN_2013")]
public static extern uds_status SvcReadDTCInformationRUDMDTCEDRBDN_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    UInt32 dtc_mask,
    Byte dtc_extended_data_record_number,
    Byte memory_selection);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationRUDMDTCEDRBDN_2013")]
static uds_status SvcReadDTCInformationRUDMDTCEDRBDN_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    UInt32 dtc_mask,
    Byte dtc_extended_data_record_number,
    Byte memory_selection);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcReadDTCInformationRUDMDTCEDRBDN_2013")>
Public Shared Function SvcReadDTCInformationRUDMDTCEDRBDN_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal dtc_mask As UInt32,
    ByVal dtc_extended_data_record_number As Byte,
    ByVal memory_selection As Byte) As uds_status
End Function
```


Parameters

| Parameter | Description |
|---------------------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |
| dtc_mask | A unique identification number (three-byte value) for a specific diagnostic trouble code. |
| dtc_extended_data_record_number | DTC extended data record number. |
| memory_selection | Memory selection. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcReadDTCInformationRUDMDTCEDRBDN_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();
UInt32 dtc_mask = 0x12345678;
Byte dtc_extended_data_record_number = 0x9A;
Byte memory_selection = 0xBC;

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;

```

```

config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRUDMDTCEDRBDN message
result = UDSApi.SvcReadDTCInformationRUDMDTCEDRBDN_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, out request, dtc_mask, dtc_extended_data_record_number, memory_selection);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};
UInt32 dtc_mask = 0x12345678;
Byte dtc_extended_data_record_number = 0x9A;
Byte memory_selection = 0xBC;

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRUDMDTCEDRBDN message
result = UDSApi::SvcReadDTCInformationRUDMDTCEDRBDN_2013(cantp_handle::PCANTP_HANDLE_USBBUS1,
    config, request, dtc_mask, dtc_extended_data_record_number, memory_selection);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(cantp_handle::PCANTP_HANDLE_USBBUS1, request,
        response, request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()

```

```

Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()
Dim dtc_mask As UInt32 = &H12345678
Dim dtc_extended_data_record_number As Byte = &H9A
Dim memory_selection As Byte = &HBC

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical ReadDTCInformationRUDMDTCEDRBDN message
result = UDSApi.SvcReadDTCInformationRUDMDTCEDRBDN_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, request, dtc_mask, dtc_extended_data_record_number, memory_selection)
If (UDSApi.StatusIsOk_2013(result)) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If (UDSApi.StatusIsOk_2013(result)) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    dtc_mask: UInt32;
    dtc_extended_data_record_number: Byte;
    memory_selection: Byte;
begin
    dtc_mask := $12345678;
    dtc_extended_data_record_number := $9A;
    memory_selection := $BC;
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=

```

```

uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr :=
  UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
config.nai.target_addr :=
  UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config.nai.target_type :=
  cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRUDMDTCEDRBDN message
result := TUDSApi.SvcReadDTCInformationRUDMDTCEDRBDN_2013
  (PCANTP_HANDLE_USBBUS1, config, request, dtc_mask,
   dtc_extended_data_record_number, memory_selection);
if (TUDSApi.StatusIsOk_2013(result)) then
begin
  result := TUDSApi.WaitForService_2013(PCANTP_HANDLE_USBBUS1, @request,
    response, @request_confirmation);
end;
if (TUDSApi.StatusIsOk_2013(result)) then
begin
  MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
  // An error occurred
  MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcReadDTCInformationRUDMDTCEDRBDN_2013](#) on page 722.

3.7.90 SvcReadDTCInformationRDTCEDI_2020

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. The subfunction reportSupportedDTCExtDataRecord (PUDS_SVC_PARAM_RDTCEI_RDTCEDI) is implicit.

Syntax

Pascal OO

```

class function SvcReadDTCInformationRDTCEDI_2020(
  channel: cantp_handle;
  request_config: uds_msgconfig;
  var out_msg_request: uds_msg;
  dtc_extended_data_record_number: Byte
): uds_status;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationRDTCEDI_2020")]
public static extern uds_status SvcReadDTCInformationRDTCEDI_2020(
  [MarshalAs(UnmanagedType.U4)]

```

```
cantp_handle channel,
uds_msgconfig request_config,
out uds_msg out_msg_request,
Byte dtc_extended_data_record_number);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationRDTCEdi_2020")]
static uds_status SvcReadDTCInformationRDTCEdi_2020(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    Byte dtc_extended_data_record_number);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcReadDTCInformationRDTCEdi_2020")>
Public Shared Function SvcReadDTCInformationRDTCEdi_2020(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal dtc_extended_data_record_number As Byte) As uds_status
End Function
```

Parameters

| Parameter | Description |
|---------------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| dtc_extended_data_record_number | DTC extended data record number. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with Reset_2013 (see Reset_2013 on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see MsgFree_2013 on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcReadDTCInformationRDTCEdi_2020` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();
Byte dtc_extended_data_record_number = 0x12;

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRDTCEdi message
result = UDSApi.SvcReadDTCInformationRDTCEdi_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    out request, dtc_extended_data_record_number);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};
Byte dtc_extended_data_record_number = 0x12;

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;
```

```
// Sends a physical ReadDTCInformationRDTCEdi message
result = UDSApi::SvcReadDTCInformationRDTCEdi_2020(cantp_handle::PCANTP_HANDLE_USBBUS1, config,
    request, dtc_extended_data_record_number);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(cantp_handle::PCANTP_HANDLE_USBBUS1, request,
        response, request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);
```

Visual Basic

```
Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()
Dim dtc_extended_data_record_number As Byte = &H12

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical ReadDTCInformationRDTCEdi message
result = UDSApi.SvcReadDTCInformationRDTCEdi_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, dtc_extended_data_record_number)
If (UDSApi.StatusIsOk_2013(result)) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If (UDSApi.StatusIsOk_2013(result)) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)
```

Pascal OO

```
var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
```



```

response: uds_msg;
config: uds_msgconfig;
dtc_extended_data_record_number: Byte;
begin
    dtc_extended_data_record_number := $12;
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

    // Sends a physical ReadDTCInformationRDTCEdi message
    result := TUDSApi.SvcReadDTCInformationRDTCEdi_2020(PCANTP_HANDLE_USBBUS1,
        config, request, dtc_extended_data_record_number);
    if (TUDSApi.StatusIsOk_2013(result)) then
    begin
        result := TUDSApi.WaitForService_2013(PCANTP_HANDLE_USBBUS1, @request,
            response, @request_confirmation);
    end;
    if (TUDSApi.StatusIsOk_2013(result)) then
    begin
        MessageBox(0, 'Response was received', 'Success', MB_OK);
    end
    else
    begin
        // An error occurred
        MessageBox(0, 'An error occurred', 'Error', MB_OK);
    end;

    // Free structures
    TUDSApi.MsgFree_2013(request);
    TUDSApi.MsgFree_2013(response);
    TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcReadDTCInformationRDTCEdi_2020](#) on page 724.

3.7.91 SvcReadDTCInformationRWWHOBDDTCBMR_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. The subfunction reportRWWHOBDDTCByMaskRecord (PUDS_SVC_PARAM_RDTCI_RWWHOBDDTCBMR) is implicit.

Syntax

Pascal OO

```
class function SvcReadDTCInformationRWWHOBDDTCBMR_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    functional_group_identifier: Byte;
    dtc_status_mask: Byte;
    dtc_severity_mask: Byte
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationRWWHOBDDTCBMR_2013")]
public static extern uds_status SvcReadDTCInformationRWWHOBDDTCBMR_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    Byte functional_group_identifier,
    Byte dtc_status_mask,
    Byte dtc_severity_mask);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationRWWHOBDDTCBMR_2013")]
static uds_status SvcReadDTCInformationRWWHOBDDTCBMR_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    Byte functional_group_identifier,
    Byte dtc_status_mask,
    Byte dtc_severity_mask);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcReadDTCInformationRWWHOBDDTCBMR_2013")>
Public Shared Function SvcReadDTCInformationRWWHOBDDTCBMR_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal functional_group_identifier As Byte,
    ByVal dtc_status_mask As Byte,
    ByVal dtc_severity_mask As Byte) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |
| functional_group_identifier | Functional group identifier. |
| dtc_status_mask | A mask of eight (8) DTC status bits. |
| dtc_severity_mask | A mask of eight (8) DTC severity bits (see <code>uds_svc_param_rdtci_dtcsvm</code> on page 91). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcReadDTCInformationRWWHOBDDTCBMR_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();
Byte functional_group_identifier = 0x12;
Byte dtc_status_mask = 0x34;
Byte dtc_severity_mask = 0x78;

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
```

```

config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRWWHOBDDTCBMR message
result = UDSApi.SvcReadDTCInformationRWWHOBDDTCBMR_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, out request, functional_group_identifider, dtc_status_mask, dtc_severity_mask);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};
Byte functional_group_identifider = 0x12;
Byte dtc_status_mask = 0x34;
Byte dtc_severity_mask = 0x78;

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRWWHOBDDTCBMR message
result = UDSApi::SvcReadDTCInformationRWWHOBDDTCBMR_2013(cantp_handle::PCANTP_HANDLE_USBBUS1,
    config, request, functional_group_identifider, dtc_status_mask, dtc_severity_mask);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(cantp_handle::PCANTP_HANDLE_USBBUS1, request,
        response, request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()

```

```

Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()
Dim functional_group_identifier As Byte = &H12
Dim dtc_status_mask As Byte = &H34
Dim dtc_severity_mask As Byte = &H78

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical ReadDTCInformationRWWHOBDDTCBMR message
result = UDSApi.SvcReadDTCInformationRWWHOBDDTCBMR_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, request, functional_group_identifier, dtc_status_mask, dtc_severity_mask)
If (UDSApi.StatusIsOk_2013(result)) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If (UDSApi.StatusIsOk_2013(result)) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    functional_group_identifier: Byte;
    dtc_status_mask: Byte;
    dtc_severity_mask: Byte;
begin
    functional_group_identifier := $12;
    dtc_status_mask := $34;
    dtc_severity_mask := $78;
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=

```

```

uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr :=
  UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
config.nai.target_addr :=
  UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config.nai.target_type :=
  cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRWWHOBDDTCBMR message
result := TUDSApi.SvcReadDTCInformationRWWHOBDDTCBMR_2013
  (PCANTP_HANDLE_USBBUS1, config, request, functional_group_identifier,
   dtc_status_mask, dtc_severity_mask);
if (TUDSApi.StatusIsOk_2013(result)) then
begin
  result := TUDSApi.WaitForService_2013(PCANTP_HANDLE_USBBUS1, @request,
    response, @request_confirmation);
end;
if (TUDSApi.StatusIsOk_2013(result)) then
begin
  MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
  // An error occurred
  MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcReadDTCInformationRWWHOBDDTCBMR_2013](#) on page 725.

3.7.92 SvcReadDTCInformationRWWHOBDDTCWPS_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. The subfunction reportRWWHOBDDTCWithPermanentStatus (PUDS_SVC_PARAM_RDTCI_RWWHOBDDTCWPS) is implicit.

Syntax

Pascal OO

```

class function SvcReadDTCInformationRWWHOBDDTCWPS_2013(
  channel: cantp_handle;
  request_config: uds_msgconfig;
  var out_msg_request: uds_msg;
  functional_group_identifier: Byte
): uds_status;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationRWWHOBDDTCWPS_2013")]
public static extern uds_status SvcReadDTCInformationRWWHOBDDTCWPS_2013(
  [MarshalAs(UnmanagedType.U4)]
  cantp_handle channel,

```

```
uds_msgconfig request_config,
out uds_msg out_msg_request,
Byte functional_group_identifier);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationRWWHOBDDTCWPS_2013")]
static uds_status SvcReadDTCInformationRWWHOBDDTCWPS_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    Byte functional_group_identifier);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcReadDTCInformationRWWHOBDDTCWPS_2013")>
Public Shared Function SvcReadDTCInformationRWWHOBDDTCWPS_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal functional_group_identifier As Byte) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| functional_group_identifier | Functional group identifier. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with Reset_2013 (see Reset_2013 on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see MsgFree_2013 on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcReadDTCInformationRWWHOBDDTCWPS_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();
Byte functional_group_identifier = 0x12;

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRWWHOBDDTCWPS message
result = UDSApi.SvcReadDTCInformationRWWHOBDDTCWPS_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, out request, functional_group_identifier);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};
Byte functional_group_identifier = 0x12;

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
```



```

config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRWWHOBDDTCWPS message
result = UDSApi::SvcReadDTCInformationRWWHOBDDTCWPS_2013(cantp_handle::PCANTP_HANDLE_USBBUS1,
    config, request, functional_group_identifiler);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(cantp_handle::PCANTP_HANDLE_USBBUS1, request,
        response, request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()
Dim functional_group_identifiler As Byte = &H12

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical ReadDTCInformationRWWHOBDDTCWPS message
result = UDSApi.SvcReadDTCInformationRWWHOBDDTCWPS_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, request, functional_group_identifiler)
If (UDSApi.StatusIsOk_2013(result)) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If (UDSApi.StatusIsOk_2013(result)) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;

```



```

request_confirmation: uds_msg;
response: uds_msg;
config: uds_msgconfig;
functional_group_identifier: Byte;
begin
    functional_group_identifier := $12;
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

    // Sends a physical ReadDTCInformationRWWHOBDDTCWPS message
    result := TUDSApi.SvcReadDTCInformationRWWHOBDDTCWPS_2013
        (PCANTP_HANDLE_USBBUS1, config, &request, functional_group_identifier);
    if (TUDSApi.StatusIsOk_2013(result)) then
    begin
        result := TUDSApi.WaitForService_2013(PCANTP_HANDLE_USBBUS1, @request,
            &response, @request_confirmation);
    end;
    if (TUDSApi.StatusIsOk_2013(result)) then
    begin
        MessageBox(0, 'Response was received', 'Success', MB_OK);
    end
    else
    begin
        // An error occurred
        MessageBox(0, 'An error occurred', 'Error', MB_OK);
    end;

    // Free structures
    TUDSApi.MsgFree_2013(request);
    TUDSApi.MsgFree_2013(response);
    TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcReadDTCInformationRWWHOBDDTCWPS_2013](#) on page 727.

3.7.93 SvcReadDTCInformationRDTCBRGI_2020

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. The subfunction reportDTCInformationByDTCReadinessGroupIdentifier (PUDS_SVC_PARAM_RDTCI_RDTCBRGI) is implicit.

Syntax

Pascal OO

```
class function SvcReadDTCInformationRDTCBRGI_2020(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    functional_group_identifier: Byte;
    dtc_readiness_group_identifier: Byte
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationRDTCBRGI_2020")]
public static extern uds_status SvcReadDTCInformationRDTCBRGI_2020(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    Byte functional_group_identifier,
    Byte dtc_readiness_group_identifier);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcReadDTCInformationRDTCBRGI_2020")]
static uds_status SvcReadDTCInformationRDTCBRGI_2020(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    Byte functional_group_identifier,
    Byte dtc_readiness_group_identifier);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcReadDTCInformationRDTCBRGI_2020")>
Public Shared Function SvcReadDTCInformationRDTCBRGI_2020(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal functional_group_identifier As Byte,
    ByVal dtc_readiness_group_identifier As Byte) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |

| Parameter | Description |
|--------------------------------|---------------------------------|
| functional_group_identifier | Functional group identifier. |
| dtc_readiness_group_identifier | DTC readiness group identifier. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcReadDTCInformationRDTCBRGI_2020` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.



Note: It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();
Byte functional_group_identifier = 0x12;
Byte dtc_readiness_group_identifier = 0x34;

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRDTCBRGI message
result = UDSApi.SvcReadDTCInformationRDTCBRGI_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    out request, functional_group_identifier, dtc_readiness_group_identifier);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);

```

```

if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};
Byte functional_group_identifier = 0x12;
Byte dtc_readiness_group_identifier = 0x34;

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRDTCBRGI message
result = UDSApi::SvcReadDTCInformationRDTCBRGI_2020(cantp_handle::PCANTP_HANDLE_USBBUS1,
    config, request, functional_group_identifier, dtc_readiness_group_identifier);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(cantp_handle::PCANTP_HANDLE_USBBUS1, request,
        response, request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()
Dim functional_group_identifier As Byte = &H12
Dim dtc_readiness_group_identifier As Byte = &H34

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0

```

```

config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical ReadDTCInformationRDTCBRGI message
result = UDSApi.SvcReadDTCInformationRDTCBRGI_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, functional_group_identifier, dtc_readiness_group_identifier)
If (UDSApi.StatusIsOk_2013(result)) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If (UDSApi.StatusIsOk_2013(result)) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    functional_group_identifier: Byte;
    dtc_readiness_group_identifier: Byte;
begin
    functional_group_identifier := $12;
    dtc_readiness_group_identifier := $34;
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.type := uds_msgtype.PUDS_MSGTYPE_USDT;

    // Sends a physical ReadDTCInformationRDTCBRGI message
    result := TUDSApi.SvcReadDTCInformationRDTCBRGI_2020(PCANTP_HANDLE_USBBUS1,
        config, &request, functional_group_identifier,
        dtc_readiness_group_identifier);

```

```

if (TUDSApi.StatusIsOk_2013(result)) then
begin
    result := TUDSApi.WaitForService_2013(PCANTP_HANDLE_USBBUS1, @request,
        &response, @request_confirmation);
end;
if (TUDSApi.StatusIsOk_2013(result)) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```



See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcReadDTCInformationRDTCBRG1_2020](#) on page 729.

3.7.94 SvcInputOutputControlByIdentifier_2013

Writes a UDS request according to the InputOutputControlByIdentifier service's specifications. The InputOutputControlByIdentifier service is used by the client to substitute a value for an input signal, internal server (ECU) method and/or control an output (actuator) of an electronic system.

Overloads

| | Method | Description |
|---|--|---|
|  | SvcInputOutputControlByIdentifier_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_di, byte[], UInt32) | Writes to the transmit queue a request for UDS service InputOutputControlByIdentifier, without control enable mask. |
|  | SvcInputOutputControlByIdentifier_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_di, byte[], UInt32, byte[], UInt32) | Writes to the transmit queue a request for UDS service InputOutputControlByIdentifier, with control enable mask. |

Plain function version: [UDS_SvcInputOutputControlByIdentifier_2013](#) on page 731.

3.7.95 SvcInputOutputControlByIdentifier_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_di, byte[], UInt32)

Writes a UDS request according to the InputOutputControlByIdentifier service's specifications (without control enable mask). The InputOutputControlByIdentifier service is used by the client to substitute a value for an input signal, internal server (ECU) method and/or control an output (actuator) of an electronic system.

Syntax

Pascal OO

```

class function SvcInputOutputControlByIdentifier_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    data_identifier: uds_svc_param_di;

```

```

    control_option_record: PByte;
    control_option_record_size: UInt32
): uds_status; overload;

```

C#

```

public static uds_status SvcInputOutputControlByIdentifier_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    uds_svc_param_di data_identifier,
    byte[] control_option_record,
    UInt32 control_option_record_size);

```

C++ / CLR

```

static uds_status SvcInputOutputControlByIdentifier_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    uds_svc_param_di data_identifier,
    array<Byte> ^control_option_record,
    UInt32 control_option_record_size);

```

Visual Basic

```

Public Shared Function SvcInputOutputControlByIdentifier_2013(
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal data_identifier As uds_svc_param_di,
    ByVal control_option_record As Byte(),
    ByVal control_option_record_size As UInt32) As uds_status
End Function

```

Parameters

| Parameter | Description |
|----------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| data_identifier | A two-byte data identifier (see uds_svc_param_di on page 82). |
| control_option_record | First byte can be used as either an input output control parameter that describes how the server (ECU) shall control its inputs or outputs (see uds_svc_param_iocbi on page 92), or as an additional control state byte. |
| control_option_record_size | Size in bytes of the control option record buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is not valid. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | The mapping given in the message configuration is unknown. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_OVERFLOW | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data length. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcInputOutputControlByIdentifier_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
Byte[] control_option_record = new Byte[10];
UInt16 control_option_record_size = 10;
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Fill Data
for (int i = 0; i < control_option_record_size; i++)
{
    control_option_record[i] = (Byte)(Convert.ToByte('A') + i);
}

// Sends a physical InputOutputControlByIdentifier message
result = UDSApi.SvcInputOutputControlByIdentifier_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, out request, UDSApi.uds_svc_param_di.PUDS_SVC_PARAM_DI_SSECUSWVNDID,
    control_option_record, control_option_record_size);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
```



```

        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
array<Byte>^ control_option_record = gcnew array<Byte>(10);
UInt16 control_option_record_size = 10;
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Fill Data
for (int i = 0; i < control_option_record_size; i++)
{
    control_option_record[i] = 'A' + i;
}

// Sends a physical InputOutputControlByIdentifier message
result = UDSApi::SvcInputOutputControlByIdentifier_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_di::PUDS_SVC_PARAM_DI_SSECUSWVNDID, control_option_record,
    control_option_record_size);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()

```

```

Dim response As uds_msg = New uds_msg()
Dim control_option_record(10) As Byte
Dim control_option_record_size As UInt16 = 10
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Fill Data
Dim start_value As Char = "A"
For i As UInt32 = 0 To control_option_record_size - 1
    control_option_record(i) = Convert.ToByte(start_value) + i
Next

' Sends a physical InputOutputControlByIdentifier message
result = UDSApi.SvcInputOutputControlByIdentifier_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, request, UDSApi.uds_svc_param_di.PUDS_SVC_PARAM_DI_SSECUSWVNDID,
    control_option_record, control_option_record_size)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    control_option_record: array [0 .. 9] of Byte;
    control_option_record_size: UInt16;
    i: UInt32;
begin
    control_option_record_size := 10;
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);

```

```

config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr := $0;
config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Fill Data
for i := 0 to control_option_record_size - 1 do
begin
    control_option_record[i] := ($41 + i);
end;

// Sends a physical InputOutputControlByIdentifier message
result := TUDSApi.SvcInputOutputControlByIdentifier_2013
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    uds_svc_param_di.PUDS_SVC_PARAM_DI_SSECUSWVNDID, @control_option_record,
    control_option_record_size);
if TUDSApi.StatusIsOk_2013(result) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248, [uds_svc_param_iocbi](#) on page 92.

Plain function version: [UDS_SvcInputOutputControlByIdentifier_2013](#) on page 731.

3.7.96 SvcInputOutputControlByIdentifier_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_di, byte[], UInt32, byte[], UInt32)

Writes a UDS request according to the InputOutputControlByIdentifier service's specifications. The InputOutputControlByIdentifier service is used by the client to substitute a value for an input signal, internal server (ECU) method and/or control an output (actuator) of an electronic system.

Syntax

Pascal OO

```
class function SvcInputOutputControlByIdentifier_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    data_identifier: uds_svc_param_di;
    control_option_record: PByte;
    control_option_record_size: UInt32;
    control_enable_mask_record: PByte;
    control_enable_mask_record_size: UInt32
): uds_status; overload;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcInputOutputControlByIdentifier_2013")]
public static extern uds_status SvcInputOutputControlByIdentifier_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    [MarshalAs(UnmanagedType.U2)]
    uds_svc_param_di data_identifier,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 5)]
    byte[] control_option_record,
    UInt32 control_option_record_size,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 7)]
    byte[] control_enable_mask_record,
    UInt32 control_enable_mask_record_size);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcInputOutputControlByIdentifier_2013")]
static uds_status SvcInputOutputControlByIdentifier_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType::U2)]
    uds_svc_param_di data_identifier,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 5)]
    array<Byte> ^control_option_record,
    UInt32 control_option_record_size,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 7)]
    array<Byte> ^control_enable_mask_record,
    UInt32 control_enable_mask_record_size);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcInputOutputControlByIdentifier_2013")>
Public Shared Function SvcInputOutputControlByIdentifier_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.U2)>
    ByVal data_identifier As uds_svc_param_di,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=5)>
    ByVal control_option_record As Byte(),
    ByVal control_option_record_size As UInt32,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=7)>
    ByVal control_enable_mask_record As Byte(),
    ByVal control_enable_mask_record_size As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|---------------------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |
| data_identifier | A two-byte data identifier (see <code>uds_svc_param_di</code> on page 82). |
| control_option_record | First byte can be used as either an input output control parameter that describes how the server (ECU) shall control its inputs or outputs (see <code>uds_svc_param_iocbi</code> on page 92), or as an additional control state byte. |
| control_option_record_size | Size in bytes of the control option record buffer. |
| control_enable_mask_record | The control enable mask shall only be supported when the input output control parameter is used and the data identifier to be controlled consists of more than one parameter (i.e. the data identifier is bit-mapped or packed by definition). There shall be one bit in the control enable mask corresponding to each individual parameter defined within the data identifier. |
| control_enable_mask_record_size | Size in bytes of the control enable mask record buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_OVERFLOW</code> | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data length. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcInputOutputControlByIdentifier_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
Byte[] control_option_record = new Byte[10];
Byte[] control_enable_mask_record = new Byte[10];
UInt16 control_option_record_size = 10;
UInt16 control_enable_mask_record_size = 5;
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Fill Data
for (int i = 0; i < control_option_record_size; i++)
{
    control_option_record[i] = (Byte)(Convert.ToByte('A') + i);
    control_enable_mask_record[i] = (Byte)(10 + i);
}

// Sends a physical InputOutputControlByIdentifier message
result = UDSApi.SvcInputOutputControlByIdentifier_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, out request, UDSApi.uds_svc_param_di.PUDS_SVC_PARAM_DI_SSECUSWVNDID,
    control_option_record, control_option_record_size, control_enable_mask_record,
    control_enable_mask_record_size);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
```

```

uds_msg response = {};
array<Byte>^ control_option_record = gcnew array<Byte>(10);
array<Byte>^ control_enable_mask_record = gcnew array<Byte>(10);
UInt16 control_option_record_size = 10;
UInt16 control_enable_mask_record_size = 5;
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Fill Data
for (int i = 0; i < control_option_record_size; i++)
{
    control_option_record[i] = 'A' + i;
    control_enable_mask_record[i] = (10 + i);
}

// Sends a physical InputOutputControlByIdentifier message
result = UDSApi::SvcInputOutputControlByIdentifier_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_di::PUDS_SVC_PARAM_DI_SSECUSWVNDID, control_option_record,
    control_option_record_size, control_enable_mask_record,
    control_enable_mask_record_size);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim control_option_record(10) As Byte
Dim control_enable_mask_record(10) As Byte
Dim control_option_record_size As UInt16 = 10
Dim control_enable_mask_record_size As UInt16 = 5
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1

```



```

config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Fill Data
Dim start_value As Char = "A"
For i As UInt32 = 0 To control_option_record_size - 1
    control_option_record(i) = Convert.ToByte(start_value) + i
    control_enable_mask_record(i) = 10 + i
Next

' Sends a physical InputOutputControlByIdentifier message
result = UDSApi.SvcInputOutputControlByIdentifier_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, request, UDSApi.uds_svc_param_di.PUDS_SVC_PARAM_DI_SSECUSWVNDID,
    control_option_record, control_option_record_size, control_enable_mask_record,
    control_enable_mask_record_size)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    control_option_record: array [0 .. 9] of Byte;
    control_enable_mask_record: array [0 .. 9] of Byte;
    control_option_record_size: UInt16;
    control_enable_mask_record_size: UInt16;
    i: UInt32;
begin
    control_option_record_size := 10;
    control_enable_mask_record_size := 5;
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=

```



```

    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Fill Data
for i := 0 to control_option_record_size - 1 do
begin
    control_option_record[i] := ($41 + i);
    control_enable_mask_record[i] := (10 + i);
end;

// Sends a physical InputOutputControlByIdentifier message
result := TUDSApi.SvcInputOutputControlByIdentifier_2013
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    uds_svc_param_di.PUDS_SVC_PARAM_DI_SSECUSWVNDID, @control_option_record,
    control_option_record_size, @control_enable_mask_record,
    control_enable_mask_record_size);
if TUDSApi.StatusIsOk_2013(result) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```



See also: [WaitForService_2013](#) on page 248, [uds_svc_param_iocbj](#) on page 92.

Plain function version: [UDS_SvcInputOutputControlByIdentifier_2013](#) on page 731.

3.7.97 SvcRoutineControl_2013

Writes a UDS request according to the RoutineControl service's specifications. The RoutineControl service is used by the client to start/stop a routine and request routine results.

Overloads

| | Method | Description |
|---|--|--|
|  | SvcRoutineControl_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_rc, uds_svc_param_rc_rid) | Writes to the transmit queue a request for UDS service RoutineControl, without routine control options. |
|  | SvcRoutineControl_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_rc, uds_svc_param_rc_rid, byte[], UInt32) | Writes to the transmit queue a request for UDS service RoutineControl, with routine control options (only with start and stop routine subfunctions). |

Plain function version: [UDS_SvcRoutineControl_2013](#) on page 733.

3.7.98 SvcRoutineControl_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_rc, uds_svc_param_rc_rid)

Writes a UDS request according to the RoutineControl service's specifications, without routine control options (with all subfunctions except "start" and "stop"). The RoutineControl service is used by the client to start/stop a routine and request routine results.

Syntax

Pascal OO

```
class function SvcRoutineControl_2013(
  channel: cantp_handle;
  request_config: uds_msgconfig;
  var out_msg_request: uds_msg;
  routine_control_type: uds_svc_param_rc;
  routine_identifier: uds_svc_param_rc_rid
): uds_status; overload;
```

C#

```
public static uds_status SvcRoutineControl_2013(
  cantp_handle channel,
  uds_msgconfig request_config,
  out uds_msg out_msg_request,
  uds_svc_param_rc routine_control_type,
  uds_svc_param_rc_rid routine_identifier);
```

C++ / CLR

```
static uds_status SvcRoutineControl_2013(
  cantp_handle channel,
  uds_msgconfig request_config,
  uds_msg %out_msg_request,
  uds_svc_param_rc routine_control_type,
  uds_svc_param_rc_rid routine_identifier);
```

Visual Basic

```
Public Shared Function SvcRoutineControl_2013(
  ByVal channel As cantp_handle,
  ByVal request_config As uds_msgconfig,
  ByRef out_msg_request As uds_msg,
  ByVal routine_control_type As uds_svc_param_rc,
  ByVal routine_identifier As uds_svc_param_rc_rid) As uds_status
End Function
```

Parameters

| Parameter | Description |
|----------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| routine_control_type | Subfunction parameter: routine control type (see uds_svc_param_rc on page 93). |
| routine_identifier | Server local routine identifier (see uds_svc_param_rc_rid on page 94). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcRoutineControl_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical RoutineControl message
result = UDSApi.SvcRoutineControl_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out request,
    UDSApi.uds_svc_param_rc.PUDS_SVC_PARAM_RC_RRR, (UDSApi.uds_svc_param_rc_rid)0xF1A2);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

```

```
// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical RoutineControl message
result = UDSApi::SvcRoutineControl_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_rc::PUDS_SVC_PARAM_RC_RRR, (UDSApi::uds_svc_param_rc_rid)0xF1A2);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);
```

Visual Basic

```
Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical RoutineControl message
result = UDSApi.SvcRoutineControl_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi.uds_svc_param_rc.PUDS_SVC_PARAM_RC_RRR, &HF1A2)
```

```

If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApI.MsgFree_2013(request)
UDSApI.MsgFree_2013(response)
UDSApI.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

    // Sends a physical RoutineControl message
    result := TUDSApI.SvcRoutineControl_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        config, request, uds_svc_param_rc.PUDS_SVC_PARAM_RC_RRR,
        uds_svc_param_rc_rid($F1A2));
    if TUDSApI.StatusIsOk_2013(result) then
    begin
        result := TUDSApI.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
            @request, response, @request_confirmation);
    end;
    if TUDSApI.StatusIsOk_2013(result) then
    begin
        MessageBox(0, 'Response was received', 'Success', MB_OK);
    end
    else
    begin
        // An error occurred
        MessageBox(0, 'An error occurred', 'Error', MB_OK);
    end
end

```

```

end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248, [uds_svc_param_rc](#) on page 93, [uds_svc_param_rc_rid](#) on page 94.

Plain function version: [UDS_SvcRoutineControl_2013](#) on page 733.

3.7.99 SvcRoutineControl_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_rc, uds_svc_param_rc_rid, byte[], UInt32)

Writes a UDS request according to the RoutineControl service's specifications (only with start and stop routine subfunctions). The RoutineControl service is used by the client to start/stop a routine and request routine results.

Syntax

Pascal OO

```

class function SvcRoutineControl_2013(
  channel: cantp_handle;
  request_config: uds_msgconfig;
  var out_msg_request: uds_msg;
  routine_control_type: uds_svc_param_rc;
  routine_identifier: uds_svc_param_rc_rid;
  routine_control_option_record: PByte;
  routine_control_option_record_size: UInt32
): uds_status; overload;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcRoutineControl_2013")]
public static extern uds_status SvcRoutineControl_2013(
  [MarshalAs(UnmanagedType.U4)]
  cantp_handle channel,
  uds_msgconfig request_config,
  out uds_msg out_msg_request,
  [MarshalAs(UnmanagedType.U1)]
  uds_svc_param_rc routine_control_type,
  [MarshalAs(UnmanagedType.U2)]
  uds_svc_param_rc_rid routine_identifier,
  [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 6)]
  byte[] routine_control_option_record,
  UInt32 routine_control_option_record_size);

```

C++ / CLR

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcRoutineControl_2013")]
static uds_status SvcRoutineControl_2013(
  [MarshalAs(UnmanagedType::U4)]
  cantp_handle channel,
  uds_msgconfig request_config,
  uds_msg %out_msg_request,
  [MarshalAs(UnmanagedType::U1)]
  uds_svc_param_rc routine_control_type,
  [MarshalAs(UnmanagedType::U2)]

```

```
uds_svc_param_rc_rid routine_identifier,
[MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 6)]
array<Byte> ^routine_control_option_record,
UInt32 routine_control_option_record_size);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcRoutineControl_2013")>
Public Shared Function SvcRoutineControl_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.U1)>
    ByVal routine_control_type As uds_svc_param_rc,
    <MarshalAs(UnmanagedType.U2)>
    ByVal routine_identifier As uds_svc_param_rc_rid,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=6)>
    ByVal routine_control_option_record As Byte(),
    ByVal routine_control_option_record_size As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|------------------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| routine_control_type | Subfunction parameter: routine control type (see uds_svc_param_rc on page 93). |
| routine_identifier | Server local routine identifier (see uds_svc_param_rc_rid on page 94). |
| routine_control_option_record | Buffer containing the routine control options (only with start and stop routine subfunctions). |
| routine_control_option_record_size | Size in bytes of the buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is not valid. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | The mapping given in the message configuration is unknown. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with Reset_2013 (see Reset_2013 on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_OVERFLOW | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data length. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see MsgFree_2013 on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcRoutineControl_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
Byte[] routine_control_option_record = new Byte[10];
UInt16 routine_control_option_record_size = 10;
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Fill data
for (int i = 0; i < routine_control_option_record_size; i++)
{
    routine_control_option_record[i] = (Byte)(Convert.ToByte('A') + i);
}

// Sends a physical RoutineControl message
result = UDSApi.SvcRoutineControl_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out request,
    UDSApi.uds_svc_param_rc.PUDS_SVC_PARAM_RC_RRR, (UDSApi.uds_svc_param_rc_rid)0xF1A2,
    routine_control_option_record, routine_control_option_record_size);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
array<Byte>^ routine_control_option_record = gcnew array<Byte>(10);
UInt16 routine_control_option_record_size = 10;
uds_msgconfig config = {};
```



```
// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Fill data
for (int i = 0; i < routine_control_option_record_size; i++)
{
    routine_control_option_record[i] = 'A' + i;
}

// Sends a physical RoutineControl message
result = UDSApi::SvcRoutineControl_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_rc::PUDS_SVC_PARAM_RC_RRR, (UDSApi::uds_svc_param_rc_rid)0xF1A2,
    routine_control_option_record, routine_control_option_record_size);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);
```

Visual Basic

```
Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim routine_control_option_record(10) As Byte
Dim routine_control_option_record_size As UInt16 = 10
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Fill data
Dim start_value As Char = "A"
For i As UInt32 = 0 To routine_control_option_record_size - 1
    routine_control_option_record(i) = Convert.ToByte(start_value) + i
Next

' Sends a physical RoutineControl message
result = UDSApi.SvcRoutineControl_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi.uds_svc_param_rc.PUDS_SVC_PARAM_RC_RRR, &HF1A2, routine_control_option_record,
```

```

    routine_control_option_record_size)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApI.MsgFree_2013(request)
UDSApI.MsgFree_2013(response)
UDSApI.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    routine_control_option_record: array [0 .. 9] of Byte;
    routine_control_option_record_size: UInt16;
    i: UInt32;
begin
    routine_control_option_record_size := 10;
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

    // Fill data
    for i := 0 to routine_control_option_record_size - 1 do
    begin
        routine_control_option_record[i] := ($41 + i);
    end;

    // Sends a physical RoutineControl message
    result := TUDSApI.SvcRoutineControl_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        config, request, uds_svc_param_rc.PUDS_SVC_PARAM_RC_RRR,
        uds_svc_param_rc_rid($F1A2), @routine_control_option_record,
        routine_control_option_record_size);
    if TUDSApI.StatusIsOk_2013(result) then

```

```

begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: `WaitForService_2013` on page 248, `uds_svc_param_rc` on page 93, `uds_svc_param_rc_rid` on page 94.

Plain function version: `UDS_SvcRoutineControl_2013` on page 733.

3.7.100 SvcRequestDownload_2013

Writes a UDS request according to the RequestDownload service's specifications. The RequestDownload service is used by the client to initiate a data transfer from the client to the server/ECU (download).

Syntax

Pascal OO

```

class function SvcRequestDownload_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    compression_method: Byte;
    encrypting_method: Byte;
    memory_address: PByte;
    memory_address_size: Byte;
    memory_size: PByte;
    memory_size_size: Byte
): uds_status;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcRequestDownload_2013")]
public static extern uds_status SvcRequestDownload_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    Byte compression_method,
    Byte encrypting_method,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 6)]
    byte[] memory_address,
    byte memory_address_size,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 8)]
    byte[] memory_size,
    byte memory_size_size);

```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcRequestDownload_2013")]
static uds_status SvcRequestDownload_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    Byte compression_method,
    Byte encrypting_method,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 6)]
    array<Byte> ^memory_address,
    Byte memory_address_size,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 8)]
    array<Byte> ^memory_size,
    Byte memory_size_size);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcRequestDownload_2013")>
Public Shared Function SvcRequestDownload_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal compression_method As Byte,
    ByVal encrypting_method As Byte,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=6)>
    ByVal memory_address As Byte(),
    ByVal memory_address_size As Byte,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=8)>
    ByVal memory_size As Byte(),
    ByVal memory_size_size As Byte) As uds_status
End Function
```

Parameters

| Parameter | Description |
|---------------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| compression_method | A nibble-value that specifies the compression method, the value 0x0 specifies that no compression method is used. |
| encrypting_method | A nibble-value that specifies the encrypting method, the value 0x0 specifies that no encrypting method is used. |
| memory_address | Starting address of server (ECU) memory to which data is to be written. |
| memory_address_size | Size in bytes of the memory address buffer (max.: 0xF). |
| memory_size | Used by the server (ECU) to compare the uncompressed memory size with the total amount of data transferred during the TransferData service. |
| memory_size_size | Size in bytes of the memory size buffer (max.: 0xF). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcRequestDownload_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
Byte[] memory_address_buffer = new Byte[8];
Byte[] memory_size_buffer = new Byte[8];
Byte memory_address_size = 8;
Byte memory_size_size = 8;
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Fill data
for (int i = 0; i < memory_address_size; i++)
{
    memory_address_buffer[i] = (Byte)(Convert.ToByte('A') + i);
    memory_size_buffer[i] = (Byte)(10 + i);
}

// Sends a physical RequestDownload message

```

```

result = UDSApi.SvcRequestDownload_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, 0x01, 0x02, memory_address_buffer, memory_address_size, memory_size_buffer,
memory_size_size);
if (UDSApI.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApI.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApI.MsgFree_2013(ref request);
UDSApI.MsgFree_2013(ref response);
UDSApI.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
array<Byte>^ memory_address_buffer = gcnew array<Byte>(8);
array<Byte>^ memory_size_buffer = gcnew array<Byte>(8);
Byte memory_address_size = 8;
Byte memory_size_size = 8;
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Fill data
for (int i = 0; i < memory_address_size; i++)
{
    memory_address_buffer[i] = 'A' + i;
    memory_size_buffer[i] = (10 + i);
}

// Sends a physical RequestDownload message
result = UDSApi::SvcRequestDownload_2013(PCANTP_HANDLE_USBBUS1, config, request, 0x01, 0x02,
    memory_address_buffer, memory_address_size, memory_size_buffer, memory_size_size);
if (UDSApI::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApI::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApI::MsgFree_2013(request);
UDSApI::MsgFree_2013(response);
UDSApI::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim memory_address_buffer(8) As Byte
Dim memory_size_buffer(8) As Byte
Dim memory_address_size As Byte = 8
Dim memory_size_size As Byte = 8
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Fill data
Dim start_value As Char = "A"
For i As UInt32 = 0 To memory_address_size - 1
    memory_address_buffer(i) = Convert.ToByte(start_value) + i
    memory_size_buffer(i) = 10 + i
Next

' Sends a physical RequestDownload message
result = UDSApi.SvcRequestDownload_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    &H1, &H2, memory_address_buffer, memory_address_size, memory_size_buffer, memory_size_size)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    memory_address_buffer: array [0 .. 7] of Byte;
    memory_size_buffer: array [0 .. 7] of Byte;
    memory_address_size: Byte;
    memory_size_size: Byte;
    i: UInt32;
begin

```

```

memory_address_size := 8;
memory_size_size := 8;
FillChar(request, sizeof(request), 0);
FillChar(request_confirmation, sizeof(request_confirmation), 0);
FillChar(response, sizeof(response), 0);

// Set request message configuration
FillChar(config, sizeof(config), 0);
config.can_id :=
  UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr := $0;
config.nai.protocol :=
  uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr :=
  UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
config.nai.target_addr :=
  UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config.nai.target_type :=
  cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Fill data
for i := 0 to memory_address_size - 1 do
begin
  memory_address_buffer[i] := ($41 + i);
  memory_size_buffer[i] := (10 + i);
end;

// Sends a physical RequestDownload message
result := TUDSApi.SvcRequestDownload_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
  config, request, $01, $02, @memory_address_buffer, memory_address_size,
  @memory_size_buffer, memory_size_size);
if TUDSApi.StatusIsOk_2013(result) then
begin
  result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
  MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
  // An error occurred
  MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcRequestDownload_2013](#) on page 735.

3.7.101 SvcRequestUpload_2013

Writes a UDS request according to the RequestUpload service's specifications. The RequestUpload service is used by the client to initiate a data transfer from the server/ECU to the client (upload).

Syntax

Pascal OO

```
class function SvcRequestUpload_2013(
  channel: cantp_handle;
  request_config: uds_msgconfig;
  var out_msg_request: uds_msg;
  compression_method: Byte;
  encrypting_method: Byte;
  memory_address: PByte;
  memory_address_size: Byte;
  memory_size: PByte;
  memory_size_size: Byte
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcRequestUpload_2013")]
public static extern uds_status SvcRequestUpload_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    byte compression_method,
    byte encrypting_method,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 6)]
    byte[] memory_address,
    byte memory_address_size,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 8)]
    byte[] memory_size,
    byte memory_size_size);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcRequestUpload_2013")]
static uds_status SvcRequestUpload_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    Byte compression_method,
    Byte encrypting_method,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 6)]
    array<Byte> ^memory_address,
    Byte memory_address_size,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 8)]
    array<Byte> ^memory_size,
    Byte memory_size_size);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcRequestUpload_2013")>
Public Shared Function SvcRequestUpload_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
```

```

ByRef out_msg_request As uds_msg,
ByVal compression_method As Byte,
ByVal encrypting_method As Byte,
<MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=6)>
ByVal memory_address As Byte(),
ByVal memory_address_size As Byte,
<MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=8)>
ByVal memory_size As Byte(),
ByVal memory_size_size As Byte) As uds_status
End Function

```

Parameters

| Parameter | Description |
|---------------------|---|
| channel | The handle of a PUDS channel (see <code>can_tp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |
| compression_method | A nibble-value that specifies the compression method, the value 0x0 specifies that no compression method is used. |
| encrypting_method | A nibble-value that specifies the encrypting method, the value 0x0 specifies that no encrypting method is used. |
| memory_address | Starting address of server (ECU) memory from which data is to be retrieved. |
| memory_address_size | Size in bytes of the memory address buffer (max.: 0xF). |
| memory_size | Used by the server (ECU) to compare the uncompressed memory size with the total amount of data transferred during the TransferData service. |
| memory_size_size | Size in bytes of the memory size buffer (max.: 0xF). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcRequestUpload_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
Byte[] memory_address_buffer = new Byte[4];
Byte[] memory_size_buffer = new Byte[4];
Byte memory_address_size = 4;
Byte memory_size_size = 4;
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical RequestUpload message
result = UDSApi.SvcRequestUpload_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out request,
    0x01, 0x02, memory_address_buffer, memory_address_size, memory_size_buffer,
    memory_size_size);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
array<Byte>^ memory_address_buffer = gcnew array<Byte>(4);
array<Byte>^ memory_size_buffer = gcnew array<Byte>(4);
Byte memory_address_size = 4;
Byte memory_size_size = 4;
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

```

```
// Sends a physical RequestUpload message
result = UDSApi::SvcRequestUpload_2013(PCANTP_HANDLE_USBBUS1, config, request, 0x01, 0x02,
    memory_address_buffer, memory_address_size, memory_size_buffer, memory_size_size);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);
```

Visual Basic

```
Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim memory_address_buffer(4) As Byte
Dim memory_size_buffer(4) As Byte
Dim memory_address_size As Byte = 4
Dim memory_size_size As Byte = 4
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical RequestUpload message
result = UDSApi.SvcRequestUpload_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request, &H1,
    &H2, memory_address_buffer, memory_address_size, memory_size_buffer, memory_size_size)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)
```

Pascal OO

```

var
  result: uds_status;
  request: uds_msg;
  request_confirmation: uds_msg;
  response: uds_msg;
  config: uds_msgconfig;
  memory_address_buffer: array [0 .. 3] of Byte;
  memory_size_buffer: array [0 .. 3] of Byte;
  memory_address_size: Byte;
  memory_size_size: Byte;
begin
  memory_address_size := 4;
  memory_size_size := 4;
  FillChar(request, sizeof(request), 0);
  FillChar(request_confirmation, sizeof(request_confirmation), 0);
  FillChar(response, sizeof(response), 0);

  // Set request message configuration
  FillChar(config, sizeof(config), 0);
  config.can_id :=
    UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
  config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
  config.nai.extension_addr := $0;
  config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
  config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
  config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
  config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
  config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

  // Sends a physical RequestUpload message
  result := TUDSApi.SvcRequestUpload_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, request, $01, $02, @memory_address_buffer, memory_address_size,
    @memory_size_buffer, memory_size_size);
  if TUDSApi.StatusIsOk_2013(result) then
  begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
      @request, response, @request_confirmation);
  end;
  if TUDSApi.StatusIsOk_2013(result) then
  begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
  end
  else
  begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
  end;

  // Free structures
  TUDSApi.MsgFree_2013(request);
  TUDSApi.MsgFree_2013(response);
  TUDSApi.MsgFree_2013(request_confirmation);
end;

```



See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcRequestUpload_2013](#) on page 737.

3.7.102 SvcTransferData_2013

Writes a UDS request according to the TransferData service's specifications. The TransferData service is used by the client to transfer data either from the client to the server/ECU (download) or from the server/ECU to the client (upload).

Overloads

| | Method | Description |
|---|--|--|
|  | SvcTransferData_2013(cantp_handle, uds_msgconfig, uds_msg, byte) | Writes to the transmit queue a request for UDS service TransferData, without transfer request parameter. |
|  | SvcTransferData_2013(cantp_handle, uds_msgconfig, uds_msg, byte, byte[], UInt32) | Writes to the transmit queue a request for UDS service TransferData, with transfer request parameter. |

Plain function version: `UDS_SvcTransferData_2013` on page 739.

3.7.103 SvcTransferData_2013(cantp_handle, uds_msgconfig, uds_msg, byte)

Writes a UDS request according to the TransferData service's specifications (without transfer request parameter). The TransferData service is used by the client to transfer data either from the client to the server/ECU (download) or from the server/ECU to the client (upload).

Syntax

Pascal OO

```
class function SvcTransferData_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    block_sequence_counter: Byte
): uds_status; overload;
```

C#

```
public static uds_status SvcTransferData_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    byte block_sequence_counter);
```

C++ / CLR

```
static uds_status SvcTransferData_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    Byte block_sequence_counter);
```

Visual Basic

```
Public Shared Function SvcTransferData_2013(
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal block_sequence_counter As Byte) As uds_status
End Function
```

Parameters

| Parameter | Description |
|------------------------|--|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |
| block_sequence_counter | The <code>block_sequence_counter</code> parameter value starts at 0x01 with the first TransferData request that follows the RequestDownload (0x34) or RequestUpload (0x35) service. Its value is incremented by 1 for each subsequent TransferData request. At the value of 0xFF, the <code>block_sequence_counter</code> rolls over and starts at 0x0 with the next TransferData request message. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcTransferData_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;
```



```
// Sends a physical TransferData message
result = UDSApi.SvcTransferData_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out request,
    0x01);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical TransferData message
result = UDSApi::SvcTransferData_2013(PCANTP_HANDLE_USBBUS1, config, request, 0x01);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);
```

Visual Basic

```
Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
```



```

config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical TransferData message
result = UDSApi.SvcTransferData_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request, &H1)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.type := uds_msgtype.PUDS_MSGTYPE_USDT;

    // Sends a physical TransferData message
    result := TUDSApi.SvcTransferData_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        config, request, $01);
    if TUDSApi.StatusIsOk_2013(result) then
    begin
        result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
            @request, response, @request_confirmation);
    end;
end;

```

```

if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcTransferData_2013](#) on page 739.

3.7.104 SvcTransferData_2013(cantp_handle, uds_msgconfig, uds_msg, byte, byte[], UInt32)

Writes a UDS request according to the TransferData service's specifications. The TransferData service is used by the client to transfer data either from the client to the server/ECU (download) or from the server/ECU to the client (upload).

Syntax

Pascal OO

```

class function SvcTransferData_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    block_sequence_counter: Byte;
    transfer_request_parameter_record: PByte;
    transfer_request_parameter_record_size: UInt32
): uds_status; overload;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcTransferData_2013")]
public static extern uds_status SvcTransferData_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    byte block_sequence_counter,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 5)]
    byte[] transfer_request_parameter_record,
    UInt32 transfer_request_parameter_record_size);

```

C++ / CLR

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcTransferData_2013")]
static uds_status SvcTransferData_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    Byte block_sequence_counter,

```

```
[MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 5)]
array<Byte> ^transfer_request_parameter_record,
UInt32 transfer_request_parameter_record_size);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcTransferData_2013")>
Public Shared Function SvcTransferData_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal block_sequence_counter As Byte,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=5)>
    ByVal transfer_request_parameter_record As Byte(),
    ByVal transfer_request_parameter_record_size As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|--|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| block_sequence_counter | The block_sequence_counter parameter value starts at 0x01 with the first TransferData request that follows the RequestDownload (0x34) or RequestUpload (0x35) service. Its value is incremented by 1 for each subsequent TransferData request. At the value of 0xFF, the block_sequence_counter rolls over and starts at 0x0 with the next TransferData request message. |
| transfer_request_parameter_record | Buffer containing the required transfer parameters. |
| transfer_request_parameter_record_size | Size in bytes of the buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is not valid. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | The mapping given in the message configuration is unknown. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with Reset_2013 (see Reset_2013 on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_OVERFLOW | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data length. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see MsgFree_2013 on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcTransferData_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
Byte[] record = new Byte[50];
Byte record_size = 50;
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Fill data
for (int i = 0; i < record_size; i++)
{
    record[i] = (Byte)(Convert.ToByte('A') + i);
}

// Sends a physical TransferData message
result = UDSApi.SvcTransferData_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out request,
    0x01, record, record_size);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
array<Byte>^ record = gcnew array<Byte>(50);
Byte record_size = 50;
uds_msgconfig config = {};

// Set request message configuration

```

```

config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Fill data
for (int i = 0; i < record_size; i++)
{
    record[i] = 'A' + i;
}

// Sends a physical TransferData message
result = UDSApi::SvcTransferData_2013(PCANTP_HANDLE_USBBUS1, config, request, 0x01, record,
    record_size);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim record(50) As Byte
Dim record_size As Byte = 50
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Fill data
Dim start_value As Char = "A"
For i As UInt32 = 0 To record_size - 1
    record(i) = Convert.ToByte(start_value) + i
Next

' Sends a physical TransferData message
result = UDSApi.SvcTransferData_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request, &H1,
    record, record_size)
If UDSApi.StatusIsOk_2013(result) Then

```

```

    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    param_record: array [0 .. 49] of Byte;
    record_size: Byte;
    i: UInt32;
begin
    record_size := 50;
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

    // Fill data
    for i := 0 to record_size - 1 do
    begin
        param_record[i] := ($41 + i);
    end;

    // Sends a physical TransferData message
    result := TUDSApi.SvcTransferData_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        config, request, $01, @param_record, record_size);
    if TUDSApi.StatusIsOk_2013(result) then
    begin
        result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
            @request, response, @request_confirmation);
    end;
end;

```

```

if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```



See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcTransferData_2013](#) on page 739.

3.7.105 SvcRequestTransferExit_2013

Writes a UDS request according to the RequestTransferExit service's specifications. The RequestTransferExit service is used by the client to terminate a data transfer between the client and server/ECU (upload or download).

Overloads

| | Method | Description |
|---|--|---|
|  | <code>SvcRequestTransferExit_2013(cantp_handle, uds_msgconfig, uds_msg)</code> | Writes to the transmit queue a request for UDS service RequestTransferExit, without transfer request parameter. |
|  | <code>SvcRequestTransferExit_2013(cantp_handle, uds_msgconfig, uds_msg, byte[], UInt32)</code> | Writes to the transmit queue a request for UDS service RequestTransferExit, with transfer request parameter. |

Plain function version: [UDS_SvcRequestTransferExit_2013](#) on page 742.

3.7.106 SvcRequestTransferExit_2013(cantp_handle, uds_msgconfig, uds_msg)

Writes a UDS request according to the RequestTransferExit service's specifications, without transfer request parameter. The RequestTransferExit service is used by the client to terminate a data transfer between the client and server/ECU (upload or download).

Syntax

Pascal OO

```

class function SvcRequestTransferExit_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg
): uds_status; overload;

```

C#

```

public static uds_status SvcRequestTransferExit_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request);

```

C++ / CLR

```
static uds_status SvcRequestTransferExit_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request);
```

Visual Basic

```
Public Shared Function SvcRequestTransferExit_2013(
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcRequestTransferExit_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();
```



```
// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical RequestTransferExit message
result = UDSApi.SvcRequestTransferExit_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical RequestTransferExit message
result = UDSApi::SvcRequestTransferExit_2013(PCANTP_HANDLE_USBBUS1, config, request);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);
```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical RequestTransferExit message
result = UDSApi.SvcRequestTransferExit_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=

```

```

cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical RequestTransferExit message
result := TUDSApi.SvcRequestTransferExit_2013
(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request);
if TUDSApi.StatusIsOk_2013(result) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: `WaitForService_2013` on page 248.

Plain function version: `UDS_SvcRequestTransferExit_2013` on page 742.

3.7.107 SvcRequestTransferExit_2013(cantp_handle, uds_msgconfig, uds_msg, byte[], UInt32)

Writes a UDS request according to the RequestTransferExit service's specifications. The RequestTransferExit service is used by the client to terminate a data transfer between the client and server/ECU (upload or download).

Syntax

Pascal OO

```

class function SvcRequestTransferExit_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    transfer_request_parameter_record: PByte;
    transfer_request_parameter_record_size: UInt32
): uds_status;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcRequestTransferExit_2013")]
public static extern uds_status SvcRequestTransferExit_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 4)]
    byte[] transfer_request_parameter_record,
    UInt32 transfer_request_parameter_record_size);

```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcRequestTransferExit_2013")]
static uds_status SvcRequestTransferExit_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 4)]
    array<Byte> ^transfer_request_parameter_record,
    UInt32 transfer_request_parameter_record_size);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcRequestTransferExit_2013")>
Public Shared Function SvcRequestTransferExit_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=4)>
    ByVal transfer_request_parameter_record As Byte(),
    ByVal transfer_request_parameter_record_size As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|--|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| transfer_request_parameter_record | Buffer containing the required transfer parameters. |
| transfer_request_parameter_record_size | Size in bytes of the buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is not valid. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | The mapping given in the message configuration is unknown. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with Reset_2013 (see Reset_2013 on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_OVERFLOW | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data length. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see MsgFree_2013 on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcRequestTransferExit_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
Byte[] transfer_request_parameter_record = new Byte[20];
Byte transfer_request_parameter_record_size = 20;
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Fill data
for (int i = 0; i < transfer_request_parameter_record_size; i++)
{
    transfer_request_parameter_record[i] = (Byte)(Convert.ToByte('A') + i);
}

// Sends a physical RequestTransferExit message
result = UDSApi.SvcRequestTransferExit_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, transfer_request_parameter_record, transfer_request_parameter_record_size);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
array<Byte>^ transfer_request_parameter_record = gcnew array<Byte>(20);
Byte transfer_request_parameter_record_size = 20;
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
```

```

config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Fill data
for (int i = 0; i < transfer_request_parameter_record_size; i++)
{
    transfer_request_parameter_record[i] = 'A' + i;
}

// Sends a physical RequestTransferExit message
result = UDSApi::SvcRequestTransferExit_2013(PCANTP_HANDLE_USBBUS1, config, request,
    transfer_request_parameter_record, transfer_request_parameter_record_size);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim transfer_request_parameter_record(20) As Byte
Dim transfer_request_parameter_record_size As Byte = 20
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Fill data
Dim start_value As Char = "A"
For i As UInt32 = 0 To transfer_request_parameter_record_size - 1
    transfer_request_parameter_record(i) = Convert.ToByte(start_value) + i
Next

' Sends a physical RequestTransferExit message
result = UDSApi.SvcRequestTransferExit_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, transfer_request_parameter_record, transfer_request_parameter_record_size)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,

```

```

        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    transfer_request_parameter_record: array [0 .. 19] of Byte;
    transfer_request_parameter_record_size: Byte;
    i: UInt32;
begin
    transfer_request_parameter_record_size := 20;
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

    // Fill data
    for i := 0 to transfer_request_parameter_record_size - 1 do
    begin
        transfer_request_parameter_record[i] := ($41 + i);
    end;

    // Sends a physical RequestTransferExit message
    result := TUDSApi.SvcRequestTransferExit_2013
        (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request, @transfer_request_parameter_record,
        transfer_request_parameter_record_size);
    if TUDSApi.StatusIsOk_2013(result) then
    begin
        result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
    end;
end;

```



```

if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcRequestTransferExit_2013](#) on page 742.

3.7.108 SvcAccessTimingParameter_2013

Writes a UDS request according to the AccessTimingParameter service's specifications.

Syntax

Pascal OO

```

class function SvcAccessTimingParameter_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    access_type: uds_svc_param_atp;
    request_record: PByte;
    record_size: UInt32
): uds_status;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcAccessTimingParameter_2013")]
public static extern uds_status SvcAccessTimingParameter_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    [MarshalAs(UnmanagedType.U1)]
    uds_svc_param_atp access_type,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 5)]
    byte[] request_record,
    UInt32 record_size);

```

C++ / CLR

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcAccessTimingParameter_2013")]
static uds_status SvcAccessTimingParameter_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType.U1)]
    uds_svc_param_atp access_type,

```



```
[MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 5)]
array<Byte> ^request_record,
UInt32 record_size);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcAccessTimingParameter_2013")>
Public Shared Function SvcAccessTimingParameter_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.U1)>
    ByVal access_type As uds_svc_param_atp,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=5)>
    ByVal request_record As Byte(),
    ByVal record_size As UInt32) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |
| access_type | Access type (see <code>uds_svc_param_atp</code> on page 95). |
| request_record | Timing parameter request record. |
| record_size | Size in bytes of the request record. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_OVERFLOW</code> | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data length. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcAccessTimingParameter_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical SvcAccessTimingParameter message
Byte[] request_record = { 0xAB, 0xCD };
UInt32 record_size = 2;
result = UDSApi.SvcAccessTimingParameter_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, UDSApi.uds_svc_param_atp.PUDS_SVC_PARAM_ATP_RCATP, request_record,
    record_size);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++ / CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical SvcAccessTimingParameter message
array<Byte>^ request_record = { 0xAB, 0xCD };
UInt32 record_size = 2;
result = UDSApi::SvcAccessTimingParameter_2013(PCANTP_HANDLE_USBBUS1, config, request,
```

```

        UDSApi::uds_svc_param_atp::PUDS_SVC_PARAM_ATP_RCATP, request_record, record_size);
if (UDSApI::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApI::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApI::MsgFree_2013(request);
UDSApI::MsgFree_2013(response);
UDSApI::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical SvcAccessTimingParameter message
Dim request_record As Byte() = {&HAB, &HCD}
Dim record_size As UInt32 = 2
result = UDSApi.SvcAccessTimingParameter_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, UDSApi.uds_svc_param_atp.PUDS_SVC_PARAM_ATP_RCATP, request_record,
    record_size)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApI.MsgFree_2013(request)
UDSApI.MsgFree_2013(response)
UDSApI.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request_record: array [0 .. 1] of Byte;
    request: uds_msg;
    request_confirmation: uds_msg;

```

```

response: uds_msg;
config: uds_msgconfig;
record_size: UInt32;
begin
  FillChar(request, sizeof(request), 0);
  FillChar(request_confirmation, sizeof(request_confirmation), 0);
  FillChar(response, sizeof(response), 0);

  // Set request message configuration
  FillChar(config, sizeof(config), 0);
  config.can_id :=
    UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
  config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
  config.nai.extension_addr := $0;
  config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
  config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
  config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
  config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
  config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

  // Sends a physical SvcAccessTimingParameter message
  request_record[0] := $AB;
  request_record[1] := $CD;
  record_size := 2;
  result := TUDSApi.SvcAccessTimingParameter_2013
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
     uds_svc_param_atp.PUDS_SVC_PARAM_ATP_RCATP, @request_record,
     record_size);
  if TUDSApi.StatusIsOk_2013(result) then
  begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
      @request, response, @request_confirmation);
  end;
  if TUDSApi.StatusIsOk_2013(result) then
  begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
  end
  else
  begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
  end;

  // Free structures
  TUDSApi.MsgFree_2013(request);
  TUDSApi.MsgFree_2013(response);
  TUDSApi.MsgFree_2013(request_confirmation);
end;

```







See also: `WaitForService_2013` on page 248, `uds_svc_param_atp` on page 95.

Plain function version: `UDS_SvcAccessTimingParameter_2013` on page 744.

3.7.109 SvcRequestFileTransfer_2013

Writes a UDS request according to the SvcRequestFileTransfer service's specifications.

Overloads

| | Method | Description |
|--|---|---|
|  | SvcRequestFileTransfer_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_rft_moop, UInt16, string) | Writes to the transmit queue a request for UDS service RequestFileTransfer. Use with a mode of operation which does not require file compressed or uncompressed sizes, nor compression or encrypting methods. |
|  | SvcRequestFileTransfer_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_rft_moop, UInt16, string, byte, byte) | Writes to the transmit queue a request for UDS service RequestFileTransfer. Use with a mode of operation which does not require file compressed or uncompressed sizes. |
|  | SvcRequestFileTransfer_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_rft_moop, UInt16, string, byte, byte, byte, byte[], byte[]) | Writes to the transmit queue a request for UDS service RequestFileTransfer. |
|  | SvcRequestFileTransfer_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_rft_moop, UInt16, byte[]) | Writes to the transmit queue a request for UDS service RequestFileTransfer. Use with a mode of operation which does not require file compressed or uncompressed sizes, nor compression or encrypting methods. |
|  | SvcRequestFileTransfer_2013(cantp_handle, uds_msgconfig, out uds_msg, uds_svc_param_rft_moop, UInt16, byte[], byte, byte) | Writes to the transmit queue a request for UDS service RequestFileTransfer. Use with a mode of operation which does not require file compressed or uncompressed sizes. |
|  | SvcRequestFileTransfer_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_rft_moop, UInt16, byte[], byte, byte, byte, byte[], byte[]) | Writes to the transmit queue a request for UDS service RequestFileTransfer. |

Plain function version: [UDS_SvcRequestFileTransfer_2013](#) on page 745.

3.7.110 SvcRequestFileTransfer_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_rft_moop, UInt16, string)

Writes to the transmit queue a request for UDS service RequestFileTransfer. Use with a mode of operation which does not require file compressed or uncompressed sizes, nor compression or encrypting methods.

Syntax

Pascal OO

```
class function SvcRequestFileTransfer_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    mode_of_operation: uds_svc_param_rft_moop;
    file_path_and_name_size: UInt16;
    file_path_and_name: PAnsiChar
): uds_status; overload;
```

C#

```
public static uds_status SvcRequestFileTransfer_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    uds_svc_param_rft_moop mode_of_operation,
    UInt16 file_path_and_name_size,
    string file_path_and_name);
```

C++ / CLR

```
static uds_status SvcRequestFileTransfer_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    uds_svc_param_rft_moop mode_of_operation,
    UInt16 file_path_and_name_size,
    String ^file_path_and_name);
```

Visual Basic

```
Public Shared Function SvcRequestFileTransfer_2013(
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal mode_of_operation As uds_svc_param_rft_moop,
    ByVal file_path_and_name_size As UInt16,
    ByVal file_path_and_name As String) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-------------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |
| mode_of_operation | Mode of operation (see <code>uds_svc_param_rft_moop</code> on page 96). |
| file_path_and_name_size | File path and name string length. |
| file_path_and_name | File path and name string. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcRequestFileTransfer_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical RequestFileTransfer message
String file_name = "toto.txt";
result = UDSApi.SvcRequestFileTransfer_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, UDSApi.uds_svc_param_rft_moop.PUDS_SVC_PARAM_RFT_MOOP_DELFIL, 8, file_name);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++ / CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical RequestFileTransfer message
String ^file_name = "toto.txt";
result = UDSApi::SvcRequestFileTransfer_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_rft_moop::PUDS_SVC_PARAM_RFT_MOOP_DELFIL, 8, file_name);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
```



```

        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical RequestFileTransfer message
Dim file_name As String = "toto.txt"
result = UDSApi.SvcRequestFileTransfer_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, UDSApi.uds_svc_param_rft_moop.PUDS_SVC_PARAM_RFT_MOOP_DELFIE, 8, file_name)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    file_name: PAnsichar;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);

```



```

FillChar(response, sizeof(response), 0);

// Set request message configuration
FillChar(config, sizeof(config), 0);
config.can_id :=
    UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr := $0;
config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical RequestFileTransfer message
file_name := 'toto.txt';
result := TUDSApi.SvcRequestFileTransfer_2013
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    uds_svc_param_rft_moop.PUDS_SVC_PARAM_RFT_MOOP_DELFIE, 8, file_name);
if TUDSApi.StatusIsOk_2013(result) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248, [uds_svc_param_rft_moop](#) on page 96,

Plain function version: [UDS_SvcRequestFileTransfer_2013](#) on page 745.

3.7.111 SvcRequestFileTransfer_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_rft_moop, UInt16, string, byte, byte)

Writes to the transmit queue a request for UDS service RequestFileTransfer. Use with a mode of operation which does not require file compressed or uncompressed sizes.

Syntax

Pascal OO

```

class function SvcRequestFileTransfer_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;

```

```

mode_of_operation: uds_svc_param_rft_moop;
file_path_and_name_size: UInt16;
file_path_and_name: PAnsiChar;
compression_method: Byte;
encrypting_method: Byte
): uds_status; overload;

```

C#

```

public static uds_status SvcRequestFileTransfer_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    uds_svc_param_rft_moop mode_of_operation,
    UInt16 file_path_and_name_size,
    string file_path_and_name,
    byte compression_method,
    byte encrypting_method);

```

C++ / CLR

```

static uds_status SvcRequestFileTransfer_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    uds_svc_param_rft_moop mode_of_operation,
    UInt16 file_path_and_name_size,
    String ^file_path_and_name,
    Byte compression_method,
    Byte encrypting_method);

```

Visual Basic

```

Public Shared Function SvcRequestFileTransfer_2013(
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal mode_of_operation As uds_svc_param_rft_moop,
    ByVal file_path_and_name_size As UInt16,
    ByVal file_path_and_name As String,
    ByVal compression_method As Byte,
    ByVal encrypting_method As Byte) As uds_status
End Function

```

Parameters

| Parameter | Description |
|-------------------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| mode_of_operation | Mode of operation (see uds_svc_param_rft_moop on page 96). |
| file_path_and_name_size | File path and name string length. |
| file_path_and_name | File path and name string. |
| compression_method | A nibble-value that specifies the compression method, the value 0x0 specifies that no compression method is used. |
| encrypting_method | A nibble-value that specifies the encrypting method, the value 0x0 specifies that no encrypting method is used. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcRequestFileTransfer_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical RequestFileTransfer message
String file_name = "toto.txt";
result = UDSApi.SvcRequestFileTransfer_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, UDSApi.uds_svc_param_rft_moop.PUDS_SVC_PARAM_RFT_MOOP_RDFILE, 8, file_name, 0x01,
    0x02);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred

```

```
MessageBox.Show("An error occurred", "Error");
```

```
// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++ / CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical RequestFileTransfer message
String ^file_name = "toto.txt";
result = UDSApi::SvcRequestFileTransfer_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_rft_moop::PUDS_SVC_PARAM_RFT_MOOP_RDFILE, 8, file_name, 0x01,
    0x02);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);
```

Visual Basic

```
Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT
```

```

' Sends a physical RequestFileTransfer message
Dim file_name As String = "toto.txt"
result = UDSApi.SvcRequestFileTransfer_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, UDSApi.uds_svc_param_rft_moop.PUDS_SVC_PARAM_RFT_MOOP_RDFILE, 8, file_name, &H1,
    &H2)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    file_name: PAnsichar;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

    // Sends a physical RequestFileTransfer message
    file_name := 'toto.txt';
    result := TUDSApi.SvcRequestFileTransfer_2013
        (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
        uds_svc_param_rft_moop.PUDS_SVC_PARAM_RFT_MOOP_RDFILE, 8, file_name,
        $01, $02);
    if TUDSApi.StatusIsOk_2013(result) then
    begin
        result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
            @request, response, @request_confirmation);
    end;
end;

```

```

if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248, [uds_svc_param_rft_moop](#) on page 96,

Plain function version: [UDS_SvcRequestFileTransfer_2013](#) on page 745.

3.7.112 SvcRequestFileTransfer_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_rft_moop, UInt16, string, byte, byte, byte, byte[], byte[])

Writes to the transmit queue a request for UDS service RequestFileTransfer.

Syntax

Pascal OO

```

class function SvcRequestFileTransfer_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    mode_of_operation: uds_svc_param_rft_moop;
    file_path_and_name_size: UInt16;
    file_path_and_name: PAnsiChar;
    compression_method: Byte;
    encrypting_method: Byte;
    file_size_parameter_size: Byte;
    file_size_uncompressed: PByte;
    file_size_compressed: PByte
): uds_status; overload;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcRequestFileTransfer_2013")]
public static extern uds_status SvcRequestFileTransfer_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    [MarshalAs(UnmanagedType.U1)]
    uds_svc_param_rft_moop mode_of_operation,
    UInt16 file_path_and_name_size,
    [MarshalAs(UnmanagedType.LPStr, SizeParamIndex = 4)]
    string file_path_and_name,
    byte compression_method,
    byte encrypting_method,
    byte file_size_parameter_size,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 8)]

```

```
byte[] file_size_uncompressed,
[MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 8)]
byte[] file_size_compressed);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcRequestFileTransfer_2013")]
static uds_status SvcRequestFileTransfer_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType::U1)]
    uds_svc_param_rft_moop mode_of_operation,
    UInt16 file_path_and_name_size,
    [MarshalAs(UnmanagedType::LPStr, SizeParamIndex = 4)]
    String ^file_path_and_name,
    Byte compression_method,
    Byte encrypting_method,
    Byte file_size_parameter_size,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 8)]
    array<Byte> ^file_size_uncompressed,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 8)]
    array<Byte> ^file_size_compressed);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcRequestFileTransfer_2013")>
Public Shared Function SvcRequestFileTransfer_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.U1)>
    ByVal mode_of_operation As uds_svc_param_rft_moop,
    ByVal file_path_and_name_size As UInt16,
    <MarshalAs(UnmanagedType.LPStr, SizeParamIndex:=4)>
    ByVal file_path_and_name As String,
    ByVal compression_method As Byte,
    ByVal encrypting_method As Byte,
    ByVal file_size_parameter_size As Byte,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=8)>
    ByVal file_size_uncompressed As Byte(),
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=8)>
    ByVal file_size_compressed As Byte() As uds_status
End Function
```

Parameters

| Parameter | Description |
|--------------------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| mode_of_operation | Mode of operation (see uds_svc_param_rft_moop on page 96). |
| file_path_and_name_size | File path and name string length. |
| file_path_and_name | File path and name string. |
| compression_method | A nibble-value that specifies the compression method, the value 0x0 specifies that no compression method is used. |
| encrypting_method | A nibble-value that specifies the encrypting method, the value 0x0 specifies that no encrypting method is used. |
| file_size_parameter_size | File size parameter length in byte |
| file_size_uncompressed | Uncompressed file size. |

| Parameter | Description |
|----------------------|-----------------------|
| file_size_compressed | Compressed file size. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcRequestFileTransfer_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical RequestFileTransfer message
String file_name = "toto.txt";
Byte[] file_size_uncompressed = { 0x00, 0xD };
Byte[] file_size_compressed = { 0x0, 0xA };
result = UDSApi.SvcRequestFileTransfer_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, UDSApi.uds_svc_param_rft_moop.PUDS_SVC_PARAM_RFT_MOOP_ADDFILE, 8, file_name, 0, 0,
    2, file_size_uncompressed, file_size_compressed);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out

```



```

        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++ / CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical RequestFileTransfer message
String ^file_name = "toto.txt";
array<Byte>^ file_size_uncompressed = { 0x00, 0xD };
array<Byte>^ file_size_compressed = { 0x0, 0xA };
result = UDSApi::SvcRequestFileTransfer_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDASvcParamRftMoop::PUDS_SVC_PARAM_RFT_MOOP_ADDFILE, 8, file_name, 0, 0, 2,
    file_size_uncompressed, file_size_compressed);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD

```

```

config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical RequestFileTransfer message
Dim file_name As String = "toto.txt"
Dim file_size_uncompressed As Byte() = {&H0, &HD}
Dim file_size_compressed As Byte() = {&H0, &HA}
result = UDSApI.SvcRequestFileTransfer_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, UDSApI.uds_svc_param_rft_moop.PUDS_SVC_PARAM_RFT_MOOP_ADDFILE, 8, file_name, 0, 0,
    2, file_size_uncompressed, file_size_compressed)
If UDSApI.StatusIsOk_2013(result) Then
    result = UDSApI.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApI.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApI.MsgFree_2013(request)
UDSApI.MsgFree_2013(response)
UDSApI.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    file_size_uncompressed: array [0 .. 1] of Byte;
    file_size_compressed: array [0 .. 1] of Byte;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    file_name: PAnsichar;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.type := uds_msgtype.PUDS_MSGTYPE_USDT;

```

```
// Sends a physical RequestFileTransfer message
file_name := 'toto.txt';
file_size_uncompressed[0] := $00;
file_size_uncompressed[1] := $D;
file_size_compressed[0] := $0;
file_size_compressed[1] := $A;
result := TUDSApi.SvcRequestFileTransfer_2013
(
  cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
  uds_svc_param_rft_moop.PUDS_SVC_PARAM_RFT_MOOP_ADDFILE, 8, file_name, 0, 0,
  2, @file_size_uncompressed, @file_size_compressed);
if TUDSApi.StatusIsOk_2013(result) then
begin
  result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
  MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
  // An error occurred
  MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;
```

See also: [WaitForService_2013](#) on page 248, [uds_svc_param_rft_moop](#) on page 96,

Plain function version: [UDS_SvcRequestFileTransfer_2013](#) on page 745.

3.7.113 SvcRequestFileTransfer_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_rft_moop, UInt16, byte[])

Writes to the transmit queue a request for UDS service RequestFileTransfer. Use with a mode of operation which does not require file compressed or uncompressed sizes, nor compression or encrypting methods.

Syntax

Pascal OO

```
class function SvcRequestFileTransfer_2013(
  channel: cantp_handle;
  request_config: uds_msgconfig;
  var out_msg_request: uds_msg;
  mode_of_operation: uds_svc_param_rft_moop;
  file_path_and_name_size: UInt16;
  file_path_and_name: PByte
): uds_status; overload;
```

C#

```
public static uds_status SvcRequestFileTransfer_2013(
  cantp_handle channel,
  uds_msgconfig request_config,
  out uds_msg out_msg_request,
```

```
uds_svc_param_rft_moop mode_of_operation,
UInt16 file_path_and_name_size,
byte[] file_path_and_name);
```

C++ / CLR

```
static uds_status SvcRequestFileTransfer_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    uds_svc_param_rft_moop mode_of_operation,
    UInt16 file_path_and_name_size,
    array<Byte> ^file_path_and_name);
```

Visual Basic

```
Public Shared Function SvcRequestFileTransfer_2013(
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal mode_of_operation As uds_svc_param_rft_moop,
    ByVal file_path_and_name_size As UInt16,
    ByVal file_path_and_name As Byte()) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-------------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |
| mode_of_operation | Mode of operation (see <code>uds_svc_param_rft_moop</code> on page 96). |
| file_path_and_name_size | File path and name string length. |
| file_path_and_name | File path and name string. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcRequestFileTransfer_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical RequestFileTransfer message
Byte[] file_name = Encoding.ASCII.GetBytes("toto.txt");
result = UDSApi.SvcRequestFileTransfer_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, UDSApi.uds_svc_param_rft_moop.PUDS_SVC_PARAM_RFT_MOOP_DELFIE, 8, file_name);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++ / CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;
```

```
// Sends a physical RequestFileTransfer message
array<Byte>^ file_name = Encoding::ASCII->GetBytes("toto.txt");
result = UDSApi::SvcRequestFileTransfer_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_rft_moop::PUDS_SVC_PARAM_RFT_MOOP_DELFIL, 8, file_name);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);
```

Visual Basic

```
Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical RequestFileTransfer message
Dim file_name As Byte() = Encoding.ASCII.GetBytes("toto.txt")
result = UDSApi.SvcRequestFileTransfer_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, UDSApi.uds_svc_param_rft_moop.PUDS_SVC_PARAM_RFT_MOOP_DELFIL, 8, file_name)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)
```

Pascal OO

```
var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
```

```

response: uds_msg;
config: uds_msgconfig;
file_name: System.TArray<Byte>;
begin
  FillChar(request, sizeof(request), 0);
  FillChar(request_confirmation, sizeof(request_confirmation), 0);
  FillChar(response, sizeof(response), 0);

  // Set request message configuration
  FillChar(config, sizeof(config), 0);
  config.can_id :=
    UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
  config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
  config.nai.extension_addr := $0;
  config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
  config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
  config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
  config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
  config.typem := uds_msgtype.PUDS_MSGTYPE_USDT;

  // Sends a physical RequestFileTransfer message
  file_name := TEncoding.ASCII.GetBytes('toto.txt');

  result := TUDSApi.SvcRequestFileTransfer_2013
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
     uds_svc_param_rft_moop.PUDS_SVC_PARAM_RFT_MOOP_DELFIE, 8,
     PByte(file_name));
  if TUDSApi.StatusIsOk_2013(result) then
  begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
      @request, response, @request_confirmation);
  end;
  if TUDSApi.StatusIsOk_2013(result) then
  begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
  end
  else
  begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
  end;

  // Free structures
  TUDSApi.MsgFree_2013(request);
  TUDSApi.MsgFree_2013(response);
  TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248, [uds_svc_param_rft_moop](#) on page 96,
Plain function version: [UDS_SvcRequestFileTransfer_2013](#) on page 745.

3.7.114 SvcRequestFileTransfer_2013(cantp_handle, uds_msgconfig, out uds_msg, uds_svc_param_rft_moop, UInt16, byte[], byte, byte)

Writes to the transmit queue a request for UDS service RequestFileTransfer. Use with a mode of operation which does not require file compressed or uncompressed sizes.

Syntax

Pascal OO

```
class function SvcRequestFileTransfer_2013(
  channel: cantp_handle;
  request_config: uds_msgconfig;
  var out_msg_request: uds_msg;
  mode_of_operation: uds_svc_param_rft_moop;
  file_path_and_name_size: UInt16;
  file_path_and_name: PByte;
  compression_method: Byte;
  encrypting_method: Byte
): uds_status; overload;
```

C#

```
public static uds_status SvcRequestFileTransfer_2013(
  cantp_handle channel,
  uds_msgconfig request_config,
  out uds_msg out_msg_request,
  uds_svc_param_rft_moop mode_of_operation,
  UInt16 file_path_and_name_size,
  byte[] file_path_and_name,
  byte compression_method,
  byte encrypting_method);
```

C++ / CLR

```
static uds_status SvcRequestFileTransfer_2013(
  cantp_handle channel,
  uds_msgconfig request_config,
  uds_msg %out_msg_request,
  uds_svc_param_rft_moop mode_of_operation,
  UInt16 file_path_and_name_size,
  array<Byte> ^file_path_and_name,
  Byte compression_method,
  Byte encrypting_method);
```

Visual Basic

```
Public Shared Function SvcRequestFileTransfer_2013(
  ByVal channel As cantp_handle,
  ByVal request_config As uds_msgconfig,
  ByRef out_msg_request As uds_msg,
  ByVal mode_of_operation As uds_svc_param_rft_moop,
  ByVal file_path_and_name_size As UInt16,
  ByVal file_path_and_name As Byte(),
  ByVal compression_method As Byte,
  ByVal encrypting_method As Byte) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |

| Parameter | Description |
|-------------------------|---|
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |
| mode_of_operation | Mode of operation (see <code>uds_svc_param_rft_moop</code> on page 96). |
| file_path_and_name_size | File path and name string length. |
| file_path_and_name | File path and name string. |
| compression_method | A nibble-value that specifies the compression method, the value 0x0 specifies that no compression method is used. |
| encrypting_method | A nibble-value that specifies the encrypting method, the value 0x0 specifies that no encrypting method is used. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcRequestFileTransfer_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

```

```
// Sends a physical RequestFileTransfer message
Byte[] file_name = Encoding.ASCII.GetBytes("toto.txt");
result = UDSApi.SvcRequestFileTransfer_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, UDSApi.uds_svc_param_rft_moop.PUDS_SVC_PARAM_RFT_MOOP_RDFILE, 8, file_name, 0x01,
    0x02);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++ / CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical RequestFileTransfer message
array<Byte>^ file_name = Encoding::ASCII->GetBytes("toto.txt");
result = UDSApi::SvcRequestFileTransfer_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_rft_moop::PUDS_SVC_PARAM_RFT_MOOP_RDFILE, 8, file_name, 0x01,
    0x02);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);
```

Visual Basic

```
Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
```

```

Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical RequestFileTransfer message
Dim file_name As Byte() = Encoding.ASCII.GetBytes("toto.txt")
result = UDSApi.SvcRequestFileTransfer_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, UDSApi.uds_svc_param_rft_moop.PUDS_SVC_PARAM_RFT_MOOP_RDFILE, 8, file_name, &H1,
    &H2)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    file_name: System.TArray<Byte>;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.type := uds_msgtype.PUDS_MSGTYPE_USDT;

```

```
// Sends a physical RequestFileTransfer message
file_name := TEncoding.ASCII.GetBytes('toto.txt');
result := TUDSApi.SvcRequestFileTransfer_2013
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    uds_svc_param_rft_moop.PUDS_SVC_PARAM_RFT_MOOP_RDFILE, 8, PByte(file_name),
    $01, $02);
if TUDSApi.StatusIsOk_2013(result) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;
```

See also: [WaitForService_2013](#) on page 248, [uds_svc_param_rft_moop](#) on page 96,

Plain function version: [UDS_SvcRequestFileTransfer_2013](#) on page 745.

3.7.115 SvcRequestFileTransfer_2013(cantp_handle, uds_msgconfig, uds_msg, uds_svc_param_rft_moop, UInt16, byte[], byte, byte, byte, byte[], byte[])

Writes to the transmit queue a request for UDS service RequestFileTransfer.

Syntax

Pascal OO

```
class function SvcRequestFileTransfer_2013(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    mode_of_operation: uds_svc_param_rft_moop;
    file_path_and_name_size: UInt16;
    file_path_and_name: PByte;
    compression_method: Byte;
    encrypting_method: Byte;
    file_size_parameter_size: Byte;
    file_size_uncompressed: PByte;
    file_size_compressed: PByte
): uds_status; overload;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcRequestFileTransfer_2013")]
private static extern uds_status SvcRequestFileTransfer_2013(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
```

```

uds_msgconfig request_config,
out uds_msg out_msg_request,
[MarshalAs(UnmanagedType.U1)]
uds_svc_param_rft_moop mode_of_operation,
UInt16 file_path_and_name_size,
[MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 4)]
byte[] file_path_and_name,
byte compression_method,
byte encrypting_method,
byte file_size_parameter_size,
IntPtr file_size_uncompressed,
IntPtr file_size_compressed);

```

C++ / CLR

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcRequestFileTransfer_2013")]
static uds_status SvcRequestFileTransfer_2013(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType::U1)]
    uds_svc_param_rft_moop mode_of_operation,
    UInt16 file_path_and_name_size,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 4)]
    array<Byte> ^file_path_and_name,
    Byte compression_method,
    Byte encrypting_method,
    Byte file_size_parameter_size,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 8)]
    array<Byte> ^file_size_uncompressed,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 8)]
    array<Byte> ^file_size_compressed);

```

Visual Basic

```

<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcRequestFileTransfer_2013")>
Public Shared Function SvcRequestFileTransfer_2013(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.U1)>
    ByVal mode_of_operation As uds_svc_param_rft_moop,
    ByVal file_path_and_name_size As UInt16,
    <MarshalAs(UnmanagedType.LPStr, SizeParamIndex:=4)>
    ByVal file_path_and_name As String,
    ByVal compression_method As Byte,
    ByVal encrypting_method As Byte,
    ByVal file_size_parameter_size As Byte,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=8)>
    ByVal file_size_uncompressed As Byte(),
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=8)>
    ByVal file_size_compressed As Byte()) As uds_status
End Function

```

Parameters

| Parameter | Description |
|----------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |

| Parameter | Description |
|--------------------------|---|
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |
| mode_of_operation | Mode of operation (see <code>uds_svc_param_rft_moop</code> on page 96). |
| file_path_and_name_size | File path and name string length. |
| file_path_and_name | File path and name string. |
| compression_method | A nibble-value that specifies the compression method, the value 0x0 specifies that no compression method is used. |
| encrypting_method | A nibble-value that specifies the encrypting method, the value 0x0 specifies that no encrypting method is used. |
| file_size_parameter_size | File size parameter length in byte |
| file_size_uncompressed | Uncompressed file size. |
| file_size_compressed | Compressed file size. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

Example

The following example shows the use of the service method `SvcRequestFileTransfer_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
```

```

config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical RequestFileTransfer message
Byte[] file_name = Encoding.ASCII.GetBytes("toto.txt");
Byte[] file_size_uncompressed = { 0x00, 0xD };
Byte[] file_size_compressed = { 0x0, 0xA };
result = UDSApi.SvcRequestFileTransfer_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, UDSApi.uds_svc_param_rft_moop.PUDS_SVC_PARAM_RFT_MOOP_ADDFILE, 8, file_name, 0, 0,
    2, file_size_uncompressed, file_size_compressed);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++ / CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical RequestFileTransfer message
array<Byte>^ file_name = Encoding::ASCII->GetBytes("toto.txt");
array<Byte>^ file_size_uncompressed = { 0x00, 0xD };
array<Byte>^ file_size_compressed = { 0x0, 0xA };
result = UDSApi::SvcRequestFileTransfer_2013(PCANTP_HANDLE_USBBUS1, config, request,
    UDSApi::uds_svc_param_rft_moop::PUDS_SVC_PARAM_RFT_MOOP_ADDFILE, 8, file_name, 0, 0, 2,
    file_size_uncompressed, file_size_compressed);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```


Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical RequestFileTransfer message
Dim file_name As Byte() = Encoding.ASCII.GetBytes("toto.txt")
Dim file_size_uncompressed As Byte() = {&H0, &HD}
Dim file_size_compressed As Byte() = {&H0, &HA}
result = UDSApI.SvcRequestFileTransfer_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, UDSApI.uds_svc_param_rft_moop.PUDS_SVC_PARAM_RFT_MOOP_ADDFILE, 8, file_name, 0, 0,
    2, file_size_uncompressed, file_size_compressed)
If UDSApI.StatusIsOk_2013(result) Then
    result = UDSApI.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApI.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApI.MsgFree_2013(request)
UDSApI.MsgFree_2013(response)
UDSApI.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    file_size_uncompressed: array [0 .. 1] of Byte;
    file_size_compressed: array [0 .. 1] of Byte;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    file_name: System.TArray<Byte>;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;

```



```

config.nai.extension_addr := $0;
config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical RequestFileTransfer message
file_name := TEncoding.ASCII.GetBytes('toto.txt');
file_size_uncompressed[0] := $00;
file_size_uncompressed[1] := $D;
file_size_compressed[0] := $00;
file_size_compressed[1] := $A;
result := TUDSApi.SvcRequestFileTransfer_2013
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    uds_svc_param_rft_moop.PUDS_SVC_PARAM_RFT_MOOP_ADDFILE, 8, PByte(file_name),
    0, 0, 2, @file_size_uncompressed, @file_size_compressed);
if TUDSApi.StatusIsOk_2013(result) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
end;
if TUDSApi.StatusIsOk_2013(result) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248, [uds_svc_param_rft_moop](#) on page 96,

Plain function version: [UDS_SvcRequestFileTransfer_2013](#) on page 745.

3.7.116 SvcAuthenticationDA_2020

Writes a UDS request according to the Authentication service's specifications (ISO-14229-1:2020). The subfunction deAuthenticate is implicit.

Syntax

Pascal OO

```

class function SvcAuthenticationDA_2020(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg
): uds_status;

```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcAuthenticationDA_2020")]
public static extern uds_status SvcAuthenticationDA_2020(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcAuthenticationDA_2020")]
static uds_status SvcAuthenticationDA_2020(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcAuthenticationDA_2020")>
Public Shared Function SvcAuthenticationDA_2020(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with Reset_2013 (see Reset_2013 on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see MsgFree_2013 on page 214). |


Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

The PCAN-UDS 2.x API provides `uds_svc_authentication_subfunction` (see on page 98) and `uds_svc_authentication_return_parameter` (see on page 99) enumerations to help user to decode Authentication service responses.

Example

The following example shows the use of the service method `SvcAuthenticationDA_2020` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/deAuthenticate message
result = UDSApi.SvcAuthenticationDA_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/deAuthenticate message
```

```

result = UDSApi::SvcAuthenticationDA_2020(cantp_handle::PCANTP_HANDLE_USBBUS1, config,
    request);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(cantp_handle::PCANTP_HANDLE_USBBUS1, request,
        response, request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical Authentication/deAuthenticate message
result = UDSApi.SvcAuthenticationDA_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);

```

```

FillChar(request_confirmation, sizeof(request_confirmation), 0);
FillChar(response, sizeof(response), 0);

// Set request message configuration
FillChar(config, sizeof(config), 0);
config.can_id :=
    UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr := $0;
config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/deAuthenticate message
result := TUDSApi.SvcAuthenticationDA_2020(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, request);
if (TUDSApi.StatusIsOk_2013(result)) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
end;
if (TUDSApi.StatusIsOk_2013(result)) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```



See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcAuthenticationDA_2020](#) on page 748.

3.7.117 SvcAuthenticationVCU_2020

Writes a UDS request according to the Authentication service's specifications (ISO-14229-1:2020). The subfunction verifyCertificateUnidirectional is implicit.

Overloads

| | Method | Description |
|---|---|---|
|  | SvcAuthenticationVCU_2020(cantp_handle, uds_msgconfig, uds_msg, Byte, byte[], UInt16, byte[], UInt16) | Writes to the transmit queue a request for UDS service Authentication with verifyCertificateUnidirectional subfunction (ISO-14229-1:2020). |
|  | SvcAuthenticationVCU_2020(cantp_handle, uds_msgconfig, uds_msg, Byte, byte[], UInt16) | Writes to the transmit queue a request for UDS service Authentication with verifyCertificateUnidirectional subfunction, without challenge client buffer (ISO-14229-1:2020). |

Plain function version: `UDS_SvcAuthenticationVCU_2020` on page 749.

3.7.118 SvcAuthenticationVCU_2020(cantp_handle, uds_msgconfig, uds_msg, Byte, byte[], UInt16, byte[], UInt16)

Writes a UDS request according to the Authentication service's specifications (ISO-14229-1:2020). The subfunction verifyCertificateUnidirectional is implicit.

Syntax

Pascal OO

```
class function SvcAuthenticationVCU_2020(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    communication_configuration: Byte;
    certificate_client: PByte;
    certificate_client_size: UInt16;
    challenge_client: PByte;
    challenge_client_size: UInt16
): uds_status; overload;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcAuthenticationVCU_2020")]
public static extern uds_status SvcAuthenticationVCU_2020(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    Byte communication_configuration,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 5)]
    byte[] certificate_client,
    UInt16 certificate_client_size,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 7)]
    byte[] challenge_client,
    UInt16 challenge_client_size);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcAuthenticationVCU_2020")]
static uds_status SvcAuthenticationVCU_2020(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    Byte communication_configuration,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 5)]
    array<Byte> ^certificate_client,
    UInt16 certificate_client_size,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 7)]
    array<Byte> ^challenge_client,
    UInt16 challenge_client_size);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcAuthenticationVCU_2020")>
Public S
hared Function SvcAuthenticationVCU_2020(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
```

```

ByVal request_config As uds_msgconfig,
ByRef out_msg_request As uds_msg,
ByVal communication_configuration As Byte,
<MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=5)>
ByVal certificate_client As Byte(),
ByVal certificate_client_size As UInt16,
<MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=7)>
ByVal challenge_client As Byte(),
ByVal challenge_client_size As UInt16) As uds_status
End Function

```

Parameters

| Parameter | Description |
|-----------------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |
| communication_configuration | Configuration information about communication. |
| certificate_client | Buffer containing the certificate of the client. |
| certificate_client_size | Size in bytes of the certificate buffer. |
| challenge_client | Buffer containing the challenge of the client. |
| challenge_client_size | Size in bytes of the challenge buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |


Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

The PCAN-UDS 2.x API provides `uds_svc_authentication_subfunction` (see on page 98) and `uds_svc_authentication_return_parameter` (see on page 99) enumerations to help user to decode Authentication service responses.

Example

The following example shows the use of the service method `SvcAuthenticationVCU_2020(cantp_handle, uds_msgconfig, uds_msg, Byte, byte[], UInt16, byte[], UInt16)` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/verifyCertificateUnidirectional message
Byte communication_configuration = 0x00;
byte[] certificate_client = new byte[2] { 0x12, 0x34 };
UInt16 certificate_client_size = 2;
byte[] challenge_client = new byte[2] { 0x56, 0x78 };
UInt16 challenge_client_size = 2;
result = UDSApi.SvcAuthenticationVCU_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, communication_configuration, certificate_client, certificate_client_size,
    challenge_client, challenge_client_size);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/verifyCertificateUnidirectional message
Byte communication_configuration = 0x00;
array<Byte>^ certificate_client = { 0x12, 0x34 };

```



```

UInt16 certificate_client_size = 2;
array<Byte>^ challenge_client = { 0x56, 0x78 };
UInt16 challenge_client_size = 2;
result = UDSApi::SvcAuthenticationVCU_2020(cantp_handle::PCANTP_HANDLE_USBBUS1, config,
    request, communication_configuration, certificate_client, certificate_client_size,
    challenge_client, challenge_client_size);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(cantp_handle::PCANTP_HANDLE_USBBUS1, request,
        response, request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical Authentication/verifyCertificateUnidirectional message
Dim communication_configuration As Byte = &H0
Dim certificate_client As Byte() = {&H12, &H34}
Dim certificate_client_size As UInt16 = 2
Dim challenge_client As Byte() = {&H56, &H78}
Dim challenge_client_size As UInt16 = 2
result = UDSApi.SvcAuthenticationVCU_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    communication_configuration, certificate_client, certificate_client_size, challenge_client,
    challenge_client_size)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
  result: uds_status;
  request: uds_msg;
  request_confirmation: uds_msg;
  response: uds_msg;
  config: uds_msgconfig;
  communication_configuration: Byte;
  certificate_client: array [0 .. 1] of Byte;
  certificate_client_size: UInt16;
  challenge_client: array [0 .. 1] of Byte;
  challenge_client_size: UInt16;
begin
  FillChar(request, sizeof(request), 0);
  FillChar(request_confirmation, sizeof(request_confirmation), 0);
  FillChar(response, sizeof(response), 0);

  // Set request message configuration
  FillChar(config, sizeof(config), 0);
  config.can_id :=
    UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
  config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
  config.nai.extension_addr := $0;
  config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
  config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
  config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
  config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
  config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

  // Sends a physical Authentication/verifyCertificateUnidirectional message
  communication_configuration := $00;
  certificate_client[0] := $12;
  certificate_client[1] := $34;
  certificate_client_size := 2;
  challenge_client[0] := $56;
  challenge_client[1] := $78;
  challenge_client_size := 2;
  result := TUDSApi.SvcAuthenticationVCU_2020
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
     communication_configuration, PByte(@certificate_client),
     certificate_client_size, PByte(@challenge_client), challenge_client_size);
  if (TUDSApi.StatusIsOk_2013(result)) then
  begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
      @request, response, @request_confirmation);
  end;
  if (TUDSApi.StatusIsOk_2013(result)) then
  begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
  end
  else
  begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
  end;

  // Free structures

```

```
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;
```

See also: `WaitForService_2013` on page 248.

Plain function version: `UDS_SvcAuthenticationVCU_2020` on page 749.

3.7.119 `SvcAuthenticationVCU_2020(cantp_handle, uds_msgconfig, uds_msg, Byte, byte[], UInt16)`

Writes a UDS request according to the Authentication service's specifications (ISO-14229-1:2020) with the subfunction `verifyCertificateUnidirectional`, without challenge client parameter.

Syntax

Pascal OO

```
class function SvcAuthenticationVCU_2020(
  channel: cantp_handle;
  request_config: uds_msgconfig;
  var out_msg_request: uds_msg;
  communication_configuration: Byte;
  certificate_client: PByte;
  certificate_client_size: UInt16
): uds_status; overload;
```

C#

```
public static uds_status SvcAuthenticationVCU_2020(
  cantp_handle channel,
  uds_msgconfig request_config,
  out uds_msg out_msg_request,
  Byte communication_configuration,
  byte[] certificate_client,
  UInt16 certificate_client_size);
```

C++ / CLR

```
static uds_status SvcAuthenticationVCU_2020(
  cantp_handle channel,
  uds_msgconfig request_config,
  uds_msg %out_msg_request,
  Byte communication_configuration,
  array<Byte> ^certificate_client,
  UInt16 certificate_client_size);
```

Visual Basic

```
Public Shared Function SvcAuthenticationVCU_2020(
  ByVal channel As cantp_handle,
  ByVal request_config As uds_msgconfig,
  ByRef out_msg_request As uds_msg,
  ByVal communication_configuration As Byte,
  ByVal certificate_client As Byte(),
  ByVal certificate_client_size As UInt16) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |
| communication_configuration | Configuration information about communication. |
| certificate_client | Buffer containing the certificate of the client. |
| certificate_client_size | Size in bytes of the certificate buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |


Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

The PCAN-UDS 2.x API provides `uds_svc_authentication_subfunction` (see on page 98) and `uds_svc_authentication_return_parameter` (see on page 99) enumerations to help user to decode Authentication service responses.

Example

The following example shows the use of the service method `SvcAuthenticationVCU_2020(cantp_handle, uds_msgconfig, uds_msg, Byte, byte[], UInt16)` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
```

```

config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/verifyCertificateUnidirectional message
Byte communication_configuration = 0x00;
byte[] certificate_client = new byte[2] { 0x12, 0x34 };
UInt16 certificate_client_size = 2;
result = UDSApi.SvcAuthenticationVCU_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, communication_configuration, certificate_client, certificate_client_size);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/verifyCertificateUnidirectional message
Byte communication_configuration = 0x00;
array<Byte>^ certificate_client = { 0x12, 0x34 };
UInt16 certificate_client_size = 2;
result = UDSApi::SvcAuthenticationVCU_2020(cantp_handle::PCANTP_HANDLE_USBBUS1, config,
    request, communication_configuration, certificate_client, certificate_client_size);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(cantp_handle::PCANTP_HANDLE_USBBUS1, request,
        response, request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical Authentication/verifyCertificateUnidirectional message
Dim communication_configuration As Byte = &H0
Dim certificate_client As Byte() = {&H12, &H34}
Dim certificate_client_size As UInt16 = 2
result = UDSApi.SvcAuthenticationVCU_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
communication_configuration, certificate_client, certificate_client_size)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    communication_configuration: Byte;
    certificate_client: array [0 .. 1] of Byte;
    certificate_client_size: UInt16;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;

```

```

config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/verifyCertificateUnidirectional message
communication_configuration := $00;
certificate_client[0] := $12;
certificate_client[1] := $34;
certificate_client_size := 2;
result := TUDSApi.SvcAuthenticationVCU_2020
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    communication_configuration, PByte(@certificate_client),
    certificate_client_size);
if (TUDSApi.StatusIsOk_2013(result)) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
end;
if (TUDSApi.StatusIsOk_2013(result)) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcAuthenticationVCU_2020](#) on page 749.

3.7.120 svcAuthenticationVCB_2020

Writes a UDS request according to the Authentication service's specifications (ISO-14229-1:2020). The subfunction verifyCertificateBidirectional is implicit.

Syntax

Pascal OO

```

class function SvcAuthenticationVCB_2020(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    communication_configuration: Byte;
    certificate_client: PByte;
    certificate_client_size: UInt16;

```



```
challenge_client: PByte;
challenge_client_size: UInt16
): uds_status;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcAuthenticationVCB_2020")]
public static extern uds_status SvcAuthenticationVCB_2020(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    Byte communication_configuration,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 5)]
    byte[] certificate_client,
    UInt16 certificate_client_size,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 7)]
    byte[] challenge_client,
    UInt16 challenge_client_size);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcAuthenticationVCB_2020")]
static uds_status SvcAuthenticationVCB_2020(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    Byte communication_configuration,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 5)]
    array<Byte> ^certificate_client,
    UInt16 certificate_client_size,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 7)]
    array<Byte> ^challenge_client,
    UInt16 challenge_client_size);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcAuthenticationVCB_2020")>
Public Shared Function SvcAuthenticationVCB_2020(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal communication_configuration As Byte,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=5)>
    ByVal certificate_client As Byte(),
    ByVal certificate_client_size As UInt16,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=7)>
    ByVal challenge_client As Byte(),
    ByVal challenge_client_size As UInt16) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| communication_configuration | Configuration information about communication. |
| certificate_client | Buffer containing the certificate of the client. |

| Parameter | Description |
|-------------------------|--|
| certificate_client_size | Size in bytes of the certificate buffer. |
| challenge_client | Buffer containing the challenge of the client. |
| challenge_client_size | Size in bytes of the challenge buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |


Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

The PCAN-UDS 2.x API provides `uds_svc_authentication_subfunction` (see on page 98) and `uds_svc_authentication_return_parameter` (see on page 99) enumerations to help user to decode Authentication service responses.

Example

The following example shows the use of the service method `SvcAuthenticationVCB_2020` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x00;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/verifyCertificateBidirectional message
Byte communication_configuration = 0x00;
```

```

byte[] certificate_client = new byte[2] { 0x12, 0x34 };
UInt16 certificate_client_size = 2;
byte[] challenge_client = new byte[2] { 0x56, 0x78 };
UInt16 challenge_client_size = 2;
result = UDSApi.SvcAuthenticationVCB_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, communication_configuration, certificate_client, certificate_client_size,
    challenge_client, challenge_client_size);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/verifyCertificateBidirectional message
Byte communication_configuration = 0x00;
array<Byte>^ certificate_client = { 0x12, 0x34 };
UInt16 certificate_client_size = 2;
array<Byte>^ challenge_client = { 0x56, 0x78 };
UInt16 challenge_client_size = 2;
result = UDSApi::SvcAuthenticationVCB_2020(cantp_handle::PCANTP_HANDLE_USBBUS1, config,
    request, communication_configuration, certificate_client, certificate_client_size,
    challenge_client, challenge_client_size);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(cantp_handle::PCANTP_HANDLE_USBBUS1, request,
        response, request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical Authentication/verifyCertificateBidirectional message
Dim communication_configuration As Byte = &H0
Dim certificate_client As Byte() = {&H12, &H34}
Dim certificate_client_size As UInt16 = 2
Dim challenge_client As Byte() = {&H56, &H78}
Dim challenge_client_size As UInt16 = 2
result = UDSApi.SvcAuthenticationVCB_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    communication_configuration, certificate_client, certificate_client_size, challenge_client,
    challenge_client_size)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    communication_configuration: Byte;
    certificate_client: array [0 .. 1] of Byte;
    certificate_client_size: UInt16;
    challenge_client: array [0 .. 1] of Byte;
    challenge_client_size: UInt16;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration

```

```

FillChar(config, sizeof(config), 0);
config.can_id :=
    UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr := $0;
config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/verifyCertificateBidirectional message
communication_configuration := $00;
certificate_client[0] := $12;
certificate_client[1] := $34;
certificate_client_size := 2;
challenge_client[0] := $56;
challenge_client[1] := $78;
challenge_client_size := 2;
result := TUDSApi.SvcAuthenticationVCB_2020
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    communication_configuration, PByte(@certificate_client),
    certificate_client_size, PByte(@challenge_client), challenge_client_size);
if (TUDSApi.StatusIsOk_2013(result)) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    @request, response, @request_confirmation);
end;
if (TUDSApi.StatusIsOk_2013(result)) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK)
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```



See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcAuthenticationVCB_2020](#) on page 752.

3.7.121 SvcAuthenticationPOWN_2020

Writes a UDS request according to the Authentication service's specifications (ISO-14229-1:2020). The subfunction proofOfOwnership is implicit.

Overloads

| | Method | Description |
|---|--|---|
|  | SvcAuthenticationPOWN_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], UInt16, byte[], UInt16) | Writes to the transmit queue a request for UDS service Authentication with proofOfOwnership subfunction (ISO-14229-1:2020). |
|  | SvcAuthenticationPOWN_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], UInt16) | Writes to the transmit queue a request for UDS service Authentication with proofOfOwnership subfunction, without ephemeral public key (ISO-14229-1:2020). |

Plain function version: `UDS_SvcAuthenticationPOWN_2020` on page 754.

3.7.122 SvcAuthenticationPOWN_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], UInt16, byte[], UInt16)

Writes a UDS request according to the Authentication service's specifications (ISO-14229-1:2020). The subfunction proofOfOwnership is implicit.

Syntax

Pascal OO

```
class function SvcAuthenticationPOWN_2020(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    proof_of_ownership_client: PByte;
    proof_of_ownership_client_size: UInt16;
    ephemeral_public_key_client: PByte;
    ephemeral_public_key_client_size: UInt16
): uds_status; overload;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcAuthenticationPOWN_2020")]
public static extern uds_status SvcAuthenticationPOWN_2020(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 4)]
    byte[] proof_of_ownership_client,
    UInt16 proof_of_ownership_client_size,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 6)]
    byte[] ephemeral_public_key_client,
    UInt16 ephemeral_public_key_client_size);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcAuthenticationPOWN_2020")]
static uds_status SvcAuthenticationPOWN_2020(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 4)]
    array<Byte> ^proof_of_ownership_client,
```

```

UInt16 proof_of_ownership_client_size,
[MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 6)]
array<Byte> ^ephemeral_public_key_client,
UInt16 ephemeral_public_key_client_size);

```

Visual Basic

```

<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcAuthenticationPOWN_2020")>
Public Shared Function SvcAuthenticationPOWN_2020(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=4)>
    ByVal proof_of_ownership_client As Byte(),
    proof_of_ownership_client_size As UInt16,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=6)>
    ByVal ephemeral_public_key_client As Byte(),
    ephemeral_public_key_client_size As UInt16) As uds_status
End Function

```

Parameters

| Parameter | Description |
|----------------------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |
| proof_of_ownership_client | Buffer containing the proof of ownership of the client. |
| proof_of_ownership_client_size | Size in bytes of the proof of ownership buffer. |
| ephemeral_public_key_client | Buffer containing the ephemeral public key of the client. |
| ephemeral_public_key_client_size | Size in bytes of the ephemeral public key buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |


Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

The PCAN-UDS 2.x API provides `uds_svc_authentication_subfunction` (see on page 98) and `uds_svc_authentication_return_parameter` (see on page 99) enumerations to help user to decode Authentication service responses.

Example

The following example shows the use of the service method `SvcAuthenticationPOWN_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], UInt16, byte[], UInt16)` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/proofOfOwnership message
byte[] proof_of_ownership_client = new byte[2] { 0x12, 0x34 };
UInt16 proof_of_ownership_client_size = 2;
byte[] ephemeral_public_key_client = new byte[2] { 0x56, 0x78 };
UInt16 ephemeral_public_key_client_size = 2;
result = UDSApi.SvcAuthenticationPOWN_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, proof_of_ownership_client, proof_of_ownership_client_size,
    ephemeral_public_key_client, ephemeral_public_key_client_size);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
```



```

config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/proofOfOwnership message
array<Byte>^ proof_of_ownership_client = { 0x12, 0x34 };
UInt16 proof_of_ownership_client_size = 2;
array<Byte>^ ephemeral_public_key_client = { 0x56, 0x78 };
UInt16 ephemeral_public_key_client_size = 2;
result = UDSApi::SvcAuthenticationPOW_2020(cantp_handle::PCANTP_HANDLE_USBBUS1, config,
    request, proof_of_ownership_client, proof_of_ownership_client_size,
    ephemeral_public_key_client, ephemeral_public_key_client_size);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(cantp_handle::PCANTP_HANDLE_USBBUS1, request,
        response, request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical Authentication/proofOfOwnership message
Dim proof_of_ownership_client As Byte() = {&H12, &H34}
Dim proof_of_ownership_client_size As UInt16 = 2
Dim ephemeral_public_key_client As Byte() = {&H56, &H78}
Dim ephemeral_public_key_client_size As UInt16 = 2
result = UDSApi.SvcAuthenticationPOW_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    proof_of_ownership_client, proof_of_ownership_client_size, ephemeral_public_key_client,
    ephemeral_public_key_client_size)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")

```


End If

```
' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)
```

Pascal OO

```
var
  result: uds_status;
  request: uds_msg;
  request_confirmation: uds_msg;
  response: uds_msg;
  config: uds_msgconfig;
  proof_of_ownership_client: array [0 .. 1] of Byte;
  proof_of_ownership_client_size: UInt16;
  ephemeral_public_key_client: array [0 .. 1] of Byte;
  ephemeral_public_key_client_size: UInt16;
begin
  FillChar(request, sizeof(request), 0);
  FillChar(request_confirmation, sizeof(request_confirmation), 0);
  FillChar(response, sizeof(response), 0);

  // Set request message configuration
  FillChar(config, sizeof(config), 0);
  config.can_id :=
    UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
  config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
  config.nai.extension_addr := $0;
  config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
  config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
  config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
  config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
  config.typem := uds_msgtype.PUDS_MSGTYPE_USDT;

  // Sends a physical Authentication/proofOfOwnership message
  proof_of_ownership_client[0] := $12;
  proof_of_ownership_client[1] := $34;
  proof_of_ownership_client_size := 2;
  ephemeral_public_key_client[0] := $56;
  ephemeral_public_key_client[1] := $78;
  ephemeral_public_key_client_size := 2;
  result := TUDSApi.SvcAuthenticationPOWN_2020
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
     PByte(@proof_of_ownership_client), proof_of_ownership_client_size,
     PByte(@ephemeral_public_key_client), ephemeral_public_key_client_size);
  if (TUDSApi.StatusIsOk_2013(result)) then
  begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
      @request, response, @request_confirmation);
  end;
  if (TUDSApi.StatusIsOk_2013(result)) then
  begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
  end
  else
  begin
```

```
// An error occurred
MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;
```

See also: `WaitForService_2013` on page 248.

Plain function version: `UDS_SvcAuthenticationPOWN_2020` on page 754.

3.7.123 SvcAuthenticationPOWN_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], UInt16)

Writes a UDS request according to the Authentication service's specifications (ISO-14229-1:2020) without the subfunction proofOfOwnership, without ephemeral public key.

Syntax

Pascal OO

```
class function SvcAuthenticationPOWN_2020(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    proof_of_ownership_client: PByte;
    proof_of_ownership_client_size: UInt16
): uds_status; overload;
```

C#

```
public static uds_status SvcAuthenticationPOWN_2020(
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    byte[] proof_of_ownership_client,
    UInt16 proof_of_ownership_client_size);
```

C++ / CLR

```
static uds_status SvcAuthenticationPOWN_2020(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    array<Byte> ^proof_of_ownership_client,
    UInt16 proof_of_ownership_client_size);
```

Visual Basic

```
Public Shared Function SvcAuthenticationPOWN_2020(
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal proof_of_ownership_client As Byte(),
    ByVal proof_of_ownership_client_size As UInt16) As uds_status
End Function
```

Parameters

| Parameter | Description |
|--------------------------------|---|
| channel | The handle of a PUDS channel (see <code>can_tp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |
| proof_of_ownership_client | Buffer containing the proof of ownership of the client. |
| proof_of_ownership_client_size | Size in bytes of the proof of ownership buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |


Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

The PCAN-UDS 2.x API provides `uds_svc_authentication_subfunction` (see on page 98) and `uds_svc_authentication_return_parameter` (see on page 99) enumerations to help user to decode Authentication service responses.

Example

The following example shows the use of the service method `SvcAuthenticationPOW_N_2020`(`can_tp_handle`, `uds_msgconfig`, `uds_msg`, `byte[]`, `UInt16`) on the channel `PCANTP_HANDLE_USBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = can_tp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
```

```

config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/proofOfOwnership message
byte[] proof_of_ownership_client = new byte[2] { 0x12, 0x34 };
UInt16 proof_of_ownership_client_size = 2;
result = UDSApi.SvcAuthenticationPOWN_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, proof_of_ownership_client, proof_of_ownership_client_size);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/proofOfOwnership message
array<Byte>^ proof_of_ownership_client = { 0x12, 0x34 };
UInt16 proof_of_ownership_client_size = 2;
result = UDSApi::SvcAuthenticationPOWN_2020(cantp_handle::PCANTP_HANDLE_USBBUS1, config,
    request, proof_of_ownership_client, proof_of_ownership_client_size);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(cantp_handle::PCANTP_HANDLE_USBBUS1, request,
        response, request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical Authentication/proofOfOwnership message
Dim proof_of_ownership_client As Byte() = {&H12, &H34}
Dim proof_of_ownership_client_size As UInt16 = 2
result = UDSApi.SvcAuthenticationPOWN_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    proof_of_ownership_client, proof_of_ownership_client_size)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    proof_of_ownership_client: array [0 .. 1] of Byte;
    proof_of_ownership_client_size: UInt16;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;

```

```

config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/proofOfOwnership message
proof_of_ownership_client[0] := $12;
proof_of_ownership_client[1] := $34;
proof_of_ownership_client_size := 2;
result := TUDSApi.SvcAuthenticationPOWN_2020
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    PByte(@proof_of_ownership_client), proof_of_ownership_client_size);
if (TUDSApi.StatusIsOk_2013(result)) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    @request, response, @request_confirmation);
end;
if (TUDSApi.StatusIsOk_2013(result)) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcAuthenticationPOWN_2020](#) on page 754.

3.7.124 SvcAuthenticationRCFA_2020

Writes a UDS request according to the Authentication service's specifications (ISO-14229-1:2020). The subfunction requestChallengeForAuthentication is implicit.

Syntax

Pascal OO

```

class function SvcAuthenticationRCFA_2020(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    communication_configuration: Byte;
    algorithm_indicator: PByte
): uds_status;

```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcAuthenticationRCFA_2020")]
public static extern uds_status SvcAuthenticationRCFA_2020(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    Byte communication_configuration,
    byte[] algorithm_indicator);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcAuthenticationRCFA_2020")]
static uds_status SvcAuthenticationRCFA_2020(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    Byte communication_configuration,
    array<Byte> ^algorithm_indicator);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcAuthenticationRCFA_2020")>
Public Shared Function SvcAuthenticationRCFA_2020(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal communication_configuration As Byte,
    ByVal algorithm_indicator As Byte()) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| communication_configuration | Configuration information about communication. |
| algorithm_indicator | Buffer of 16 bytes containing the algorithm indicator. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is not valid. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | The mapping given in the message configuration is unknown. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with Reset_2013 (see Reset_2013 on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see MsgFree_2013 on page 214). |


Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see [MsgFree_2013](#) on page 214).

The PCAN-UDS 2.x API provides [uds_svc_authentication_subfunction](#) (see on page 98) and [uds_svc_authentication_return_parameter](#) (see on page 99) enumerations to help user to decode Authentication service responses.

Example

The following example shows the use of the service method [SvcAuthenticationRCFA_2020](#) on the channel [PCANTP_HANDLE_USBBUS1](#). A UDS physical service request is transmitted, and the [WaitForService_2013](#) method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/requestChallengeForAuthentication message
Byte communication_configuration = 0x00;
byte[] algorithm_indicator = new byte[16] { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F };
result = UDSApi.SvcAuthenticationRCFA_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, communication_configuration, algorithm_indicator);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
```



```

uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/requestChallengeForAuthentication message
Byte communication_configuration = 0x00;
array<Byte>^ algorithm_indicator = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
    0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F };
result = UDSApi::SvcAuthenticationRCFA_2020(cantp_handle::PCANTP_HANDLE_USBBUS1, config,
    request, communication_configuration, algorithm_indicator);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(cantp_handle::PCANTP_HANDLE_USBBUS1, request,
        response, request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical Authentication/requestChallengeForAuthentication message
Dim communication_configuration As Byte = &H0
Dim algorithm_indicator As Byte() = {&H0, &H1, &H2, &H3, &H4, &H5, &H6, &H7, &H8, &H9, &HA,
    &HB, &HC, &HD, &HE, &HF}
result = UDSApi.SvcAuthenticationRCFA_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    communication_configuration, algorithm_indicator)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")

```

```

Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    communication_configuration: Byte;
    algorithm_indicator: array [0 .. 15] of Byte;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.typem := uds_msgtype.PUDS_MSGTYPE_USDT;

    // Sends a physical Authentication/requestChallengeForAuthentication message
    communication_configuration := $00;
    algorithm_indicator[0] := $00;
    algorithm_indicator[1] := $01;
    algorithm_indicator[2] := $02;
    algorithm_indicator[3] := $03;
    algorithm_indicator[4] := $04;
    algorithm_indicator[5] := $05;
    algorithm_indicator[6] := $06;
    algorithm_indicator[7] := $07;
    algorithm_indicator[8] := $08;
    algorithm_indicator[9] := $09;
    algorithm_indicator[10] := $0A;
    algorithm_indicator[11] := $0B;
    algorithm_indicator[12] := $0C;
    algorithm_indicator[13] := $0D;
    algorithm_indicator[14] := $0E;
    algorithm_indicator[15] := $0F;
    result := TUDSApi.SvcAuthenticationRCFA_2020
        (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
        communication_configuration, PByte(@algorithm_indicator));

```

```

if (TUDSApi.StatusIsOk_2013(result)) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
end;
if (TUDSApi.StatusIsOk_2013(result)) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```




See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcAuthenticationRCFA_2020](#) on page 756.

3.7.125 SvcAuthenticationVPOWNU_2020

Writes a UDS request according to the Authentication service's specifications (ISO-14229-1:2020). The subfunction `verifyProofOfOwnershipUnidirectional` is implicit.

Overloads

| | Method | Description |
|---|---|--|
|  | <code>SvcAuthenticationVPOWNU_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], byte[], UInt16, byte[], UInt16, byte[], UInt16)</code> | Writes to the transmit queue a request for UDS service Authentication with <code>verifyProofOfOwnershipUnidirectional</code> subfunction (ISO-14229-1:2020). |
|  | <code>SvcAuthenticationVPOWNU_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], byte[], UInt16, byte[], UInt16)</code> | Writes to the transmit queue a request for UDS service Authentication with <code>verifyProofOfOwnershipUnidirectional</code> subfunction, without additional parameters (ISO-14229-1:2020). |
|  | <code>SvcAuthenticationVPOWNU_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], byte[], UInt16)</code> | Writes to the transmit queue a request for UDS service Authentication with <code>verifyProofOfOwnershipUnidirectional</code> subfunction, without additional parameters, without challenge client buffer (ISO-14229-1:2020). |

Plain function version: [UDS_SvcAuthenticationVPOWNU_2020](#) on page 758.

3.7.126 SvcAuthenticationVPOWNU_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], byte[], UInt16, byte[], UInt16, byte[], UInt16)

Writes a UDS request according to the Authentication service's specifications (ISO-14229-1:2020). The subfunction `verifyProofOfOwnershipUnidirectional` is implicit.

Syntax

Pascal OO

```

class function SvcAuthenticationVPOWNU_2020(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;

```

```

algorithm_indicator: PByte;
proof_of_ownership_client: PByte;
proof_of_ownership_client_size: UInt16;
challenge_client: PByte;
challenge_client_size: UInt16;
additional_parameter: PByte;
additional_parameter_size: UInt16
): uds_status; overload;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcAuthenticationVPOWNU_2020")]
public static extern uds_status SvcAuthenticationVPOWNU_2020(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 4)]
    byte[] algorithm_indicator,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 5)]
    byte[] proof_of_ownership_client,
    UInt16 proof_of_ownership_client_size,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 7)]
    byte[] challenge_client,
    UInt16 challenge_client_size,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 9)]
    byte[] additional_parameter,
    UInt16 additional_parameter_size);

```

C++ / CLR

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcAuthenticationVPOWNU_2020")]
static uds_status SvcAuthenticationVPOWNU_2020(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 4)]
    array<Byte> ^algorithm_indicator,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 5)]
    array<Byte> ^proof_of_ownership_client,
    UInt16 proof_of_ownership_client_size,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 7)]
    array<Byte> ^challenge_client,
    UInt16 challenge_client_size,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 9)]
    array<Byte> ^additional_parameter,
    UInt16 additional_parameter_size);

```

Visual Basic

```

<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcAuthenticationVPOWNU_2020")>
Public Shared Function SvcAuthenticationVPOWNU_2020(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=4)>
    ByVal algorithm_indicator As Byte(),
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=5)>
    ByVal proof_of_ownership_client As Byte(),
    ByVal proof_of_ownership_client_size As UInt16,

```

```

    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=7)>
    ByVal challenge_client As Byte(),
    ByVal challenge_client_size As UInt16,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=9)>
    ByVal additional_parameter As Byte(),
    ByVal additional_parameter_size As UInt16) As uds_status
End Function

```

Parameters

| Parameter | Description |
|--------------------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the method (see <code>uds_msg</code> on page 21). |
| algorithm_indicator | Buffer of 16 bytes containing the algorithm indicator |
| proof_of_ownership_client | Buffer containing the proof of ownership of the client. |
| proof_of_ownership_client_size | Size in bytes of the proof of ownership buffer. |
| challenge_client | Buffer containing the challenge of the client. |
| challenge_client_size | Size in bytes of the challenge buffer. |
| additional_parameter | Buffer containing additional parameters. |
| additional_parameter_size | Size in bytes of the additional parameter buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |


Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

The PCAN-UDS 2.x API provides `uds_svc_authentication_subfunction` (see on page 98) and `uds_svc_authentication_return_parameter` (see on page 99) enumerations to help user to decode Authentication service responses.

Example

The following example shows the use of the service method `SvcAuthenticationVPOWNU_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], byte[], UInt16, byte[], UInt16, byte[], UInt16)` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/verifyProofOfOwnershipUnidirectional message
byte[] algorithm_indicator = new byte[16] { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F };
byte[] proof_of_ownership_client = new byte[2] { 0x12, 0x34 };
UInt16 proof_of_ownership_client_size = 2;
byte[] challenge_client = new byte[2] { 0x56, 0x78 };
UInt16 challenge_client_size = 2;
byte[] additional_parameter = new byte[2] { 0x9A, 0xBC };
UInt16 additional_parameter_size = 2;
result = UDSApi.SvcAuthenticationVPOWNU_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, algorithm_indicator, proof_of_ownership_client, proof_of_ownership_client_size,
    challenge_client, challenge_client_size, additional_parameter, additional_parameter_size);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

```

```
// Sends a physical Authentication/verifyProofOfOwnershipUnidirectional message
array<Byte>^ algorithm_indicator = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
    0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F };
array<Byte>^ proof_of_ownership_client = { 0x12, 0x34 };
UInt16 proof_of_ownership_client_size = 2;
array<Byte>^ challenge_client = { 0x56, 0x78 };
UInt16 challenge_client_size = 2;
array<Byte>^ additional_parameter = { 0x9A, 0xBC };
UInt16 additional_parameter_size = 2;
result = UDSApi::SvcAuthenticationVPOWNU_2020(cantp_handle::PCANTP_HANDLE_USBBUS1, config,
    request, algorithm_indicator, proof_of_ownership_client, proof_of_ownership_client_size,
    challenge_client, challenge_client_size, additional_parameter,
    additional_parameter_size);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(cantp_handle::PCANTP_HANDLE_USBBUS1, request,
        response, request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);
```

Visual Basic

```
Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical Authentication/verifyProofOfOwnershipUnidirectional message
Dim algorithm_indicator As Byte() = {&H0, &H1, &H2, &H3, &H4, &H5, &H6, &H7, &H8, &H9, &HA,
    &HB, &HC, &HD, &HE, &HF}
Dim proof_of_ownership_client As Byte() = {&H12, &H34}
Dim proof_of_ownership_client_size As UInt16 = 2
Dim challenge_client As Byte() = {&H56, &H78}
Dim challenge_client_size As UInt16 = 2
Dim additional_parameter As Byte() = {&H9A, &HBC}
Dim additional_parameter_size As UInt16 = 2
result = UDSApi.SvcAuthenticationVPOWNU_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, algorithm_indicator, proof_of_ownership_client, proof_of_ownership_client_size,
    challenge_client, challenge_client_size, additional_parameter, additional_parameter_size)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
```



```

    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    algorithm_indicator: array [0 .. 15] of Byte;
    proof_of_ownership_client: array [0 .. 1] of Byte;
    proof_of_ownership_client_size: UInt16;
    challenge_client: array [0 .. 1] of Byte;
    challenge_client_size: UInt16;
    additional_parameter: array [0 .. 1] of Byte;
    additional_parameter_size: UInt16;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=
        cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
    config.type := uds_msgtype.PUDS_MSGTYPE_USDT;

    // Sends a physical Authentication/verifyProofOfOwnershipUnidirectional message
    algorithm_indicator[0] := $00;
    algorithm_indicator[1] := $01;
    algorithm_indicator[2] := $02;
    algorithm_indicator[3] := $03;
    algorithm_indicator[4] := $04;
    algorithm_indicator[5] := $05;
    algorithm_indicator[6] := $06;
    algorithm_indicator[7] := $07;
    algorithm_indicator[8] := $08;
    algorithm_indicator[9] := $09;
    algorithm_indicator[10] := $0A;
    algorithm_indicator[11] := $0B;
    algorithm_indicator[12] := $0C;
    algorithm_indicator[13] := $0D;

```



```

algorithm_indicator[14] := $0E;
algorithm_indicator[15] := $0F;
proof_of_ownership_client[0] := $12;
proof_of_ownership_client[1] := $34;
proof_of_ownership_client_size := 2;
challenge_client[0] := $56;
challenge_client[1] := $78;
challenge_client_size := 2;
additional_parameter[0] := $9A;
additional_parameter[1] := $BC;
additional_parameter_size := 2;
result := TUDSApi.SvcAuthenticationVPOWNU_2020
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    PByte(@algorithm_indicator), PByte(@proof_of_ownership_client),
    proof_of_ownership_client_size, PByte(@challenge_client),
    challenge_client_size, PByte(@additional_parameter),
    additional_parameter_size);
if (TUDSApi.StatusIsOk_2013(result)) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
end;
if (TUDSApi.StatusIsOk_2013(result)) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: `WaitForService_2013` on page 248.

Plain function version: `UDS_SvcAuthenticationVPOWNU_2020` on page 758.

3.7.127 `SvcAuthenticationVPOWNU_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], byte[], UInt16, byte[], UInt16)`

Writes a UDS request according to the Authentication service's specifications with `verifyProofOfOwnershipUnidirectional` subfunction, without additional parameters (ISO-14229-1:2020).

Syntax

Pascal OO

```

class function SvcAuthenticationVPOWNU_2020(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    algorithm_indicator: PByte;
    proof_of_ownership_client: PByte;
    proof_of_ownership_client_size: UInt16;
    challenge_client: PByte;
    challenge_client_size: UInt16
): uds_status; overload;

```

C#

```
public static uds_status SvcAuthenticationVPOWNU_2020(
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    byte[] algorithm_indicator,
    byte[] proof_of_ownership_client,
    UInt16 proof_of_ownership_client_size,
    byte[] challenge_client,
    UInt16 challenge_client_size);
```

C++ / CLR

```
static uds_status SvcAuthenticationVPOWNU_2020(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    array<Byte> ^algorithm_indicator,
    array<Byte> ^proof_of_ownership_client,
    UInt16 proof_of_ownership_client_size,
    array<Byte> ^challenge_client,
    UInt16 challenge_client_size);
```

Visual Basic

```
Public Shared Function SvcAuthenticationVPOWNU_2020(
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal algorithm_indicator As Byte(),
    ByVal proof_of_ownership_client As Byte(),
    ByVal proof_of_ownership_client_size As UInt16,
    ByVal challenge_client As Byte(),
    ByVal challenge_client_size As UInt16) As uds_status
End Function
```

Parameters

| Parameter | Description |
|--------------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| algorithm_indicator | Buffer of 16 bytes containing the algorithm indicator |
| proof_of_ownership_client | Buffer containing the proof of ownership of the client. |
| proof_of_ownership_client_size | Size in bytes of the proof of ownership buffer. |
| challenge_client | Buffer containing the challenge of the client. |
| challenge_client_size | Size in bytes of the challenge buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|-------------------------------------|--|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is not valid. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | The mapping given in the message configuration is unknown. |

| | |
|---|---|
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |


Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

The PCAN-UDS 2.x API provides `uds_svc_authentication_subfunction` (see on page 98) and `uds_svc_authentication_return_parameter` (see on page 99) enumerations to help user to decode Authentication service responses.

Example

The following example shows the use of the service method `SvcAuthenticationVPOWNU_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], byte[], UInt16, byte[], UInt16)` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/verifyProofOfOwnershipUnidirectional message
byte[] algorithm_indicator = new byte[16] { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F };
byte[] proof_of_ownership_client = new byte[2] { 0x12, 0x34 };
UInt16 proof_of_ownership_client_size = 2;
byte[] challenge_client = new byte[2] { 0x56, 0x78 };
UInt16 challenge_client_size = 2;
result = UDSApi.SvcAuthenticationVPOWNU_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, algorithm_indicator, proof_of_ownership_client, proof_of_ownership_client_size,
    challenge_client, challenge_client_size);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else

```

```
// An error occurred
MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/verifyProofOfOwnershipUnidirectional message
array<Byte>^ algorithm_indicator = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
    0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F };
array<Byte>^ proof_of_ownership_client = { 0x12, 0x34 };
UInt16 proof_of_ownership_client_size = 2;
array<Byte>^ challenge_client = { 0x56, 0x78 };
UInt16 challenge_client_size = 2;
result = UDSApi::SvcAuthenticationVPOWNU_2020(cantp_handle::PCANTP_HANDLE_USBBUS1, config,
    request, algorithm_indicator, proof_of_ownership_client, proof_of_ownership_client_size,
    challenge_client, challenge_client_size);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(cantp_handle::PCANTP_HANDLE_USBBUS1, request,
        response, request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);
```

Visual Basic

```
Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
```

```

config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical Authentication/verifyProofOfOwnershipUnidirectional message
Dim algorithm_indicator As Byte() = {&H0, &H1, &H2, &H3, &H4, &H5, &H6, &H7, &H8, &H9, &HA,
    &HB, &HC, &HD, &HE, &HF}
Dim proof_of_ownership_client As Byte() = {&H12, &H34}
Dim proof_of_ownership_client_size As UInt16 = 2
Dim challenge_client As Byte() = {&H56, &H78}
Dim challenge_client_size As UInt16 = 2
result = UDSApi.SvcAuthenticationVPOWNU_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, algorithm_indicator, proof_of_ownership_client, proof_of_ownership_client_size,
    challenge_client, challenge_client_size)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;

    algorithm_indicator: array [0 .. 15] of Byte;
    proof_of_ownership_client: array [0 .. 1] of Byte;
    proof_of_ownership_client_size: UInt16;
    challenge_client: array [0 .. 1] of Byte;
    challenge_client_size: UInt16;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=

```

```

    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/verifyProofOfOwnershipUnidirectional message
algorithm_indicator[0] := $00;
algorithm_indicator[1] := $01;
algorithm_indicator[2] := $02;
algorithm_indicator[3] := $03;
algorithm_indicator[4] := $04;
algorithm_indicator[5] := $05;
algorithm_indicator[6] := $06;
algorithm_indicator[7] := $07;
algorithm_indicator[8] := $08;
algorithm_indicator[9] := $09;
algorithm_indicator[10] := $0A;
algorithm_indicator[11] := $0B;
algorithm_indicator[12] := $0C;
algorithm_indicator[13] := $0D;
algorithm_indicator[14] := $0E;
algorithm_indicator[15] := $0F;
proof_of_ownership_client[0] := $12;
proof_of_ownership_client[1] := $34;
proof_of_ownership_client_size := 2;
challenge_client[0] := $56;
challenge_client[1] := $78;
challenge_client_size := 2;
result := TUDSApi.SvcAuthenticationVPOWNU_2020
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    PByte(@algorithm_indicator), PByte(@proof_of_ownership_client),
    proof_of_ownership_client_size, PByte(@challenge_client),
    challenge_client_size);
if (TUDSApi.StatusIsOk_2013(result)) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
end;
if (TUDSApi.StatusIsOk_2013(result)) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcAuthenticationVPOWNU_2020](#) on page 758.

3.7.128 SvcAuthenticationVPOWNU_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], byte[], UInt16)

Writes a UDS request according to the Authentication service's specifications with verifyProofOfOwnershipUnidirectional subfunction, without additional parameters, without challenge client buffer (ISO-14229-1:2020).

Syntax

Pascal OO

```
class function SvcAuthenticationVPOWNU_2020(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    algorithm_indicator: PByte;
    proof_of_ownership_client: PByte;
    proof_of_ownership_client_size: UInt16
): uds_status; overload;
```

C#

```
public static uds_status SvcAuthenticationVPOWNU_2020(
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    byte[] algorithm_indicator,
    byte[] proof_of_ownership_client,
    UInt16 proof_of_ownership_client_size)
```

C++ / CLR

```
static uds_status SvcAuthenticationVPOWNU_2020(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    array<Byte> ^algorithm_indicator,
    array<Byte> ^proof_of_ownership_client,
    UInt16 proof_of_ownership_client_size);
```

Visual Basic

```
Public Shared Function SvcAuthenticationVPOWNU_2020(
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal algorithm_indicator As Byte(),
    ByVal proof_of_ownership_client As Byte(),
    ByVal proof_of_ownership_client_size As UInt16) As uds_status
End Function
```

Parameters

| Parameter | Description |
|--------------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| algorithm_indicator | Buffer of 16 bytes containing the algorithm indicator |
| proof_of_ownership_client | Buffer containing the proof of ownership of the client. |
| proof_of_ownership_client_size | Size in bytes of the proof of ownership buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

The PCAN-UDS 2.x API provides `uds_svc_authentication_subfunction` (see on page 98) and `uds_svc_authentication_return_parameter` (see on page 99) enumerations to help user to decode Authentication service responses.

Example

The following example shows the use of the service method `SvcAuthenticationVPOWNU_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], byte[], UInt16)` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/verifyProofOfOwnershipUnidirectional message
byte[] algorithm_indicator = new byte[16] { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F };
byte[] proof_of_ownership_client = new byte[2] { 0x12, 0x34 };
UInt16 proof_of_ownership_client_size = 2;
result = UDSApi.SvcAuthenticationVPOWNU_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, algorithm_indicator, proof_of_ownership_client, proof_of_ownership_client_size);

```



```

if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDDS_MSGTYPE_USDT;

// Sends a physical Authentication/verifyProofOfOwnershipUnidirectional message
array<Byte>^ algorithm_indicator = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
    0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F };
array<Byte>^ proof_of_ownership_client = { 0x12, 0x34 };
UInt16 proof_of_ownership_client_size = 2;
result = UDSApi::SvcAuthenticationVPOWNU_2020(cantp_handle::PCANTP_HANDLE_USBBUS1, config,
    request, algorithm_indicator, proof_of_ownership_client,
    proof_of_ownership_client_size);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(cantp_handle::PCANTP_HANDLE_USBBUS1, request,
        response, request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

```

```

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical Authentication/verifyProofOfOwnershipUnidirectional message
Dim algorithm_indicator As Byte() = {&H0, &H1, &H2, &H3, &H4, &H5, &H6, &H7, &H8, &H9, &HA,
    &HB, &HC, &HD, &HE, &HF}
Dim proof_of_ownership_client As Byte() = {&H12, &H34}
Dim proof_of_ownership_client_size As UInt16 = 2
result = UDSApi.SvcAuthenticationVPOWNU_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, algorithm_indicator, proof_of_ownership_client, proof_of_ownership_client_size)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    algorithm_indicator: array [0 .. 15] of Byte;
    proof_of_ownership_client: array [0 .. 1] of Byte;
    proof_of_ownership_client_size: UInt16;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
    config.nai.extension_addr := $0;
    config.nai.protocol :=
        uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
    config.nai.source_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
    config.nai.target_addr :=
        UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
    config.nai.target_type :=

```

```

cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/verifyProofOfOwnershipUnidirectional message
algorithm_indicator[0] := $00;
algorithm_indicator[1] := $01;
algorithm_indicator[2] := $02;
algorithm_indicator[3] := $03;
algorithm_indicator[4] := $04;
algorithm_indicator[5] := $05;
algorithm_indicator[6] := $06;
algorithm_indicator[7] := $07;
algorithm_indicator[8] := $08;
algorithm_indicator[9] := $09;
algorithm_indicator[10] := $0A;
algorithm_indicator[11] := $0B;
algorithm_indicator[12] := $0C;
algorithm_indicator[13] := $0D;
algorithm_indicator[14] := $0E;
algorithm_indicator[15] := $0F;
proof_of_ownership_client[0] := $12;
proof_of_ownership_client[1] := $34;
proof_of_ownership_client_size := 2;
result := TUDSApi.SvcAuthenticationVPOWNU_2020
  (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
   PByte(@algorithm_indicator), PByte(@proof_of_ownership_client),
   proof_of_ownership_client_size);
if (TUDSApi.StatusIsOk_2013(result)) then
begin
  result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
    @request, response, @request_confirmation);
end;
if (TUDSApi.StatusIsOk_2013(result)) then
begin
  MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
  // An error occurred
  MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```



See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcAuthenticationVPOWNU_2020](#) on page 758.

3.7.129 SvcAuthenticationVPOWNB_2020

Writes a UDS request according to the Authentication service's specifications (ISO-14229-1:2020). The subfunction verifyProofOfOwnershipBidirectional is implicit.

Overloads

| | Method | Description |
|---|--|---|
|  | SvcAuthenticationVPOWNB_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], byte[], UInt16, byte[], UInt16, byte[], UInt16) | Writes to the transmit queue a request for UDS service Authentication with verifyProofOfOwnershipBidirectional subfunction (ISO-14229-1:2020). |
|  | SvcAuthenticationVPOWNB_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], byte[], UInt16, byte[], UInt16) | Writes to the transmit queue a request for UDS service Authentication with verifyProofOfOwnershipBidirectional subfunction, without additional parameters (ISO-14229-1:2020). |

Plain function version: `UDS_SvcAuthenticationVPOWNB_2020` on page 760.

3.7.130 SvcAuthenticationVPOWNB_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], byte[], UInt16, byte[], UInt16, byte[], UInt16)

Writes a UDS request according to the Authentication service's specifications (ISO-14229-1:2020). The subfunction verifyProofOfOwnershipBidirectional is implicit.

Syntax

Pascal OO

```
class function SvcAuthenticationVPOWNB_2020(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    algorithm_indicator: PByte;
    proof_of_ownership_client: PByte;
    proof_of_ownership_client_size: UInt16;
    challenge_client: PByte;
    challenge_client_size: UInt16;
    additional_parameter: PByte;
    additional_parameter_size: UInt16
): uds_status; overload;
```

C#

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcAuthenticationVPOWNB_2020")]
public static extern uds_status SvcAuthenticationVPOWNB_2020(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 4)]
    byte[] algorithm_indicator,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 5)]
    byte[] proof_of_ownership_client,
    UInt16 proof_of_ownership_client_size,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 7)]
    byte[] challenge_client,
    UInt16 challenge_client_size,
    [MarshalAs(UnmanagedType.LPArray, SizeParamIndex = 9)]
    byte[] additional_parameter,
    UInt16 additional_parameter_size);
```

C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcAuthenticationVPOWNB_2020")]
static uds_status SvcAuthenticationVPOWNB_2020(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 4)]
    array<Byte> ^algorithm_indicator,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 5)]
    array<Byte> ^proof_of_ownership_client,
    UInt16 proof_of_ownership_client_size,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 7)]
    array<Byte> ^challenge_client,
    UInt16 challenge_client_size,
    [MarshalAs(UnmanagedType::LPArray, SizeParamIndex = 9)]
    array<Byte> ^additional_parameter,
    UInt16 additional_parameter_size);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcAuthenticationVPOWNB_2020")>
Public Shared Function SvcAuthenticationVPOWNB_2020(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=4)>
    ByVal algorithm_indicator As Byte(),
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=5)>
    ByVal proof_of_ownership_client As Byte(),
    ByVal proof_of_ownership_client_size As UInt16,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=7)>
    ByVal challenge_client As Byte(),
    ByVal challenge_client_size As UInt16,
    <MarshalAs(UnmanagedType.LPArray, SizeParamIndex:=9)>
    ByVal additional_parameter As Byte(),
    ByVal additional_parameter_size As UInt16) As uds_status
End Function
```

Parameters

| Parameter | Description |
|--------------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| algorithm_indicator | Buffer of 16 bytes containing the algorithm indicator. |
| proof_of_ownership_client | Buffer containing the proof of ownership of the client. |
| proof_of_ownership_client_size | Size in bytes of the proof of ownership buffer. |
| challenge_client | Buffer containing the challenge of the client. |
| challenge_client_size | Size in bytes of the challenge buffer. |
| additional_parameter | Buffer containing additional parameters. |
| additional_parameter | Size in bytes of the additional parameter buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>Reset_2013</code> (see <code>Reset_2013</code> on page 234). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>MsgFree_2013</code> on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

The PCAN-UDS 2.x API provides `uds_svc_authentication_subfunction` (see on page 98) and `uds_svc_authentication_return_parameter` (see on page 99) enumerations to help user to decode Authentication service responses.

Example

The following example shows the use of the service method `SvcAuthenticationVPOWNB_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], byte[], UInt16, byte[], UInt16, byte[], UInt16)` on the channel `PCANTP_HANDLE_USBUSB1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/verifyProofOfOwnershipBidirectional message
byte[] algorithm_indicator = new byte[16] { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F };
byte[] proof_of_ownership_client = new byte[2] { 0x12, 0x34 };
UInt16 proof_of_ownership_client_size = 2;
byte[] challenge_client = new byte[2] { 0x56, 0x78 };
UInt16 challenge_client_size = 2;
```

```

byte[] additional_parameter = new byte[2] { 0x9A, 0xBC };
UInt16 additional_parameter_size = 2;
result = UDSApi.SvcAuthenticationVPOWNB_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, algorithm_indicator, proof_of_ownership_client, proof_of_ownership_client_size,
    challenge_client, challenge_client_size, additional_parameter, additional_parameter_size);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/verifyProofOfOwnershipBidirectional message
array<Byte>^ algorithm_indicator = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
    0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F };
array<Byte>^ proof_of_ownership_client = { 0x12, 0x34 };
UInt16 proof_of_ownership_client_size = 2;
array<Byte>^ challenge_client = { 0x56, 0x78 };
UInt16 challenge_client_size = 2;
array<Byte>^ additional_parameter = { 0x9A, 0xBC };
UInt16 additional_parameter_size = 2;
result = UDSApi::SvcAuthenticationVPOWNB_2020(cantp_handle::PCANTP_HANDLE_USBBUS1, config,
    request, algorithm_indicator, proof_of_ownership_client, proof_of_ownership_client_size,
    challenge_client, challenge_client_size, additional_parameter,
    additional_parameter_size);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(cantp_handle::PCANTP_HANDLE_USBBUS1, request,
        response, request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures

```



```
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);
```

Visual Basic

```
Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical Authentication/verifyProofOfOwnershipBidirectional message
Dim algorithm_indicator As Byte() = {&H0, &H1, &H2, &H3, &H4, &H5, &H6, &H7, &H8, &H9, &HA,
    &HB, &HC, &HD, &HE, &HF}
Dim proof_of_ownership_client As Byte() = {&H12, &H34}
Dim proof_of_ownership_client_size As UInt16 = 2
Dim challenge_client As Byte() = {&H56, &H78}
Dim challenge_client_size As UInt16 = 2
Dim additional_parameter As Byte() = {&H9A, &HBC}
Dim additional_parameter_size As UInt16 = 2
result = UDSApi.SvcAuthenticationVPOWNB_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, algorithm_indicator, proof_of_ownership_client, proof_of_ownership_client_size,
    challenge_client, challenge_client_size, additional_parameter, additional_parameter_size)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)
```

Pascal OO

```
var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
    algorithm_indicator: array [0 .. 15] of Byte;
    proof_of_ownership_client: array [0 .. 1] of Byte;
    proof_of_ownership_client_size: UInt16;
    challenge_client: array [0 .. 1] of Byte;
```



```

challenge_client_size: UInt16;
additional_parameter: array [0 .. 1] of Byte;
additional_parameter_size: UInt16;
begin
  FillChar(request, sizeof(request), 0);
  FillChar(request_confirmation, sizeof(request_confirmation), 0);
  FillChar(response, sizeof(response), 0);

  // Set request message configuration
  FillChar(config, sizeof(config), 0);
  config.can_id :=
    UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
  config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
  config.nai.extension_addr := $0;
  config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
  config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
  config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
  config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
  config.typem := uds_msgtype.PUDS_MSGTYPE_USDT;

  // Sends a physical Authentication/verifyProofOfOwnershipBidirectional message
  algorithm_indicator[0] := $00;
  algorithm_indicator[1] := $01;
  algorithm_indicator[2] := $02;
  algorithm_indicator[3] := $03;
  algorithm_indicator[4] := $04;
  algorithm_indicator[5] := $05;
  algorithm_indicator[6] := $06;
  algorithm_indicator[7] := $07;
  algorithm_indicator[8] := $08;
  algorithm_indicator[9] := $09;
  algorithm_indicator[10] := $0A;
  algorithm_indicator[11] := $0B;
  algorithm_indicator[12] := $0C;
  algorithm_indicator[13] := $0D;
  algorithm_indicator[14] := $0E;
  algorithm_indicator[15] := $0F;
  proof_of_ownership_client[0] := $12;
  proof_of_ownership_client[1] := $34;
  proof_of_ownership_client_size := 2;
  challenge_client[0] := $56;
  challenge_client[1] := $78;
  challenge_client_size := 2;
  additional_parameter[0] := $9A;
  additional_parameter[1] := $BC;
  additional_parameter_size := 2;
  result := TUDSApi.SvcAuthenticationVPOWNB_2020
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
     PByte(@algorithm_indicator), PByte(@proof_of_ownership_client),
     proof_of_ownership_client_size, PByte(@challenge_client),
     challenge_client_size, PByte(@additional_parameter),
     additional_parameter_size);
  if (TUDSApi.StatusIsOk_2013(result)) then
  begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
      @request, response, @request_confirmation);
  end;
  if (TUDSApi.StatusIsOk_2013(result)) then

```

```

begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcAuthenticationVPOWNB_2020](#) on page 760.

3.7.131 SvcAuthenticationVPOWNB_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], byte[], UInt16, byte[], UInt16)

Writes a UDS request according to the Authentication service's specifications with the subfunction verifyProofOfOwnershipBidirectional, without additional parameters (ISO-14229-1:2020).

Syntax

Pascal OO

```

class function SvcAuthenticationVPOWNB_2020(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg;
    algorithm_indicator: PByte;
    proof_of_ownership_client: PByte;
    proof_of_ownership_client_size: UInt16;
    challenge_client: PByte;
    challenge_client_size: UInt16
): uds_status; overload;

```

C#

```

public static uds_status SvcAuthenticationVPOWNB_2020(
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request,
    byte[] algorithm_indicator,
    byte[] proof_of_ownership_client,
    UInt16 proof_of_ownership_client_size,
    byte[] challenge_client,
    UInt16 challenge_client_size);

```

C++ / CLR

```

static uds_status SvcAuthenticationVPOWNB_2020(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request,
    array<Byte> ^algorithm_indicator,
    array<Byte> ^proof_of_ownership_client,

```

```
UInt16 proof_of_ownership_client_size,
array<Byte> ^challenge_client,
UInt16 challenge_client_size);
```

Visual Basic

```
Public Shared Function SvcAuthenticationVPOWNB_2020(
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg,
    ByVal algorithm_indicator As Byte(),
    ByVal proof_of_ownership_client As Byte(),
    ByVal proof_of_ownership_client_size As UInt16,
    ByVal challenge_client As Byte(),
    ByVal challenge_client_size As UInt16) As uds_status
End Function
```

Parameters

| Parameter | Description |
|--------------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |
| algorithm_indicator | Buffer of 16 bytes containing the algorithm indicator. |
| proof_of_ownership_client | Buffer containing the proof of ownership of the client. |
| proof_of_ownership_client_size | Size in bytes of the proof of ownership buffer. |
| challenge_client | Buffer containing the challenge of the client. |
| challenge_client_size | Size in bytes of the challenge buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is not valid. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | The mapping given in the message configuration is unknown. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with Reset_2013 (see Reset_2013 on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see MsgFree_2013 on page 214). |

Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

The PCAN-UDS 2.x API provides `uds_svc_authentication_subfunction` (see on page 98) and `uds_svc_authentication_return_parameter` (see on page 99) enumerations to help user to decode Authentication service responses.

Example

The following example shows the use of the service method `SvcAuthenticationVPOWNB_2020(cantp_handle, uds_msgconfig, uds_msg, byte[], byte[], UInt16, byte[], UInt16)` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical

service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

C#

```
uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/verifyProofOfOwnershipBidirectional message
byte[] algorithm_indicator = new byte[16] { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F };
byte[] proof_of_ownership_client = new byte[2] { 0x12, 0x34 };
UInt16 proof_of_ownership_client_size = 2;
byte[] challenge_client = new byte[2] { 0x56, 0x78 };
UInt16 challenge_client_size = 2;
result = UDSApi.SvcAuthenticationVPOWNB_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request, algorithm_indicator, proof_of_ownership_client, proof_of_ownership_client_size,
    challenge_client, challenge_client_size);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);
```

C++/CLR

```
uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
```

```

config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/verifyProofOfOwnershipBidirectional message
array<Byte>^ algorithm_indicator = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
    0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F };
array<Byte>^ proof_of_ownership_client = { 0x12, 0x34 };
UInt16 proof_of_ownership_client_size = 2;
array<Byte>^ challenge_client = { 0x56, 0x78 };
UInt16 challenge_client_size = 2;
result = UDSApi::SvcAuthenticationVPOWNB_2020(cantp_handle::PCANTP_HANDLE_USBBUS1, config,
    request, algorithm_indicator, proof_of_ownership_client, proof_of_ownership_client_size,
    challenge_client, challenge_client_size);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(cantp_handle::PCANTP_HANDLE_USBBUS1, request,
        response, request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else
    // An error occurred
    MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical Authentication/verifyProofOfOwnershipBidirectional message
Dim algorithm_indicator As Byte() = {&H0, &H1, &H2, &H3, &H4, &H5, &H6, &H7, &H8, &H9, &HA,
    &HB, &HC, &HD, &HE, &HF}
Dim proof_of_ownership_client As Byte() = {&H12, &H34}
Dim proof_of_ownership_client_size As UInt16 = 2
Dim challenge_client As Byte() = {&H56, &H78}
Dim challenge_client_size As UInt16 = 2
result = UDSApi.SvcAuthenticationVPOWNB_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config,
    request, algorithm_indicator, proof_of_ownership_client, proof_of_ownership_client_size,
    challenge_client, challenge_client_size)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else

```

```

    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
  result: uds_status;
  request: uds_msg;
  request_confirmation: uds_msg;
  response: uds_msg;
  config: uds_msgconfig;
  algorithm_indicator: array [0 .. 15] of Byte;
  proof_of_ownership_client: array [0 .. 1] of Byte;
  proof_of_ownership_client_size: UInt16;
  challenge_client: array [0 .. 1] of Byte;
  challenge_client_size: UInt16;
begin
  FillChar(request, sizeof(request), 0);
  FillChar(request_confirmation, sizeof(request_confirmation), 0);
  FillChar(response, sizeof(response), 0);

  // Set request message configuration
  FillChar(config, sizeof(config), 0);
  config.can_id :=
    UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
  config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
  config.nai.extension_addr := $0;
  config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
  config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
  config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
  config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
  config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

  // Sends a physical Authentication/verifyProofOfOwnershipBidirectional message
  algorithm_indicator[0] := $00;
  algorithm_indicator[1] := $01;
  algorithm_indicator[2] := $02;
  algorithm_indicator[3] := $03;
  algorithm_indicator[4] := $04;
  algorithm_indicator[5] := $05;
  algorithm_indicator[6] := $06;
  algorithm_indicator[7] := $07;
  algorithm_indicator[8] := $08;
  algorithm_indicator[9] := $09;
  algorithm_indicator[10] := $0A;
  algorithm_indicator[11] := $0B;
  algorithm_indicator[12] := $0C;
  algorithm_indicator[13] := $0D;
  algorithm_indicator[14] := $0E;
  algorithm_indicator[15] := $0F;
  proof_of_ownership_client[0] := $12;
  proof_of_ownership_client[1] := $34;

```

```

proof_of_ownership_client_size := 2;
challenge_client[0] := $56;
challenge_client[1] := $78;
challenge_client_size := 2;
result := TUDSApi.SvcAuthenticationVPOWNB_2020
    (cantp_handle.PCANTP_HANDLE_USBBUS1, config, request,
    PByte(@algorithm_indicator), PByte(@proof_of_ownership_client),
    proof_of_ownership_client_size, PByte(@challenge_client),
    challenge_client_size);
if (TUDSApi.StatusIsOk_2013(result)) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
end;
if (TUDSApi.StatusIsOk_2013(result)) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK);
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcAuthenticationVPOWNB_2020](#) on page 760.

3.7.132 SvcAuthenticationAC_2020

Writes a UDS request according to the Authentication service's specifications (ISO-14229-1:2020). The subfunction authenticationConfiguration is implicit.

Syntax

Pascal OO

```

class function SvcAuthenticationAC_2020(
    channel: cantp_handle;
    request_config: uds_msgconfig;
    var out_msg_request: uds_msg
): uds_status;

```

C#

```

[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcAuthenticationAC_2020")]
public static extern uds_status SvcAuthenticationAC_2020(
    [MarshalAs(UnmanagedType.U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    out uds_msg out_msg_request);

```


C++ / CLR

```
[DllImport("PCAN-UDS.dll", EntryPoint = "UDS_SvcAuthenticationAC_2020")]
static uds_status SvcAuthenticationAC_2020(
    [MarshalAs(UnmanagedType::U4)]
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg %out_msg_request);
```

Visual Basic

```
<DllImport("PCAN-UDS.dll", EntryPoint:="UDS_SvcAuthenticationAC_2020")>
Public Shared Function SvcAuthenticationAC_2020(
    <MarshalAs(UnmanagedType.U4)>
    ByVal channel As cantp_handle,
    ByVal request_config As uds_msgconfig,
    ByRef out_msg_request As uds_msg) As uds_status
End Function
```

Parameters

| Parameter | Description |
|-----------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the method (see uds_msg on page 21). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with Reset_2013 (see Reset_2013 on page 234). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see MsgFree_2013 on page 214). |


Remarks

This method creates a new message using a given message configuration and set the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `MsgFree_2013` on page 214).

The PCAN-UDS 2.x API provides `uds_svc_authentication_subfunction` (see on page 98) and `uds_svc_authentication_return_parameter` (see on page 99) enumerations to help user to decode Authentication service responses.

Example

The following example shows the use of the service method `SvcAuthenticationAC_2020` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `WaitForService_2013` method is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C#

```

uds_status result;
uds_msg request = new uds_msg();
uds_msg request_confirmation = new uds_msg();
uds_msg response = new uds_msg();
uds_msgconfig config = new uds_msgconfig();

// Set request message configuration
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/authenticationConfiguration message
result = UDSApi.SvcAuthenticationAC_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config, out
    request);
if (UDSApi.StatusIsOk_2013(result))
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, ref request, out
        response, out request_confirmation);
if (UDSApi.StatusIsOk_2013(result))
    MessageBox.Show("Response was received", "Success");
else
    // An error occurred
    MessageBox.Show("An error occurred", "Error");

// Free structures
UDSApi.MsgFree_2013(ref request);
UDSApi.MsgFree_2013(ref response);
UDSApi.MsgFree_2013(ref request_confirmation);

```

C++/CLR

```

uds_status result;
uds_msg request = {};
uds_msg request_confirmation = {};
uds_msg response = {};
uds_msgconfig config = {};

// Set request message configuration
config.can_id = PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/authenticationConfiguration message
result = UDSApi::SvcAuthenticationAC_2020(cantp_handle::PCANTP_HANDLE_USBBUS1, config,
    request);
if (UDSApi::StatusIsOk_2013(result))
    result = UDSApi::WaitForService_2013(cantp_handle::PCANTP_HANDLE_USBBUS1, request,
        response, request_confirmation);
if (UDSApi::StatusIsOk_2013(result))
    MessageBox::Show("Response was received", "Success");
else

```

```

        // An error occurred
        MessageBox::Show("An error occurred", "Error");

// Free structures
UDSApi::MsgFree_2013(request);
UDSApi::MsgFree_2013(response);
UDSApi::MsgFree_2013(request_confirmation);

```

Visual Basic

```

Dim result As uds_status
Dim request As uds_msg = New uds_msg()
Dim request_confirmation As uds_msg = New uds_msg()
Dim response As uds_msg = New uds_msg()
Dim config As uds_msgconfig = New uds_msgconfig()

' Set request message configuration
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.extension_addr = &H0
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT

' Sends a physical Authentication/authenticationConfiguration message
result = UDSApi.SvcAuthenticationAC_2020(cantp_handle.PCANTP_HANDLE_USBBUS1, config, request)
If UDSApi.StatusIsOk_2013(result) Then
    result = UDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1, request, response,
        request_confirmation)
End If
If UDSApi.StatusIsOk_2013(result) Then
    MessageBox.Show("Response was received", "Success")
Else
    ' An error occurred
    MessageBox.Show("An error occurred", "Error")
End If

' Free structures
UDSApi.MsgFree_2013(request)
UDSApi.MsgFree_2013(response)
UDSApi.MsgFree_2013(request_confirmation)

```

Pascal OO

```

var
    result: uds_status;
    request: uds_msg;
    request_confirmation: uds_msg;
    response: uds_msg;
    config: uds_msgconfig;
begin
    FillChar(request, sizeof(request), 0);
    FillChar(request_confirmation, sizeof(request_confirmation), 0);
    FillChar(response, sizeof(response), 0);

    // Set request message configuration
    FillChar(config, sizeof(config), 0);
    config.can_id :=
        UInt32(uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1);
    config.can_msgtype := cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;

```

```

config.nai.extension_addr := $0;
config.nai.protocol :=
    uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT);
config.nai.target_addr :=
    UInt16(uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1);
config.nai.target_type :=
    cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.typeem := uds_msgtype.PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/authenticationConfiguration message
result := TUDSApi.SvcAuthenticationAC_2020(cantp_handle.PCANTP_HANDLE_USBBUS1,
    config, request);
if (TUDSApi.StatusIsOk_2013(result)) then
begin
    result := TUDSApi.WaitForService_2013(cantp_handle.PCANTP_HANDLE_USBBUS1,
        @request, response, @request_confirmation);
end;
if (TUDSApi.StatusIsOk_2013(result)) then
begin
    MessageBox(0, 'Response was received', 'Success', MB_OK)
end
else
begin
    // An error occurred
    MessageBox(0, 'An error occurred', 'Error', MB_OK);
end;

// Free structures
TUDSApi.MsgFree_2013(request);
TUDSApi.MsgFree_2013(response);
TUDSApi.MsgFree_2013(request_confirmation);
end;

```

See also: [WaitForService_2013](#) on page 248.

Plain function version: [UDS_SvcAuthenticationAC_2020](#) on page 763.

3.7.133 GetDataServiceId_2013

This method is a C# and VB helper: it gets PUDS message data service identifier in a safe way.

Syntax

C#

```

public static bool GetDataServiceId_2013(
    ref uds_msg msg,
    out Byte val);

```

Visual Basic

```

Public Shared Function GetDataServiceId_2013(
    ByRef msg As uds_msg,
    ByRef val As Byte) As Boolean
End Function

```

Parameters

| Parameter | Description |
|-----------|--|
| msg | UDS message containing the service identifier (see <code>uds_msg</code> on page 21). |
| val | Output, service identifier value. |

Returns

True if ok, false if not ok.

Example

The following example shows the use of the method `GetDataServiceId_2013`. It allocates a message, sets a service identifier, gets it, and prints it.

C#

```
uds_msgconfig config = new uds_msgconfig();
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.extension_addr = 0;

uds_msg request = new uds_msg();
uds_status result;
result = UDSApi.MsgAlloc_2013(out request, config, 1);
if (UDSApi.StatusIsOk_2013(result))
{
    UDSApi.SetDataServiceId_2013(ref request,
        (byte)uds_service.PUDS_SERVICE_SI_ClearDiagnosticInformation);
    Byte service_id;
    if (UDSApi.GetDataServiceId_2013(ref request, out service_id))
    {
        MessageBox.Show("Set request service id = " + service_id, "Success");
    }
    else
    {
        MessageBox.Show("An error occurred", "Error");
    }
}
```

Visual Basic

```
Dim config As uds_msgconfig = New uds_msgconfig()
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.extension_addr = 0

Dim request As uds_msg = New uds_msg()
Dim result As uds_status
result = UDSApi.MsgAlloc_2013(request, config, 1)
If UDSApi.StatusIsOk_2013(result) Then
    UDSApi.SetDataServiceId_2013(request,
```

```

uds_service.PUDS_SERVICE_SI_ClearDiagnosticInformation)
Dim service_id As Byte
If UDSApi.GetDataServiceId_2013(request, service_id) Then
    MessageBox.Show("Set request service id = " + service_id.ToString(), "Success")
Else
    MessageBox.Show("An error occurred", "Error")
End If
End If
End If

```

See also: [uds_msg](#) on page 21.

3.7.134 SetDataServiceId_2013

This method is a C# and VB helper: it sets PUDS message data service id, in a safe way.

Syntax

C#

```

public static bool SetDataServiceId_2013(
    ref uds_msg msg,
    Byte val);

```

Visual Basic

```

Public Shared Function SetDataServiceId_2013(
    ByRef msg As uds_msg,
    ByVal val As Byte) As Boolean
End Function

```

Parameters

| Parameter | Description |
|-----------|---|
| msg | UDS message containing the service identifier (see uds_msg on page 21). |
| val | New service identifier value. |

Returns

True if ok, false if not ok.

Example

The following example shows the use of the method [SetDataServiceId_2013](#). It allocates a message, sets a service identifier, gets it, and prints it.

C#

```

uds_msgconfig config = new uds_msgconfig();
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.extension_addr = 0;

uds_msg request = new uds_msg();
uds_status result;
result = UDSApi.MsgAlloc_2013(out request, config, 1);
if (UDSApi.StatusIsOk_2013(result))
{

```

```

UDSApi.SetDataServiceId_2013(ref request,
    (byte)uds_service.PUDS_SERVICE_SI_ClearDiagnosticInformation);
Byte service_id;
if (UDSApi.GetDataServiceId_2013(ref request, out service_id))
{
    MessageBox.Show("Set request service id = " + service_id, "Success");
}
else
{
    MessageBox.Show("An error occurred", "Error");
}
}

```

Visual Basic

```

Dim config As uds_msgconfig = New uds_msgconfig()
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_REQUEST_1
config.can_msgtype = cantp_can_msgtype.PCAN_TP_CAN_MSGTYPE_STANDARD
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.target_type = cantp_isotp_addressing.PCAN_TP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.extension_addr = 0

Dim request As uds_msg = New uds_msg()
Dim result As uds_status
result = UDSApi.MsgAlloc_2013(request, config, 1)
If UDSApi.StatusIsOk_2013(result) Then
    UDSApi.SetDataServiceId_2013(request,
        uds_service.PUDS_SERVICE_SI_ClearDiagnosticInformation)
    Dim service_id As Byte
    If UDSApi.GetDataServiceId_2013(request, service_id) Then
        MessageBox.Show("Set request service id = " + service_id.ToString(), "Success")
    Else
        MessageBox.Show("An error occurred", "Error")
    End If
End If

```

See also: `uds_msg` on page 21.

3.7.135 GetDataNrc_2013

This method is a C# and VB helper: it gets PUDS message data negative response code (nrc) in a safe way.

Syntax

C#

```

public static bool GetDataNrc_2013(
    ref uds_msg msg,
    out Byte val);

```

Visual Basic

```

Public Shared Function GetDataNrc_2013(
    ByRef msg As uds_msg,
    ByRef val As Byte) As Boolean
End Function

```

Parameters

| Parameter | Description |
|-----------|--|
| msg | UDS message containing the nrc (see uds_msg on page 21). |
| val | Output, nrc value. |

Returns

True if ok, false if not ok.

Example

The following example shows the use of the method `GetDataNrc_2013`. It allocates a message, a nrc pointer, sets a new NRC value, then gets, and prints this value.

C#

```
uds_msgconfig config = new uds_msgconfig();
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.extension_addr = 0;

uds_msg response = new uds_msg();
uds_status result = UDSApi.MsgAlloc_2013(out response, config, 2);
if (UDSApi.StatusIsOk_2013(result))
{
    // Set nrc pointer in message structure
    response.links.nrc = IntPtr.Add(response.links.param, 1);

    UDSApi.SetDataNrc_2013(ref response, UDSApi.PUDS_NRC_EXTENDED_TIMING);
    Byte nrc;
    if (UDSApi.GetDataNrc_2013(ref response, out nrc))
    {
        MessageBox.Show("Set NRC extended timing value = " + nrc, "Success");
    }
    else
    {
        MessageBox.Show("An error occurred", "Error");
    }
}
```

Visual Basic

```
Dim config As uds_msgconfig = New uds_msgconfig()
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.extension_addr = 0

Dim response As uds_msg = New uds_msg()
Dim result As uds_status
result = UDSApi.MsgAlloc_2013(response, config, 2)
If UDSApi.StatusIsOk_2013(result) Then
```

```

' Set nrc pointer in message structure
response.links.nrc = IntPtr.Add(response.links.param, 1)

UDSApi.SetDataNrc_2013(response, UDSApi.PUDS_NRC_EXTENDED_TIMING)
Dim nrc As Byte
If UDSApi.GetDataNrc_2013(response, nrc) Then
    MessageBox.Show("Set NRC extended timing value = " + nrc.ToString(), "Success")
Else
    MessageBox.Show("An error occurred", "Error")
End If
End If

```

See also: `uds_msg` on page 21.

3.7.136 SetDataNrc_2013

This method is a C# and VB helper: it sets PUDS message data negative response code (nrc) in a safe way.

Syntax

C#

```

public static bool SetDataNrc_2013(
    ref uds_msg msg,
    Byte val);

```

Visual Basic

```

Public Shared Function SetDataNrc_2013(
    ByRef msg As uds_msg,
    ByVal val As Byte) As Boolean
End Function

```

Parameters

| Parameter | Description |
|-----------|---|
| msg | UDS message containing the nrc (see <code>uds_msg</code> on page 21). |
| val | New nrc value. |

Returns

True if ok, false if not ok.

Example

The following example shows the use of the method `SetDataNrc_2013`. It allocates a message, a nrc pointer, sets a new NRC value, then gets, and prints this value. Depending on the result, a message will be shown to the user.

C#

```

uds_msgconfig config = new uds_msgconfig();
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.extension_addr = 0;

```



```

uds_msg response = new uds_msg();
uds_status result = UDSApi.MsgAlloc_2013(out response, config, 2);
if (UDSApi.StatusIsOk_2013(result))
{
    // Set nrc pointer in message structure
    response.links.nrc = IntPtr.Add(response.links.param, 1);

    UDSApi.SetDataNrc_2013(ref response, UDSApi.PUDS_NRC_EXTENDED_TIMING);
    Byte nrc;
    if (UDSApi.GetDataNrc_2013(ref response, out nrc))
    {
        MessageBox.Show("Set NRC extended timing value = " + nrc, "Success");
    }
    else
    {
        MessageBox.Show("An error occurred", "Error");
    }
}
}

```

Visual Basic

```

Dim config As uds_msgconfig = New uds_msgconfig()
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.extension_addr = 0

Dim response As uds_msg = New uds_msg()
Dim result As uds_status
result = UDSApi.MsgAlloc_2013(response, config, 2)
If UDSApi.StatusIsOk_2013(result) Then

    ' Set nrc pointer in message structure
    response.links.nrc = IntPtr.Add(response.links.param, 1)

    UDSApi.SetDataNrc_2013(response, UDSApi.PUDS_NRC_EXTENDED_TIMING)
    Dim nrc As Byte
    If UDSApi.GetDataNrc_2013(response, nrc) Then
        MessageBox.Show("Set NRC extended timing value = " + nrc.ToString(), "Success")
    Else
        MessageBox.Show("An error occurred", "Error")
    End If
End If

```

See also: [uds_msg](#) on page 21.

3.7.137 GetDataParameter_2013

This method is a C# and VB helper: it gets PUDS message data parameter in a safe way.

Syntax

C#

```
public static bool GetDataParameter_2013(
    ref uds_msg msg,
    int nump,
    out Byte val);
```

Visual Basic

```
Public Shared Function GetDataParameter_2013(
    ByRef msg As uds_msg,
    ByVal nump As Integer,
    ByRef val As Byte) As Boolean
End Function
```

Parameters

| Parameter | Description |
|-----------|---|
| msg | UDS message containing the data parameter (see uds_msg on page 21). |
| nump | Data parameter number (index start from 0). |
| val | Output, data parameter value. |

Returns

True if ok, false if not ok.

Example

The following example shows the use of the method `GetDataParameter_2013`. It sets a new data parameter, gets it, and prints it.

C#

```
uds_msgconfig config = new uds_msgconfig();
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.extension_addr = 0;

uds_msg response = new uds_msg();
uds_status result = UDSApi.MsgAlloc_2013(out response, config, 2);
if (UDSApi.StatusIsOk_2013(result))
{
    UDSApi.SetDataParameter_2013(ref response, 0, 42);
    Byte param;
    if (UDSApi.GetDataParameter_2013(ref response, 0, out param))
    {
        MessageBox.Show("Set param[0] = " + param, "Success");
    }
    else
    {
        MessageBox.Show("An error occurred", "Error");
    }
}
```

Visual Basic

```

Dim config As uds_msgconfig = New uds_msgconfig()
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.extension_addr = 0

Dim response As uds_msg = New uds_msg()
Dim result As uds_status
result = UDSApi.MsgAlloc_2013(response, config, 2)
If UDSApi.StatusIsOk_2013(result) Then

    UDSApi.SetDataParameter_2013(response, 0, 42)
    Dim param As Byte
    If UDSApi.GetDataParameter_2013(response, 0, param) Then
        MessageBox.Show("Set param[0] = " + param.ToString(), "Success")
    Else
        MessageBox.Show("An error occurred", "Error")
    End If
End If

```

See also: `uds_msg` on page 21.

3.7.138 SetDataParameter_2013

This method is a C# and VB helper: it sets PUDS message data parameter in a safe way.

Syntax

C#

```

public static bool SetDataParameter_2013(
    ref uds_msg msg,
    int nump,
    Byte val);

```

Visual Basic

```

Public Shared Function SetDataParameter_2013(
    ByRef msg As uds_msg,
    ByVal nump As Integer,
    ByVal val As Byte) As Boolean
End Function

```

Parameters

| Parameter | Description |
|-----------|---|
| msg | PUDS message containing the data parameter (see <code>uds_msg</code> on page 21). |
| nump | Data parameter number (index start from 0). |
| val | New data parameter value. |

Returns

True if ok, false if not ok.

Example

The following example shows the use of the method `SetDataParameter_2013`. It sets a new data parameter, gets it, and prints it.

C#

```
uds_msgconfig config = new uds_msgconfig();
config.can_id = (UInt16)uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_1;
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = uds_msgtype.PUDS_MSGTYPE_USDT;
config.nai.source_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_addr = (UInt16)uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.extension_addr = 0;

uds_msg response = new uds_msg();
uds_status result = UDSApi.MsgAlloc_2013(out response, config, 2);
if (UDSApi.StatusIsOk_2013(result))
{
    UDSApi.SetDataParameter_2013(ref response, 0, 42);
    Byte param;
    if (UDSApi.GetDataParameter_2013(ref response, 0, out param))
    {
        MessageBox.Show("Set param[0] = " + param, "Success");
    }
    else
    {
        MessageBox.Show("An error occurred", "Error");
    }
}
```

Visual Basic

```
Dim config As uds_msgconfig = New uds_msgconfig()
config.can_id = uds_can_id.PUDS_CAN_ID_ISO_15765_4_PHYSICAL_RESPONSE_1
config.can_msgtype = cantp_can_msgtype.PCANTP_CAN_MSGTYPE_STANDARD
config.nai.protocol = uds_msgprotocol.PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL
config.nai.target_type = cantp_isotp_addressing.PCANTP_ISOTP_ADDRESSING_PHYSICAL
config.type = uds_msgtype.PUDS_MSGTYPE_USDT
config.nai.source_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_ECU_1
config.nai.target_addr = uds_address.PUDS_ADDRESS_ISO_15765_4_ADDR_TEST_EQUIPMENT
config.nai.extension_addr = 0

Dim response As uds_msg = New uds_msg()
Dim result As uds_status
result = UDSApi.MsgAlloc_2013(response, config, 2)
If UDSApi.StatusIsOk_2013(result) Then




    UDSApi.SetDataParameter_2013(response, 0, 42)
    Dim param As Byte
    If UDSApi.GetDataParameter_2013(response, 0, param) Then
        MessageBox.Show("Set param[0] = " + param.ToString(), "Success")
    Else
        MessageBox.Show("An error occurred", "Error")
    End If
End If
```

See also: `uds_msg` on page 21.







3.8 Functions

The functions of the PCAN-UDS 2.x API are divided into 5 groups of functionalities.








Connection

| | Function | Description |
|---|-----------------------|---|
|  | UDS_Initialize_2013 | Initializes a PUDS channel based on a PCANTP Channel handle (without CAN FD support). |
|  | UDS_InitializeFD_2013 | Initializes a PUDS channel based on a PCANTP Channel handle (including CAN FD support). |
|  | UDS_Uninitialize_2013 | Uninitializes a PUDS channel |





Configuration

| | Function | Description |
|---|-------------------------------|---|
|  | UDS_SetValue_2013 | Sets a configuration or information value within a PUDS channel. |
|  | UDS_AddMapping_2013 | Adds a user-defined mapping between a CAN identifier and a network address information. |
|  | UDS_RemoveMappingByCanId_2013 | Removes all user defined PUDS mappings corresponding to a CAN identifier. |
|  | UDS_RemoveMapping_2013 | Removes a user defined PUDS mapping. |
|  | UDS_AddCanIdFilter_2013 | Adds an entry to the CAN identifier white-list filtering. |
|  | UDS_RemoveCanIdFilter_2013 | Removes an entry from the CAN identifier white-list filtering. |
















Information

| | Function | Description |
|---|--------------------------------|---|
|  | UDS_GetValue_2013 | Retrieves information from a PUDS channel. |
|  | UDS_GetCanBusStatus_2013 | Gets information about the internal BUS status of a PUDS channel. |
|  | UDS_GetMapping_2013 | Retrieves a mapping matching the given CAN identifier and message type (11bits, 29 bits, FD, etc.). |
|  | UDS_GetMappings_2013 | Retrieves all the PUDS mappings defined for a PUDS channel. |
|  | UDS_GetSessionInformation_2013 | Gets current ECU session information. |
|  | UDS_StatusIsOk_2013 | Checks if a PUDS status matches an expected result (default is PUDS_STATUS_OK). |
|  | UDS_GetErrorText_2013 | Gets a descriptive text for a PUDS error code. |

















Message handling

| | Function | Description |
|---|-------------------|---|
|  | UDS_MsgAlloc_2013 | Allocates a PUDS message using the given configuration. |
|  | UDS_MsgFree_2013 | Deallocates a PUDS message. |
|  | UDS_MsgCopy_2013 | Copies a PUDS message to another buffer. |
|  | UDS_MsgMove_2013 | Moves a PUDS message to another buffer (and cleans the original message structure). |

Communication

| | Function | Description |
|---|---|--|
|  | UDS_Read_2013 | Reads a message from the receive queue of a PUDS channel. |
|  | UDS_Write_2013 | Transmits a message using a connected PUDS channel. |
|  | UDS_Reset_2013 | Resets the receive and transmit queues of a PUDS channel. |
|  | UDS_WaitForSingleMessage_2013 | Waits for a message (response or a transmit confirmation) based on a PUDS message request. |
|  | UDS_WaitForFunctionalResponses_2013 | Waits for multiple messages (multiple responses from a functional request for instance) based on a PUDS message request. |
|  | UDS_WaitForService_2013 | Handles the communication workflow for a UDS service expecting a single response. |
|  | UDS_WaitForServiceFunctional_2013 | Handles the communication workflow for a UDS service expecting multiple responses. |
|  | UDS_SvcDiagnosticSessionControl_2013 | Writes to the transmit queue a request for UDS service DiagnosticSessionControl. |
|  | UDS_SvcECUReset_2013 | Writes to the transmit queue a request for UDS service ECUReset. |
|  | UDS_SvcSecurityAccess_2013 | Writes to the transmit queue a request for UDS service SecurityAccess. |
|  | UDS_SvcCommunicationControl_2013 | Writes to the transmit queue a request for UDS service CommunicationControl. |
|  | UDS_SvcTesterPresent_2013 | Writes to the transmit queue a request for UDS service TesterPresent. |
|  | UDS_SvcSecuredDataTransmission_2013 | Writes to the transmit queue a request for UDS service SecuredDataTransmission(ISO-14229-1:2013). |
|  | UDS_SvcSecuredDataTransmission_2020 | Writes to the transmit queue a request for UDS service SecuredDataTransmission(ISO-14229-1:2020). |
|  | UDS_SvcControlDTCSetting_2013 | Writes to the transmit queue a request for UDS service ControlDTCSetting. |
|  | UDS_SvcResponseOnEvent_2013 | Writes to the transmit queue a request for UDS service ResponseOnEvent. |
|  | UDS_SvcLinkControl_2013 | Writes to the transmit queue a request for UDS service LinkControl. |
|  | UDS_SvcReadDataByIdentifier_2013 | Writes to the transmit queue a request for UDS service ReadDataByIdentifier. |
|  | UDS_SvcReadMemoryByAddress_2013 | Writes to the transmit queue a request for UDS service ReadMemoryByAddress. |
|  | UDS_SvcReadScalingDataByIdentifier_2013 | Writes to the transmit queue a request for UDS service ReadScalingDataByIdentifier. |
|  | UDS_SvcReadDataByPeriodicIdentifier_2013 | Writes to the transmit queue a request for UDS service ReadDataByPeriodicIdentifier. |
|  | UDS_SvcDynamicallyDefineDataIdentifierDBID_2013 | Writes to the transmit queue a request for UDS service DynamicallyDefineDataIdentifier. |

| | Function | Description |
|---|--|---|
|  | UDS_SvcDynamicallyDefineDataIdentifierDBMA_2013 | Writes to the transmit queue a request for UDS service DynamicallyDefineDataIdentifier. |
|  | UDS_SvcDynamicallyDefineDataIdentifierCDDDI_2013 | Writes to the transmit queue a request for UDS service DynamicallyDefineDataIdentifier. |
|  | UDS_SvcDynamicallyDefineDataIdentifierClearAllIDDDI_2013 | Writes to the transmit queue a request for UDS service DynamicallyDefineDataIdentifier/clearDynamicallyDefinedDataIdentifier subfunction. |
|  | UDS_SvcWriteDataByIdentifier_2013 | Writes to the transmit queue a request for UDS service WriteDataByIdentifier. |
|  | UDS_SvcWriteMemoryByAddress_2013 | Writes to the transmit queue a request for UDS service WriteMemoryByAddress. |
|  | UDS_SvcClearDiagnosticInformation_2013 | Writes to the transmit queue a request for UDS service ClearDiagnosticInformation. |
|  | UDS_SvcClearDiagnosticInformation_2020 | Writes to the transmit queue a request for UDS service ClearDiagnosticInformation with memory selection parameter (ISO-14229-1:2020). |
|  | UDS_SvcReadDTCInformation_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | UDS_SvcReadDTCInformationRDTCSBDTC_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | UDS_SvcReadDTCInformationRDTCSBRN_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | UDS_SvcReadDTCInformationReportExtended_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | UDS_SvcReadDTCInformationReportSeverity_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | UDS_SvcReadDTCInformationRSIODTC_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | UDS_SvcReadDTCInformationNoParam_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | UDS_SvcReadDTCInformationRDTCEDBR_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | UDS_SvcReadDTCInformationRUDMDTCBSM_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | UDS_SvcReadDTCInformationRUDMDTCSSBDTC_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | UDS_SvcReadDTCInformationRUDMDTCEDRBDN_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | UDS_SvcReadDTCInformationRDTCEDI_2020 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | UDS_SvcReadDTCInformationRWWHOBDDTCBMR_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | UDS_SvcReadDTCInformationRWWHOBDDTCWPS_2013 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |
|  | UDS_SvcReadDTCInformationRDTCBGI_2020 | Writes to the transmit queue a request for UDS service ReadDTCInformation. |

| | Function | Description |
|---|--|---|
|  | UDS_SvcInputOutputControlByIdentifier_2013 | Writes to the transmit queue a request for UDS service InputOutputControlByIdentifier. |
|  | UDS_SvcRoutineControl_2013 | Writes to the transmit queue a request for UDS service RoutineControl. |
|  | UDS_SvcRequestDownload_2013 | Writes to the transmit queue a request for UDS service RequestDownload. |
|  | UDS_SvcRequestUpload_2013 | Writes to the transmit queue a request for UDS service RequestUpload. |
|  | UDS_SvcTransferData_2013 | Writes to the transmit queue a request for UDS service TransferData. |
|  | UDS_SvcRequestTransferExit_2013 | Writes to the transmit queue a request for UDS service RequestTransferExit. |
|  | UDS_SvcAccessTimingParameter_2013 | Writes to the transmit queue a request for UDS service AccessTimingParameter. |
|  | UDS_SvcRequestFileTransfer_2013 | Writes to the transmit queue a request for UDS service RequestFileTransfer. |
|  | UDS_SvcAuthenticationDA_2020 | Writes to the transmit queue a request for UDS service Authentication with deAuthenticate subfunction (ISO-14229-1:2020). |
|  | UDS_SvcAuthenticationVCU_2020 | Writes to the transmit queue a request for UDS service Authentication with verifyCertificateUnidirectional subfunction (ISO-14229-1:2020). |
|  | UDS_SvcAuthenticationVCB_2020 | Writes to the transmit queue a request for UDS service Authentication with verifyCertificateBidirectional subfunction (ISO-14229-1:2020). |
|  | UDS_SvcAuthenticationPOWN_2020 | Writes to the transmit queue a request for UDS service Authentication with proofOfOwnership subfunction (ISO-14229-1:2020). |
|  | UDS_SvcAuthenticationRCFA_2020 | Writes to the transmit queue a request for UDS service Authentication with requestChallengeForAuthentication subfunction (ISO-14229-1:2020). |
|  | UDS_SvcAuthenticationVPOWNU_2020 | Writes to the transmit queue a request for UDS service Authentication with verifyProofOfOwnershipUnidirectional subfunction (ISO-14229-1:2020). |
|  | UDS_SvcAuthenticationVPOWNB_2020 | Writes to the transmit queue a request for UDS service Authentication with verifyProofOfOwnershipBidirectional subfunction (ISO-14229-1:2020). |
|  | UDS_SvcAuthenticationAC_2020 | Writes to the transmit queue a request for UDS service Authentication with authenticationConfiguration subfunction (ISO-14229-1:2020). |

3.8.1 UDS_Initialize_2013

Initializes a PUDS channel based on a PCANTP channel handle (without CAN FD support).

Syntax

C

```
uds_status UDS_Initialize_2013(
    cantp_handle channel,
    cantp_baudrate baudrate,
    cantp_hwtype hw_type,
    uint32_t io_port,
    uint16_t interrupt);
```


C++

```
uds_status UDS_Initialize_2013(
    cantp_handle channel,
    cantp_baudrate baudrate,
    cantp_hwtype hw_type = 0,
    uint32_t io_port = 0,
    uint16_t interrupt = 0);
```

Parameters

| Parameter | Description |
|-----------|---|
| channel | The handle of a PCANTP channel (see cantp_handle on page 105). |
| baudrate | The speed for the communication (see cantp_baudrate on page 116). |
| hw_type | Non-plug and play: the type of hardware (see cantp_hwtype on page 113). |
| io_port | Non-plug and play: the I/O address for the parallel port. |
| interrupt | Non-plug and play: interrupt number of the parallel port. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_ALREADY_INITIALIZED</code> | Indicates that the desired PUDS channel is already in use. |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory. |
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Channel not available. |
| <code>PUDS_STATUS_FLAG_PCAN_STATUS</code> | This error flag states that the error is composed of a more precise PCAN-Basic error. |

Remarks

As indicated by its name, the `UDS_Initialize_2013` function initiates a PUDS channel, preparing it for communication within the CAN bus connected to it. Calls to the other functions will fail if they are used with a channel handle, different than `PCANTP_HANDLE_NONEBUS`, that has not been initialized yet. Each initialized channel should be released when it is not needed anymore.

Initializing a PUDS channel means:

- to reserve the channel for the calling application/process
- to allocate channel resources, like receive and transmit queues
- to forward initialization to PCAN-ISO-TP 3.x API and PCAN-Basic API, hence registering/connecting the Hardware denoted by the channel handle
- To set up the default values of the different parameters (see `UDS_SetValue_2013` on page 629).
- To configure default standard ISO-TP mappings (see UDS and ISO-TP Network Addressing Information on page 770):
 - Functional request using 11 bits CAN identifier and normal addressing, from External Test Equipment address (`PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT`) to OBD functional address (`PUDS_ISO_15765_4_ADDR_OBD_FUNCTIONAL`),
 - Physical requests and responses using 11 bits CAN identifier and normal addressing, between the External Test Equipment address (`PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT`) and standard ECU addresses (ECU #1 to #8)
- To configure PCAN-ISO-TP 3.x to filter CAN frames to increase performance (frames that do not match a mapping or a CAN ID in the white-list filter are ignored and discarded).

The initialization process will fail if an application tries to initialize a PCANTP channel handle that has already been initialized within the same process.

Take into consideration that initializing a channel causes a reset of the CAN hardware. In this way errors like BUSOFF, BUSHEAVY, and BUSLIGHT, are removed.

Example

The following example shows the initialize and uninitialized processes for a Plug-And-Play channel (channel 2 of the PCAN-PCI).

C/C++

```
uds_status result;

// The Plug & Play channel (PCAN-PCI) is initialized
result = UDS_Initialize_2013(PCANTP_HANDLE_PCIBUS2, PCANTP_BAUDRATE_500K, (cantp_hwtype)0, 0, 0);
if (!UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Initialization failed\n");
else
    printf("PCAN-PCI (Ch-2) was initialized\n");

// All initialized channels are released
UDS_Uninitialize_2013(PCANTP_HANDLE_NONEBUS);
```

See also: [UDS_Uninitialize_2013](#) on page 628, [UDS_InitializeFD_2013](#) on page 626, [UDS_GetValue_2013](#) on page 630, [Understanding PCAN-UDS 2.x](#) on page 11.

Class-method version: [Initialize_2013](#) on page 142.

3.8.2 UDS_InitializeFD_2013

Initializes a PUDS channel based on a PCANTP channel handle (including CAN FD support).

Syntax

C/C++

```
uds_status UDS_InitializeFD_2013(
    cantp_handle channel,
    const cantp_bitrate bitrate_fd);
```

Parameters

| Parameter | Description |
|------------|---|
| channel | The handle of a FD capable PUDS channel (see cantp_handle on page 105). |
| bitrate_fd | The speed for the communication (see cantp_bitrate on page 101, FD Bit Rate Parameter Definitions on page 102). |

Returns

The return value is a [uds_status](#) code. [PUDS_STATUS_OK](#) is returned on success. The typical errors in case of failure are:

| | |
|---------------------------------|---|
| PUDS_STATUS_ALREADY_INITIALIZED | Indicates that the desired PUDS channel is already in use. |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory. |
| PUDS_STATUS_NOT_INITIALIZED | Channel not available. |
| PUDS_STATUS_FLAG_PCAN_STATUS | This error flag states that the error is composed of a more precise PCAN-Basic error. |

Remarks

The `UDS_InitializeFD_2013` function initiates a FD capable PUDS channel, preparing it for communication within the CAN bus connected to it. Calls to the other functions will fail, if they are used with a channel handle, different than `PCANTP_HANDLE_NONEBUS`, that has not been initialized yet. Each initialized channel should be released when it is not needed anymore.

Initializing a PUDS channel means:

- └ To reserve the channel for the calling application/process.
- └ To allocate channel resources, like receive and transmit queues.
- └ To forward initialization to PCAN-ISO-TP 3.x API and PCAN-Basic API, hence registering/connecting the Hardware denoted by the channel handle.
- └ To set up the default values of the different parameters (see `UDS_SetValue_2013` on page 629).
- └ To configure default standard ISO-TP mappings (see UDS and ISO-TP Network Addressing Information on page 770):
 - Functional request using 11 bits CAN identifier and normal addressing, from External Test Equipment address (`PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT`) to OBD functional address (`PUDS_ISO_15765_4_ADDR_OBD_FUNCTIONAL`),
 - Physical requests and responses using 11 bits CAN identifier and normal addressing, between the External Test Equipment address (`PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT`) and standard ECU addresses (ECU #1 to #8)
- └ To configure PCAN-ISO-TP 3.x to filter CAN frames to increase performance (frames that do not match a mapping or a CAN ID in the white-list filter are ignored and discarded).

The initialization process will fail if an application tries to initialize a PCANTP channel handle that has already been initialized within the same process.

Take into consideration, that initializing a channel causes a reset of the CAN hardware. In this way errors like BUSOFF, BUSHEAVY, and BUSLIGHT, are removed.

Example

The following example shows the initialize and uninitialized processes for a Plug and Play, FD capable channel (channel 2 of a PCAN-USB hardware).

C/C++

```
uds_status result;

// The Plug and Play channel (PCAN-USB) is initialized @500kbps/2Mbps.
result = UDS_InitializeFD_2013(PCANTP_HANDLE_USBBUS2, "f_clock=8000000, nom_brp=10,
nom_tseg1=12, nom_tseg2=3, nom_sjw=1, data_brp=4, data_tseg1=7, data_tseg2=2, data_sjw=1");
if (!UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Initialization failed\n");
else
    printf("PCAN-USB (Ch-2) was initialized\n");

// All initialized channels are released.
UDS_Uninitialize_2013(PCANTP_HANDLE_NONEBUS);
```

See also: `UDS_Uninitialize_2013` on page 628, `Using PCAN-UDS 2.x` on page 12, `cantp_bitrate` on page 101, `FD Bit Rate Parameter Definitions` on page 102.

Class-method version: `InitializeFD_2013` on page 148.

3.8.3 UDS_Uninitialize_2013

Uninitializes a PUDS channel.

Syntax

C/C++

```
uds_status UDS_Uninitialize_2013(
    cantp_handle channel);
```

Parameters

| Parameter | Description |
|-----------|---|
| channel | The handle of a PCANTP channel handle (see cantp_handle on page 105). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical error in case of failure is:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel cannot be uninitialized because it was not found in the list of reserved channels of the calling application. |
|--|---|

Remarks

A PUDS channel can be released using one of these possibilities:

- **Single-Release:** Giving the handle of a PUDS channel initialized before with the function `UDS_Initialize_2013`. If the given channel cannot be found, then an error is returned.
- **Multiple-Release:** Giving the handle value `PCANTP_HANDLE_NONEBUS` which instructs the API to search for all channels initialized by the calling application and release them all. This option causes no errors if no hardware were uninitialized.

Example

The following example shows the initialize and uninitialized processes for a Plug-And-Play channel (channel 2 of a PCAN-PCI hardware).

C/C++

```
uds_status result;

// The Plug & Play channel (PCAN-PCI) is initialized
result = UDS_Initialize_2013(PCANTP_HANDLE_PCIBUS2, PCANTP_BAUDRATE_500K, (cantp_hwtype)0, 0, 0);
if (!UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Initialization failed\n");
else
    printf("PCAN-PCI (Ch-2) was initialized\n");

// Release channel
result = UDS_Uninitialize_2013(PCANTP_HANDLE_PCIBUS2);
if (!UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Uninitialization failed\n");
else
    printf("PCAN-PCI (Ch-2) was released\n");
```

See also: `UDS_Initialize_2013` on page 624, `UDS_InitializeFD_2013` on page 626.

Class-method version: `Uninitialize_2013` on page 148.

3.8.4 UDS_SetValue_2013

Sets a configuration or information value within a PUDS channel.

Syntax

C/C++

```
uds_status UDS_SetValue_2013(
    cantp_handle channel,
    uds_parameter parameter,
    void* buffer,
    uint32_t buffer_size);
```

Parameters

| Parameter | Description |
|-------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| Parameter | The code of the value to be set (see uds_parameter on page 41). |
| Buffer | The buffer containing the value to be set. |
| Buffer_size | The size in bytes of the given buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | Indicates that the parameters passed to the function are invalid. Check the value of the buffer and assert it is compatible with the buffer length. |

Remarks

Use the function `UDS_SetValue_2013` to set configuration information or environment values of a PUDS channel.

Note: That any calls with non PCAN-UDS 2.x API parameters (i.e. `uds_parameter`) will be forwarded to PCAN-ISO-TP 3.x API or PCAN-Basic API.

More information about the parameters and values that can be set can be found in Detailed Parameters Characteristics on page 45.

Example

The following example shows the use of the function `UDS_SetValue_2013` on the channel `PCANTP_HANDLE_PCIBUS2` to enable debug mode.

Note: It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uint8_t buffer;

// Enable error messages
buffer = PUDS_DEBUG_LVL_ERROR;
result = UDS_SetValue_2013(PCANTP_HANDLE_PCIBUS2, PUDS_PARAMETER_DEBUG, &buffer,
    sizeof(uint8_t));
if (!UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Failed to set value\n");
else
    printf("Value changed successfully\n");
```

See also: `UDS_GetValue_2013` on page 630, `uds_parameter` on page 41, Detailed Parameters Characteristics on page 45.

Class-method version: `SetValue_2013` on page 153.

3.8.5 UDS_GetValue_2013

Retrieves information from a PUDS channel.

Syntax

C/C++

```
uds_status UDS_GetValue_2013(
    cantp_handle channel,
    uds_parameter parameter,
    void* buffer,
    uint32_t buffer_size);
```

Parameters

| Parameter | Description |
|-------------|--|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| parameter | The code of the value to retrieve (see <code>uds_parameter</code> on page 41). |
| buffer | The buffer to return the required value. |
| buffer_size | The size in bytes of the given buffer. |


Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | Indicates that the parameters passed to the function are invalid. Check the value of the buffer and assert it is compatible with the buffer length. |

Example

The following example shows the use of the function `UDS_GetValue_2013` to retrieve the separation time parameter (STmin) on the USB channel 1. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uint8_t buffer = 0;

// Get the value of the Separation Time (STmin) parameter
result = UDS_GetValue_2013(PCANTP_HANDLE_USBBUS1, PUDS_PARAMETER_SEPARATION_TIME, &buffer,
    sizeof(uint8_t));
if (!UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
{
    printf("Failed to get value\n");
}
else
{
    printf("%d\n", buffer);
}
```

See also: `UDS_SetValue_2013` on page 629, `uds_parameter` on page 41, Detailed Parameters Characteristics on page 45.

Class-method version: `GetValue_2013` on page 178.

3.8.6 UDS_AddMapping_2013

Adds a user-defined mapping between a CAN identifier and a network address information. Defining a mapping enables PCAN-ISO-TP 3.x communication with 11Bits CAN identifier or with opened Addressing Formats (like `PCANTP_ISOTP_FORMAT_NORMAL` or `PCANTP_ISOTP_FORMAT_EXTENDED`).

Syntax

C/C++

```
uds_status UDS_AddMapping_2013(
    cantp_handle channel,
    uds_mapping* mapping);
```

Parameters

| Parameter | Description |
|-----------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| Mapping | PUDS mapping to be added (see <code>uds_mapping</code> on page 25). |

Remark

By default, some mappings are initialized in the PCAN-UDS 2.x API (see UDS and ISO-TP Network Addressing Information on page 770).

With version 1.0, to handle custom ISO-TP messages, support advised to use `CANTP_AddMapping` function in conjunction with PCAN-UDS. Updates of version 1 then added new parameters to manage mappings (like `PUDS_PARAM_ADD_MAPPING`). With version 2.0, PCAN-UDS will ignore ISO-TP messages that do not strictly correspond to the mappings and the configuration of the UDS node. It is now advised not to directly use lower APIs to define mappings for higher layer API. Nevertheless, to ensure retro-compatibility, an ISO-TP message (with no mapping defined in UDS) will still be readable in UDS if its target address matches the server address.


Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

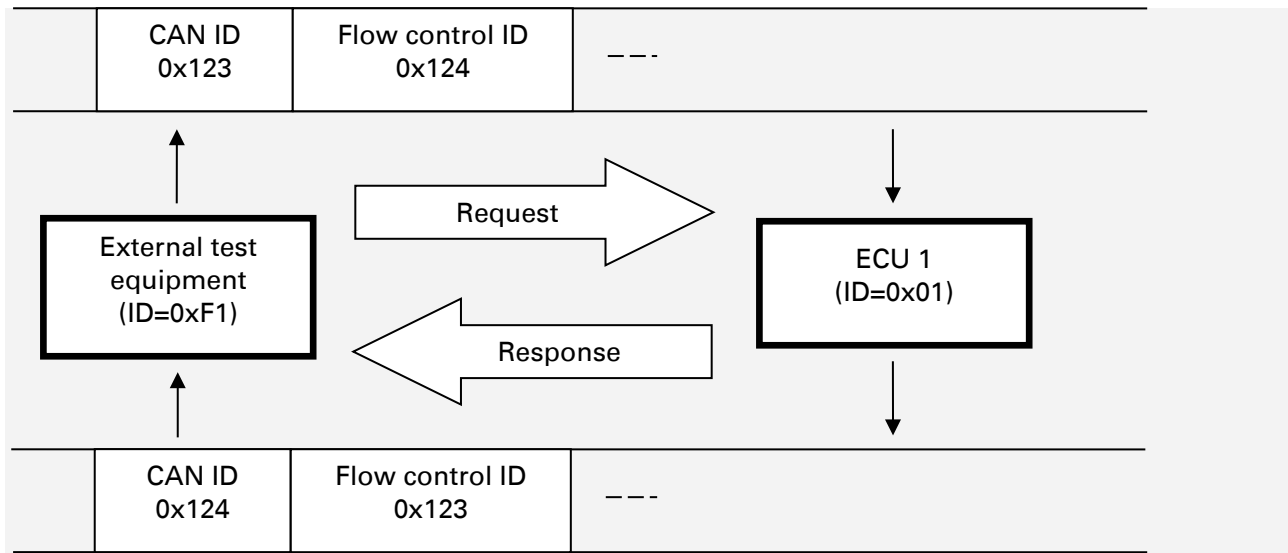
| | |
|--|--|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | Indicates that the given mapping is null. |
| <code>PUDS_STATUS_ALREADY_INITIALIZED</code> | A mapping with the same CAN identifier already exists. |
| <code>PUDS_STATUS_MAPPING_INVALID</code> | Mapping is not valid regarding the UDS standard. |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory to define mapping. |

Example

The following example shows the use of the function `UDS_AddMapping_2013` on the USB channel 1. It creates mappings to communicate with custom CAN identifiers between test equipment and ECU 1 in ISO-15765-2 11bits normal addressing on initialized USB channel 1:

 **Note:** It is assumed that the channel was already initialized.

Set CAN identifier = 0x123 and flow control id = 0x124 for request and set CAN identifier = 0x124 and flow control id = 0x123 for response. Test equipment address corresponds to 0xF1 and ECU 1 address corresponds to 0x01. Here is a small scheme of the mapping:

**C/C++**

```
uds_mapping request_mapping;
memset(&request_mapping, 0, sizeof(request_mapping));
request_mapping.can_id = 0x123;
request_mapping.can_id_flow_ctrl = 0x124;
request_mapping.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
request_mapping.nai.extension_addr = 0;
request_mapping.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
request_mapping.can_tx_dlc = 8;
request_mapping.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
request_mapping.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
request_mapping.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;

uds_mapping response_mapping;
memset(&response_mapping, 0, sizeof(response_mapping));
response_mapping = request_mapping;
response_mapping.can_id = request_mapping.can_id_flow_ctrl;
response_mapping.can_id_flow_ctrl = request_mapping.can_id;
response_mapping.nai.source_addr = request_mapping.nai.target_addr;
response_mapping.nai.target_addr = request_mapping.nai.source_addr;

uds_status status;
status = UDS_AddMapping_2013(PCANTP_HANDLE_USBBUS1, &request_mapping);
if (UDS_StatusIsOk_2013(status, PUDS_STATUS_OK, false))
    printf("Add request mapping\n");
else
    printf("Failed to add request mapping\n");
status = UDS_AddMapping_2013(PCANTP_HANDLE_USBBUS1, &response_mapping);
if (UDS_StatusIsOk_2013(status, PUDS_STATUS_OK, false))
    printf("Add response mapping\n");
else
    printf("Failed to add response mapping\n");
```

See also: [UDS_RemoveMapping_2013](#) on page 634, [UDS_RemoveMappingByCanId_2013](#) on page 633.

Class-method version: [AddMapping_2013](#) on page 164.

3.8.7 UDS_RemoveMappingByCanId_2013

Removes all user defined PUDS mappings corresponding to a CAN identifier.

Syntax

C/C++

```
uds_status UDS_RemoveMappingByCanId_2013(
    cantp_handle channel,
    uint32_t can_id);
```

Parameters

| Parameter | Description |
|-----------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| can_id | The mapped CAN identifier to search for that identifies the mappings to remove (see predefined uds_can_id values on page 58). |


Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|--|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The CAN identifier to remove is not specified in a mapping. |

Example

The following example shows the use of the function `UDS_RemoveMappingByCanId_2013` on the USB channel 1. It creates a mapping then removes it using its CAN identifier on initialized USB channel 1:

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_mapping request_mapping;
memset(&request_mapping, 0, sizeof(request_mapping));
request_mapping.can_id = 0x123;
request_mapping.can_id_flow_ctrl = 0x124;
request_mapping.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
request_mapping.nai.extension_addr = 0;
request_mapping.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
request_mapping.can_tx_dlc = 8;
request_mapping.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
request_mapping.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
request_mapping.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;

uds_status status;
status = UDS_AddMapping_2013(PCANTP_HANDLE_USBBUS1, &request_mapping);
if (UDS_StatusIsOk_2013(status, PUDS_STATUS_OK, false))
{
    // Remove mapping using its can identifier
    status = UDS_RemoveMappingByCanId_2013(PCANTP_HANDLE_USBBUS1, 0x123);
    if (UDS_StatusIsOk_2013(status, PUDS_STATUS_OK, false))
        printf("Remove request mapping using its can identifier\n");
    else
        printf("Failed to remove request mapping\n");
}
else
```

```
{
    printf("Failed to add request mapping\n");
}
```

See also: `uds_mapping` on page 25, `UDS_RemoveMapping_2013` on page 634, `UDS_AddMapping_2013` on page 631.
Class-method version: `RemoveMappingByCanId_2013` on page 171.

3.8.8 UDS_RemoveMapping_2013

Removes a user defined PUDS mapping.

Syntax

C/C++

```
uds_status UDS_RemoveMapping_2013(
    cantp_handle channel,
    uds_mapping mapping);
```

Parameters

| Parameter | Description |
|-----------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| mapping | The mapping to remove (see <code>uds_mapping</code> on page 25). |


Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|--|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The mapping is not a valid mapping. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping is not in the mapping list. |

Example

The following example shows the use of the function `UDS_RemoveMapping_2013` on the USB channel 1. It creates a mapping then removes it.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_mapping request_mapping;
memset(&request_mapping, 0, sizeof(request_mapping));
request_mapping.can_id = 0x123;
request_mapping.can_id_flow_ctrl = 0x124;
request_mapping.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
request_mapping.nai.extension_addr = 0;
request_mapping.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
request_mapping.can_tx_dlc = 8;
request_mapping.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
request_mapping.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
request_mapping.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;

uds_status status;
status = UDS_AddMapping_2013(PCANTP_HANDLE_USBBUS1, &request_mapping);
if (UDS_StatusIsOk_2013(status, PUDS_STATUS_OK, false))
{
    // Remove the request mapping
```

```

status = UDS_RemoveMapping_2013(PCANTP_HANDLE_USBBUS1, request_mapping);
if (UDS_StatusIsOk_2013(status, PUDS_STATUS_OK, false))
{
    printf("Remove request mapping\n");
}
else
{
    printf("Failed to remove request mapping\n");
}
}
else
{
    printf("Failed to add request mapping\n");
}
}

```

See also: [uds_mapping](#) on page 25, [UDS_RemoveMappingByCanId_2013](#) on page 633, [UDS_AddMapping_2013](#) on page 631.

Class-method version: [RemoveMapping_2013](#) on page 168.

3.8.9 UDS_AddCanIdFilter_2013

Adds an entry to the CAN identifier white-list filtering. This function allows the user to listen to non-UDS CAN frames or UUDT messages with no configured mapping.

Syntax

C/C++

```

uds_status UDS_AddCanIdFilter_2013(
    cantp_handle channel,
    uint32_t can_id);

```

Parameters

| Parameter | Description |
|-----------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| can_id | CAN identifier to add in the white-list (see predefined uds_can_id values on page 58). |


Returns

The return value is a [uds_status](#) code. [PUDS_STATUS_OK](#) is returned on success. The typical errors in case of failure are:

| | |
|---|--|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_ALREADY_INITIALIZED | The CAN identifier is already in the white list. |
| PUDS_STATUS_NO_MEMORY | Memory allocation error when adding the new element in the white list. |

Example

The following example shows the use of the function [UDS_AddCanIdFilter_2013](#) the channel [PCANTP_HANDLE_USBBUS1](#). It adds a filter on 0xD1 CAN identifier.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
result = UDS_AddCanIdFilter_2013(PCANTP_HANDLE_USBBUS1, 0xD1);
if (!UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Error adding CAN ID filter.\n");
```

See also: `UDS_RemoveCanIdFilter_2013` on page 636, `uds_can_id` on page 58.

Class-method version: `AddCanIdFilter_2013` on page 175.

3.8.10 UDS_RemoveCanIdFilter_2013

Removes an entry from the CAN identifier white-list filtering.

Syntax**C/C++**

```
uds_status UDS_RemoveCanIdFilter_2013(
    cantp_handle channel,
    uint32_t can_id);
```

Parameters

| Parameter | Description |
|-----------|--|
| Channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| can_id | CAN identifier to remove (see predefined <code>uds_can_id</code> values on page 58). |


Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical error in case of failure is:

| | |
|--|--|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. Or the CAN identifier is not in the white list. |
|--|--|

Example

The following example shows the use of the function `UDS_RemoveCanIdFilter_2013` on the channel `PCANTP_HANDLE_USBBUS1`. It adds a filter on 0xD1 CAN identifier then removes it.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
result = UDS_AddCanIdFilter_2013(PCANTP_HANDLE_USBBUS1, 0xD1);
if (!UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Error adding CAN ID filter.\n");

// Remove previously added can identifier filter
result = UDS_RemoveCanIdFilter_2013(PCANTP_HANDLE_USBBUS1, 0xD1);
if (!UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Error removing CAN ID filter.\n");
```

See also: `UDS_AddCanIdFilter_2013` on page 635.

Class-method version: `RemoveCanIdFilter_2013` on page 176.

3.8.11 UDS_GetMapping_2013

Retrieves a mapping matching the given CAN identifier and message type (11bits, 29 bits, FD, etc.).

Syntax

C/C++

```
uds_status UDS_GetMapping_2013(
    cantp_handle channel,
    uds_mapping* buffer,
    uint32_t can_id,
    cantp_can_msgtype can_msgtype);
```

Parameters

| Parameter | Description |
|-------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| Buffer | Output, buffer to store the searched mapping (see uds_mapping on page 25). |
| Can_id | The CAN identifier to look for (see predefined uds_can_id values on page 58). |
| Can_msgtype | The CAN message type to look for (11bits, 29 bits, FD, etc.). See also cantp_can_msgtype on page 121. |


Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical error in case of failure is:

| | |
|-------------------------------------|--|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The buffer is invalid. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | Indicates that no matching mapping was found in the registered mapping list. |

Example

The following example shows the use of the function `UDS_GetMapping_2013` on the channel `PCANTP_HANDLE_USBBUS1`. It gets the mapping for the 0x7E0 CAN identifier then print it.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_mapping result_mapping;
result = UDS_GetMapping_2013(PCANTP_HANDLE_USBBUS1, &result_mapping,
    PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1, PCANTP_CAN_MSGTYPE_STANDARD);

if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
{
    printf(« Mapping defined for the %d can identifier : », result_mapping.can_id) ;
    printf(« \n\t- Flow control identifier : %d », result_mapping.can_id_flow_ctrl) ;
    printf(« \n\t- Can message type : %d », result_mapping.can_msgtype) ;
    printf(« \n\t- TX DLC : %d », result_mapping.can_tx_dlc) ;
    printf(« \n\t- Extension address : %d », result_mapping.nai.extension_addr) ;
    printf(« \n\t- Protocol : %d », result_mapping.nai.protocol) ;
    printf(« \n\t- Source address : %d », result_mapping.nai.source_addr) ;
    printf(« \n\t- Target address : %d », result_mapping.nai.target_addr) ;
    printf(« \n\t- Target type : %d\n », result_mapping.nai.target_type) ;
}
else
{
    printf(« Error, cannot get mapping information.\n ») ;
}
```

See also: [uds_mapping](#) on page 25, [UDS_GetMappings_2013](#) on page 638.

Class-method version: [GetMapping_2013](#) on page 193.

3.8.12 UDS_GetMappings_2013

Retrieves all the mappings defined for a PUDS channel.

Syntax

C/C++

```
uds_status UDS_GetMappings_2013(
    cantp_handle channel,
    uds_mapping * buffer,
    uint16_t buffer_length,
    uint16_t* count);
```

Parameters

| Parameter | Description |
|---------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| Buffer | Output, a buffer to store an array of uds_mapping (see uds_mapping on page 25). |
| Buffer_length | The number of uds_mapping element the buffer can store. |
| Count | Output, the actual number of elements copied in the buffer. |

Remark

By default, some mappings are initialized in the PCAN-UDS 2.x API (see UDS and ISO-TP Network Addressing Information on page 770).


Returns

The return value is a [uds_status](#) code. [PUDS_STATUS_OK](#) is returned on success. The typical errors in case of failure are:

| | |
|--|--|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_BUFFER_TOO_SMALL | The given buffer is too small to store all mappings. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The buffer is invalid. |

Example

The following example shows the use of the function [UDS_GetMappings_2013](#) on [PCANTP_HANDLE_USBBUS1](#). It displays all mappings added on the channel.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uint16_t count = 256;
uds_mapping mappings[256];
result = UDS_GetMappings_2013(PCANTP_HANDLE_USBBUS1, mappings, count, &count);

if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
{
    for (int i = 0; i < count; i++)
    {
        printf("mappings[%d]:", i);
        printf("\n\t- Can identifier: %d", mappings[i].can_id);
        printf("\n\t- Flow control identifier: %d", mappings[i].can_id_flow_ctrl);
    }
}
```

```

        printf("\n\t- Can message type: %d", mappings[i].can_msgtype);
        printf("\n\t- TX DLC: %d", mappings[i].can_tx_dlc);
        printf("\n\t- Extension address: %d", mappings[i].nai.extension_addr);
        printf("\n\t- Protocol: %d", mappings[i].nai.protocol);
        printf("\n\t- Source address: %d", mappings[i].nai.source_addr);
        printf("\n\t- Target address: %d", mappings[i].nai.target_addr);
        printf("\n\t- Target type: %d\n", mappings[i].nai.target_type);
    }
}
else
{
    printf("Error, cannot get mappings\n");
}

```

See also: `uds_mapping` on page 25, `UDS_GetMapping_2013` on page 637.

Class-method version: `GetMappings_2013` on page 196.

3.8.13 UDS_GetSessionInformation_2013

Gets current ECU session information.

Syntax

C/C++

```

uds_status UDS_GetSessionInformation_2013(
    cantp_handle channel,
    uds_sessioninfo *session_info);

```

Parameters

| Parameter | Description |
|--------------|--|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| session_info | Input, the mapping to search for. Output, the session filled if an ECU session exists (see <code>uds_sessioninfo</code> on page 22). |


Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. Or the ECU session information is not initialized. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is invalid. |

Example

The following example shows the use of the function `UDS_GetSessionInformation_2013` on the channel `PCANTP_HANDLE_USBUSB1`. It gets the session information for a given mapping and prints it.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```

uds_status result;
uds_sessioninfo session_info;

memset(&session_info, 0, sizeof(session_info));

// Mapping to search
session_info.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;

```

```

session_info.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
session_info.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
session_info.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
session_info.nai.extension_addr = 0;

result = UDS_GetSessionInformation_2013(PCANTP_HANDLE_USBBUS1, &session_info);

if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
{
    printf("Current session info:");
    printf("\n\t- Can message type: %d", session_info.can_msg_type);
    printf("\n\t- Extension address: %d", session_info.nai.extension_addr);
    printf("\n\t- Protocol: %d", session_info.nai.protocol);
    printf("\n\t- Source address: %d", session_info.nai.source_addr);
    printf("\n\t- Target address: %d", session_info.nai.target_addr);
    printf("\n\t- Target type: %d", session_info.nai.target_type);
    printf("\n\t- S3 client value: %d", session_info.s3_client_ms);
    printf("\n\t- Session type: %d", session_info.session_type);
    printf("\n\t- P2can server max timeout enhanced: %d",
           session_info.timeout_enhanced_p2can_server_max);
    printf("\n\t- P2can server max timeout: %d\n", session_info.timeout_p2can_server_max);
}
else
{
    printf("Error, cannot get session information.\n");
}

```

See also: [uds_sessioninfo](#) on page 22, [UDS_SetValue_2013](#) on page 629.

Class-method version: [GetSessionInformation_2013](#) on page 200.

3.8.14 UDS_StatusIsOk_2013

Checks if a PUDS status matches an expected result (default is [PUDS_STATUS_OK](#)).

Syntax

C

```

bool UDS_StatusIsOk_2013(
    const uds_status status,
    const uds_status status_expected,
    bool strict_mode);

```

C++

```

bool UDS_StatusIsOk_2013(
    const uds_status status,
    const uds_status status_expected = PUDS_STATUS_OK,
    bool strict_mode = false);

```

Parameters

| Parameter | Description |
|-----------------|---|
| status | The PUDS status to analyze (see uds_status on page 32). |
| status_expected | The expected PUDS status (see uds_status on page 32). The default value is PUDS_STATUS_OK . |
| strict_mode | Enable strict mode (default is false). Strict mode ensures that bus or extra information are the same. |

Returns

The return value is true if the status matches expected parameter.

Remarks

When checking a `uds_status`, it is preferred to use `UDS_StatusIsOk_2013` instead of comparing it with the `==` operator because `UDS_StatusIsOk_2013` can remove information flag (in non-strict mode).

Example

The following example shows the use of the function `UDS_StatusIsOk_2013` after initializing the channel `PCANTP_HANDLE_PCIBUS2`.

C/C++

```
uds_status result;
result = UDS_Initialize_2013(PCANTP_HANDLE_PCIBUS2, PCANTP_BAUDRATE_500K, (cantp_hwtype)0, 0, 0);
if (!UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Initialization failed\n");
else
    printf("PCAN-PCI (Ch-2) was initialized\n");
```

See also: `uds_status` on page 32.

Class-method version: `StatusIsOk_2013` on page 203.

3.8.15 UDS_GetErrorText_2013

Gets a descriptive text for a PUDS error code.

Syntax

C/C++

```
uds_status UDS_GetErrorText_2013(
    uds_status error_code,
    uint16_t language,
    char* buffer,
    uint32_t buffer_size);
```

Parameters

| Parameter | Description |
|-------------|---|
| error_code | A <code>uds_status</code> error code (see <code>uds_status</code> on page 32). |
| Language | The current languages available for translation are: Neutral (0x00), German (0x07), English (0x09), Spanish (0x0A), Italian (0x10) and French (0x0C). |
| buffer | A buffer for a null-terminated char array. |
| Buffer_size | Buffer size in bytes. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical error in case of failure is:

| | |
|--|--|
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | Indicates that the parameters passed to the function are invalid. Check the parameter 'buffer'; it should point to a char array, big enough to allocate the text for the given error code. |
|--|--|

Remarks

The Primary Language IDs are codes used by Windows OS from Microsoft, to identify a human language. The API currently supports the following languages:

| Language | Primary Language ID |
|----------------------------|---------------------|
| Neutral (system-dependent) | 00h (0) |

| Language | Primary Language ID |
|----------|---------------------|
| English | 09h (9) |
| German | 07h (7) |
| French | 0Ch (12) |
| Italian | 10h (16) |
| Spanish | 0Ah (10) |

Note: If the buffer is too small for the resulting text, the error 0x80008000 (`PUDS_STATUS_MASK_PCAN|PCAN_ERROR_ILLPARAMVAL`) is returned. Even when only short texts are being currently returned, a text within this function can have a maximum of 255 characters. For this reason, it is recommended to use a buffer with a length of at least 256 bytes.

Example

The following example shows the use of the function `UDS_GetErrorText_2013` to get the description of an error. The language of the description's text will be the same used by the operating system (if its language is supported; otherwise English is used).

Note: It is assumed that the channel was NOT initialized (to generate an error).

C/C++

```
char str_msg[256];
uds_status result;
uds_status error_result;
error_result = UDS_Uninitialize_2013(PCANTP_HANDLE_USBBUS1);
result = UDS_GetErrorText_2013(error_result, 0x0, str_msg, 256);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false) && !UDS_StatusIsOk_2013(error_result,
    PUDS_STATUS_OK, false))
    printf("%s\n", str_msg);
```

See also: `uds_status` on page 32.

Class-method version: `GetErrorText_2013` on page 209.

3.8.16 UDS_GetCanBusStatus_2013

Gets information about the internal CAN bus status of a PUDS channel.

Syntax

C/C++

```
uds_status UDS_GetCanBusStatus_2013(
    cantp_handle channel);
```

Parameters

| Parameter | Description |
|-----------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---|---|
| <code>PUDS_STATUS_FLAG_BUS_LIGHT</code> | Indicates a bus error within the given PUDS channel. The hardware is in bus-light status. |
| <code>PUDS_STATUS_FLAG_BUS_HEAVY</code> | Indicates a bus error within the given PUDS channel. The hardware is in bus-heavy status. |
| <code>PUDS_STATUS_FLAG_BUS_OFF</code> | Indicates a bus error within the given PUDS channel. The hardware is in bus-off status. |

| | |
|-----------------------------|--|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
|-----------------------------|--|


Remarks

When the hardware status is bus-off, an application cannot communicate anymore. Consider using the PCAN-Basic property `PCAN_BUSOFF_AUTORESET` which instructs the API to automatically reset the CAN controller when a bus-off state is detected.

Another way to reset errors like bus-off, bus-heavy and bus-light, is to uninitialized and initialize again the channel used. This causes a hardware reset.

Example

The following example shows the use of the function `UDS_GetCanBusStatus_2013` on the channel `PCANTP_HANDLE_PCIBUS1`. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;

// Check the status of the PCI channel
result = UDS_GetCanBusStatus_2013(PCANTP_HANDLE_PCIBUS1);
switch (result)
{
case PUDS_STATUS_FLAG_BUS_LIGHT:
    printf("PCAN-PCI (Ch-1): Handling a BUS-LIGHT status\n");
    break;
case PUDS_STATUS_FLAG_BUS_HEAVY:
    printf("PCAN-PCI (Ch-1): Handling a BUS-HEAVY status\n");
    break;
case PUDS_STATUS_FLAG_BUS_OFF:
    printf("PCAN-PCI (Ch-1): Handling a BUS-OFF status\n");
    break;
case PUDS_STATUS_OK:
    printf("PCAN-PCI (Ch-1): Status is OK\n");
    break;
default:
    // An error occurred
    printf("Failed to retrieve status\n");
    break;
}
```

See also: `uds_status` on page 32.

Class-method version: `GetCanBusStatus_2013` on page 189.

3.8.17 UDS_MsgAlloc_2013

Allocates a PUDS message using the given configuration.

Syntax

C/C++

```
uds_status UDS_MsgAlloc_2013(
    uds_msg* msg_buffer,
    uds_msgconfig msg_configuration,
    uint32_t msg_data_length
);
```

Parameters

| Parameter | Description |
|-------------------|--|
| msg_buffer | A uds_msg structure buffer. It will be freed if required (see uds_msg on page 21). |
| msg_configuration | Configuration of the message to allocate (see uds_msgconfig on page 27). |
| msg_data_length | Length of the message data. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|--|
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the parameters is not valid. |
| <code>PUDS_STATUS_CAUTION_INPUT_MODIFIED</code> | Extra information, the message has been modified by the API. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

The `uds_msg` structure is automatically initialized and allocated by the PCAN-UDS 2.x API using:

- `UDS_MsgAlloc_2013` function
- UDS services functions (suffixed `UDS_Svc`)
- `UDS_Read_2013` function
- `UDS_WaitFor` functions

Once processed, the `uds_msg` structure should be released using `UDS_MsgFree_2013` function.

Example

The following example shows the use of the function `UDS_MsgAlloc_2013`. It allocates a ClearDiagnosticInformation service positive response (fixed length of one byte), with a physical configuration between ECU 1 and test equipment using in ISO15765-2 11bits normal addressing.

C/C++

```
uds_msgconfig config_physical;
memset(&config_physical, 0, sizeof(config_physical));
config_physical.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_RESPONSE_1;
config_physical.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config_physical.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config_physical.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config_physical.type = PUDS_MSGTYPE_USDT;
config_physical.nai.source_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config_physical.nai.target_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config_physical.nai.extension_addr = 0;

uds_msg response_msg;
memset(&response_msg, 0, sizeof(uds_msg));
uds_status status = UDS_MsgAlloc_2013(&response_msg, config_physical, 1);
if (UDS_StatusIsOk_2013(status, PUDS_STATUS_OK, false))
    printf("Allocate ClearDiagnosticInformation response.\n");
else
    printf("Message allocation failed.\n");
```

See also: `uds_msg` on page 21, `UDS_MsgFree_2013` on page 645.

Class-method version: `MsgAlloc_2013` on page 211.

3.8.18 UDS_MsgFree_2013

Deallocates a PUDS message.

Syntax

C/C++

```
uds_status UDS_MsgFree_2013(
    uds_msg* msg_buffer
);
```

Parameters

| Parameter | Description |
|------------|--|
| msg_buffer | An allocated uds_msg structure buffer. |


Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|--|
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The uds_msg structure is invalid. |
| <code>PUDS_STATUS_CAUTION_BUFFER_IN_USE</code> | The message structure is currently in use. It cannot be deleted. |

Example

The following example shows the use of the function `UDS_MsgFree_2013`. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the message was already allocated.

C/C++

```
uds_status result;
result = UDS_MsgFree_2013(&msg);
if (!UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Free message error\n");
else
    printf("Message released\n");
```

See also: `uds_msg` on page 21, `UDS_MsgAlloc_2013` on page 643.

Class-method version: `MsgFree_2013` on page 214.

3.8.19 UDS_MsgCopy_2013

Copies a PUDS message to another buffer.

Syntax

C/C++

```
uds_status UDS_MsgCopy_2013(
    uds_msg* msg_buffer_dst,
    const uds_msg* msg_buffer_src
);
```

Parameters

| Parameter | Description |
|----------------|---|
| msg_buffer_dst | A uds_msg structure buffer to store the copied message. |
| msg_buffer_src | The uds_msg structure to copy. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|--|
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the uds_msg structure is invalid. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

When a message is copied, a new buffer is allocated. Once processed, user has to release the source and the destination messages (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the function `UDS_MsgCopy_2013`. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the source message was already allocated.

C/C++

```
uds_msg destination;
memset(&destination, 0, sizeof(destination));
uds_status result = UDS_MsgCopy_2013(&destination, &source);
if (!UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Copy error\n");
else
    printf("Message copied\n");
```

See also: `uds_msg` on page 21.

Class-method version: `MsgCopy_2013` on page 216.

3.8.20 UDS_MsgMove_2013

Moves a PUDS message to another buffer (and cleans the original message structure).

Syntax

C/C++

```
uds_status UDS_MsgMove_2013(
    uds_msg* msg_buffer_dst,
    uds_msg* msg_buffer_src
);
```

Parameters

| Parameter | Description |
|----------------|--|
| msg_buffer_dst | A uds_msg structure buffer to store the message. |
| msg_buffer_src | The uds_msg structure buffer used as the source (will be cleaned). |

Remarks

When a message is moved, the source buffer is cleaned: once processed, user only has to release the destination message (see [UDS_MsgFree_2013](#) on page 645).


Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the <code>uds_msg</code> structure is invalid. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see UDS_MsgFree_2013 on page 645). |

Example

The following example shows the use of the function [UDS_MsgMove_2013](#). Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the source message was already allocated.

C/C++

```
uds_msg destination;
memset(&destination, 0, sizeof(destination));
uds_status result = UDS_MsgMove_2013(&destination, &source);
if (!UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Move error\n");
else
    printf("Message moved\n");
```

See also: [uds_msg](#) on page 21.

Class-method version: [MsgMove_2013](#) on page 218.

3.8.21 UDS_Read_2013

Reads a message from the receive queue of a PUDS channel.

Syntax

C

```
uds_status UDS_Read_2013(
    cantp_handle channel,
    uds_msg* out_msg_buffer,
    uds_msg* in_msg_request,
    cantp_timestamp* out_timestamp);
```

C++

```
uds_status UDS_Read_2013(
    cantp_handle channel,
    uds_msg* out_msg_buffer,
    uds_msg* in_msg_request = 0,
    cantp_timestamp* out_timestamp = 0);
```

Parameters

| Parameter | Description |
|----------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| out_msg_buffer | A uds_msg buffer to store the PUDS message (see uds_msg on page 21). |
| in_msg_request | Optional, filters read message via a request message and its network address information (see uds_msg on page 21). If NULL the first available message is fetched, otherwise in_msg_request must represent a sent PUDS request. To look for the request confirmation, in_msg_request->type should not have the loopback flag; otherwise a response from the target ECU will be searched. |
| out_timestamp | A cantp_timestamp structure buffer to get the reception time of the message. If this value is not desired, this parameter should be passed as NULL (see cantp_timestamp on page 104). |


Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---|---|
| PUDS_STATUS_NO_MESSAGE | Indicates that the receive queue of the channel is empty. |
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see UDS_MsgFree_2013 on page 645). |

Remarks

- In addition to checking `uds_status` code, the `cantp_netstatus` should be checked as it contains the network status of the message (see field `msg.msgdata.any->netstatus` in `uds_msg` on page 21).
- In case of ISO-TP message, the message type contained in the message `cantp_netaddrinfo` which indicates if the message is a complete ISO-TP message (no pending message flag) should be checked too (see field `msg.msgdata.isotp->netaddrinfo` in `uds_msg` on page 21).
- Specifying a `NULL` value for the parameter `out_timestamp` causes reading a message without timestamp when the reception time is not desired.
- The message structure is automatically allocated and initialized in `UDS_Read_2013` function. So once the message is processed, the structure must be released (see `UDS_MsgFree_2013` on page 645).

 **Note:** Higher level functions like `UDS_WaitForService_2013` or `UDS_WaitForServiceFunctional_2013` should be preferred in cases where a client just has to read the response from a service request.

Example

The following example shows the use of the function `UDS_Read_2013` on the channel `PCANTP_HANDLE_USBBUS1`. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized. This example is basic, the preferred way to read messages is Using Events (see on page 777).

C/C++

```
uds_status result;
uds_msg msg;
memset(&msg, 0, sizeof(msg));
bool stop = false;

do
{
    // Read the first message in the queue
    result = UDS_Read_2013(PCANTP_HANDLE_USBBUS1, &msg, NULL, NULL);
    if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    {
        printf("A message was received\n");

        // Process the received message
        // ProcessMessage(msg);

        result = UDS_MsgFree_2013(&msg);
        if (!UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
            printf("Message free error\n");
    }
    else
    {
        // An error occurred
        printf("An error occurred\n");
        // Here can be decided if the loop has to be terminated
        // stop = HandleReadError(result);
    }
} while (!stop);
```

See also: `UDS_Write_2013` on page 649, `uds_msg` on page 21, UUDT Read/Write Example on page 775.

Class-method version: `Read_2013` on page 219.

3.8.22 UDS_Write_2013

Transmits a message using a connected PUDS channel.

Syntax

C/C++

```
uds_status UDS_Write_2013(
    cantp_handle channel,
    uds_msg* msg_buffer);
```

Parameters

| Parameter | Description |
|------------|--|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| msg_buffer | A <code>uds_msg</code> buffer containing the message to be sent (see <code>uds_msg</code> on page 21). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of initialized channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The message is not a valid message. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_PARAM_INVALID_TYPE</code> | The message type is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The corresponding mapping is unknown. |

Remarks

The `UDS_Write_2013` function does not actually send the UDS message, the transmission is asynchronous. Should a message fail to be transmitted, it will be added to the reception queue with a specific network error code in the `msg.msgdata.any -> netstatus` value of the `uds_msg`.

Note: To transmit a standard UDS service request, it is recommended to use the corresponding API Service function starting with `UDS_Svc` (like `UDS_SvcDiagnosticSessionControl_2013`).

Example

The following example shows the use of the function `UDS_Write_2013` on the channel `PCANTP_HANDLE_USBBUS1`. It adds to the transmit queue a UDS request then waits until a confirmation message is received. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized, mapping and message configuration receive event were configured. The content of data is not initialized in the example.

C/C++

```
uds_status result;
uds_msg request_msg;
uds_msg loopback_msg;
int wait_result;

memset(&request_msg, 0, sizeof(request_msg));
memset(&loopback_msg, 0, sizeof(loopback_msg));

// Allocate and initialize message structure
result = UDS_MsgAlloc_2013(&request_msg, config, 4095);
if (!UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
{
    printf("Message initialization error: %d\n", result);
}
else
{
    // The message is sent using the PCAN-USB.
    result = UDS_Write_2013(PCANTP_HANDLE_USBBUS1, &request_msg);
    if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    {
        // Read the transmission confirmation.
        wait_result = WaitForSingleObject(receive_event, 5000);
        if (wait_result == WAIT_OBJECT_0)
        {
            result = UDS_Read_2013(PCANTP_HANDLE_USBBUS1, &loopback_msg, NULL, NULL);
            if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
            {

```

```

        printf("Read = %d, type=%d, netstatus=%d\n", result,
               loopback_msg.type, loopback_msg.msg.msgdata.any->netstatus);
        UDS_MsgFree_2013(&loopback_msg);
    }
    else
    {
        printf("Read error: %d\n", result);
    }
}
UDS_MsgFree_2013(&request_msg);
}
else
{
    printf("Write error: %d\n", result);
}
}

```

See also: [UDS_Read_2013](#) on page 648, UUDT Read/Write Example on page 775.

Class-method version: [Write_2013](#) on page 229.

3.8.23 UDS_Reset_2013

Resets the receive and transmit queues of a PUDS channel.

Syntax

C/C++

```

uds_status UDS_Reset_2013(
    cantp_handle channel);

```

Parameters

| Parameter | Description |
|-----------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |

Returns

The return value is a [uds_status](#) code. [PUDS_STATUS_OK](#) is returned on success. The typical error in case of failure is:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of initialized channels of the calling application. |
|---|---|

Remark

Calling this function ONLY clears the queues of a channel. A reset of the CAN controller does not take place.

Example

The following example shows the use of the function [UDS_Reset_2013](#) on the channel [PCANTP_HANDLE_PCIBUS1](#). Depending on the result, a message will be shown to the user

 **Note:** It is assumed that the channel was already initialized.

C/C++

```

uds_status result;

```

```
result = UDS_Reset_2013(PCANTP_HANDLE_PCIBUS1);
if (!UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("An error occurred\n");
else
    printf("PCAN-PCI (Ch-1) was reset\n");
```

See also: [UDS_Uninitialize_2013](#) on page 628.

Class-method version: [Reset_2013](#) on page 234.

3.8.24 UDS_WaitForSingleMessage_2013

Waits for a message (a response or a transmit confirmation) based on a PUDS message request.

Syntax

C/C++

```
uds_status UDS_WaitForSingleMessage_2013(
    cantp_handle channel,
    uds_msg* msg_request,
    bool is_waiting_for_tx,
    uint32_t timeout,
    uint32_t timeout_enhanced,
    uds_msg* out_msg_response);
```

Parameters

| Parameter | Description |
|-------------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| msg_request | A sent uds_msg message used as a reference to find the waited message (see uds_msg on page 21). |
| is_waiting_for_tx | States whether the message to wait for is a transmit confirmation or not. |
| timeout | Maximum time to wait (in milliseconds) for a message indication corresponding to the message request. Note: A zero value means unlimited time. |
| timeout_enhanced | Maximum time to wait (in milliseconds) for a message indication if ECU ask more time. |
| out_msg_response | A uds_msg buffer to store the response message (see uds_msg on page 21). |

Returns

The return value is a [uds_status](#) code. [PUDS_STATUS_OK](#) is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of initialized channels of the calling application. |
| PUDS_STATUS_SERVICE_TIMEOUT_CONFIRMATION | Timeout while waiting for request confirmation (request message loopback). |
| PUDS_STATUS_SERVICE_TIMEOUT_RESPONSE | Timeout while waiting for response message. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is invalid. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see UDS_MsgFree_2013 on page 645). |

Remarks

The criteria to identify if a response matches the message request is based only on the network addressing information and the UDS service identifier: if the same service is requested multiple times with different parameters (like service [ReadDataByIdentifier](#) with different data identifiers), the user will have to ensure that the extra content matches the original request.


The timeout or timeout enhanced parameters are ignored once a message indication matching the request is received (i.e. the first frame of the message).

Since the duration of a message transmission/reception depends on various factors (bitrate, data length, FD support, dlc, STmin, BS, etc.) user can call PCAN-ISO-TP function `CANTP_GetMsgProgress_2016` to follow up the progress of the communication (the `cantp_msg` parameter to use will be `out_msg_response->msg`).

The parameters of `UDS_WaitFor*_2013` functions have a new order. They do not keep the order of the previous version.

Example

The following example shows the use of the function `UDS_WaitForSingleMessage_2013` on the channel `PCANTP_HANDLE_USBBUS1`. It writes a PUDS message on the CAN Bus and waits for the confirmation of the transmission. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel and the mapping were already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg confirmation;

memset(&request, 0, sizeof(request));
memset(&confirmation, 0, sizeof(confirmation));

// Prepare an 11bit CAN ID, physically addressed UDS message containing 4 Bytes of data
uds_msgconfig config;
memset(&config, 0, sizeof(config));
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

result = UDS_MsgAlloc_2013(&request, config, 4);
if (!UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
{
    printf("Error occurred while allocating request message.\n");
}
else
{
    // The message is sent using the PCAN-USB
    result = UDS_Write_2013(PCANTP_HANDLE_USBBUS1, &request);
    if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    {
        // Wait for the transmit confirmation
        result = UDS_WaitForSingleMessage_2013(PCANTP_HANDLE_USBBUS1, &request, true, 10,
            100, &confirmation);
        if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
        {
            printf("Message was transmitted.\n");
            UDS_MsgFree_2013(&confirmation);
        }
        else
        {
            // An error occurred
            printf("Error occurred while waiting for transmit confirmation.\n");
        }
        UDS_MsgFree_2013(&request);
    }
}
```

```

    }
    else
    {
        // An error occurred
        printf("An error occurred\n");
    }
}

```

See also: [UDS_WaitForService_2013](#) on page 656, [UDS_WaitForServiceFunctional_2013](#) on page 659, [UDS_WaitForFunctionalResponses_2013](#) on page 654.

Class-method version: [WaitForSingleMessage_2013](#) on page 235.

3.8.25 UDS_WaitForFunctionalResponses_2013

Waits for multiple messages (multiple responses from a functional request for instance) based on a PUDS message request.

Syntax

C/C++

```

uds_status UDS_WaitForFunctionalResponses_2013(
    cantp_handle channel,
    uds_msg* msg_request,
    uint32_t timeout,
    uint32_t timeout_enhanced,
    bool wait_until_timeout,
    uint32_t max_msg_count,
    uds_msg* out_msg_responses,
    uint32_t* out_msg_count
);

```

Parameters

| Parameter | Description |
|--------------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| msg_request | A uds_msg containing the PUDS request message that was previously sent (see uds_msg on page 21). |
| timeout | Maximum time to wait (in milliseconds) for a message indication corresponding to the message request. Note: A zero value means unlimited time. |
| timeout_enhanced | Maximum time to wait (in milliseconds) for a message to be complete if ECU ask more time. |
| wait_until_timeout | if false the function is interrupted if out_msg_count reaches max_msg_count . |
| max_msg_count | Length of the responses buffer array (maximum messages that can be received). |
| out_msg_responses | A uds_msg buffer array to store the responses messages (see uds_msg on page 21). |
| out_msg_count | Output, number of read messages. |

Returns

The return value is a [uds_status](#) code. [PUDS_STATUS_OK](#) is returned on success. The typical errors in case of failure are:

| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of initialized channels of the calling application. |
| PUDS_STATUS_NO_MESSAGE | Indicates that no matching messages were received in the given time. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is invalid. |
| PUDS_STATUS_SERVICE_RX_OVERFLOW | Service received more messages than input buffer expected. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see UDS_MsgFree_2013 on page 645). |

Remarks


The criteria to identify if a response matches the message request is based only on the network addressing information and the UDS service identifier: if the same service is requested multiple times with different parameters (like service ReadDataByIdentifier with different data identifiers), the user will have to ensure that the extra content matches the original request.

The timeout or timeout enhanced parameters are ignored once a message indication matching the request is received (i.e. the first frame of the message). They are re-enabled when the segmented message is fully received.

The parameters of `UDS_WaitFor*_2013` functions have a new order. They do not keep the order of the previous version.

Example

The following example shows the use of the function `UDS_WaitForFunctionalResponses_2013` on the channel `PCANTP_HANDLE_USBBUS1`. It writes a UDS functional message on the CAN Bus, waits for the confirmation of the transmission, and then waits to receive responses from ECUs until a timeout occurs. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg confirmation;
uint32_t response_array_length = 5;
uds_msg response_array[5];
uint32_t count = 0;
int i;

memset(&request, 0, sizeof(request));
memset(&confirmation, 0, sizeof(confirmation));
for (i = 0; i < response_array_length; i++)
    memset(&response_array[i], 0, sizeof(uds_msg));

// prepare an 11bit CAN ID, functionally addressed UDS message
uds_msgconfig config;
memset(&config, 0, sizeof(config));
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_OBD_FUNCTIONAL;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_FUNCTIONAL;
config.type = PUDS_MSGTYPE_USDT;

result = UDS_MsgAlloc_2013(&request, config, 4);
if (!UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
{
    printf("Error occurred while allocating request message.\n");
}
else
{
    // [...] fill data (functional message is limited to 1 CAN frame)

    // The message is sent using the PCAN-USB
    result = UDS_Write_2013(PCANTP_HANDLE_USBBUS1, &request);
    if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
```

```

{
    // Wait for the transmit confirmation
    result = UDS_WaitForSingleMessage_2013(PCANTP_HANDLE_USBBUS1, &request, true, 10,
        100, &confirmation);
    if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false) &&
        confirmation.msg.msgdata.any->netstatus == PCANTP_NETSTATUS_OK)
    {
        printf("Message was transmitted.\n");
        // wait for the responses
        result = UDS_WaitForFunctionalResponses_2013(PCANTP_HANDLE_USBBUS1,
            &request, 10, 100, true, response_array_length, response_array,
            &count);
        if (count > 0)
        {
            printf("Responses were received\n");
            for (uint32_t i = 0; i < count; i++)
            {
                UDS_MsgFree_2013(&(response_array[i]));
            }
        }
        else
        {
            printf("No response was received\n");
        }
        UDS_MsgFree_2013(&confirmation);
    }
    else
    {
        // An error occurred
        printf("Error occurred while waiting for transmit confirmation.\n");
    }
    UDS_MsgFree_2013(&request);
}
else
{
    // An error occurred
    printf("An error occurred\n");
}
}
}

```

See also: [uds_msg](#) on page 21, [UDS_WaitForSingleMessage_2013](#) on page 652, [UDS_WaitForService_2013](#) on page 656, [UDS_WaitForServiceFunctional_2013](#) on page 659.

Class-method version: [WaitForFunctionalResponses_2013](#) on page 241.

3.8.26 UDS_WaitForService_2013

Handles the communication workflow for a UDS service expecting a single response. The function waits for a transmit confirmation then for a response message. Even if the [PUDS_MSGTYPE_FLAG_NO_POSITIVE_RESPONSE](#) flag is set, the function will still wait.

Syntax

C/C++

```

uds_status UDS_WaitForService_2013(
    cantp_handle channel,
    uds_msg* msg_request,
    uds_msg* out_msg_response,
    uds_msg* out_msg_request_confirmation);

```


Parameters

| Parameter | Description |
|------------------------------|--|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| msg_request | A <code>uds_msg</code> containing the PUDS request message that was previously sent (see <code>uds_msg</code> on page 21). |
| out_msg_response | A <code>uds_msg</code> buffer to store the PUDS response message (see <code>uds_msg</code> on page 21). |
| out_msg_request_confirmation | A <code>uds_msg</code> buffer to store the PUDS request confirmation message also known as loopback message (see <code>uds_msg</code> on page 21). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---|--|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of initialized channels of the calling application. |
| <code>PUDS_STATUS_SERVICE_TIMEOUT_CONFIRMATION</code> | Timeout while waiting for request confirmation (request message loopback). |
| <code>PUDS_STATUS_SERVICE_TIMEOUT_RESPONSE</code> | Timeout while waiting for response message. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is invalid. |
| <code>PUDS_STATUS_NO_MESSAGE</code> | Indicates that no matching message was received in the given time. |
| <code>PUDS_STATUS_NETWORK_ERROR</code> | A network error occurred. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

The `UDS_WaitForService_2013` function is a utility function that calls other PCAN-UDS 2.x API functions to simplify UDS communication workflow:


1. The function gets the defined timeouts (global API request and response timeouts and timeouts defined by the current session information with the ECU),
2. Waits for the confirmation of the request's transmission,
3. On success, waits for the response.
4. If a negative response code is received stating that the ECU requires extended timing (`PUDS_NRC_EXTENDED_TIMING`, 0x78), the function switches to the enhanced timeout (enhanced P2CAN server max timeout, see `uds_sessioninfo` on page 22) and waits for another response.
5. Fills the output message with response data.

Even if the `PUDS_MSGTYPE_FLAG_NO_POSITIVE_RESPONSE` flag is set in the PUDS request message, the function will still wait for an eventual Negative Response. If no error message is received the function will return `PUDS_STATUS_NO_MESSAGE`, although in this case it must not be considered as an error. Moreover, if a negative response code `PUDS_NRC_EXTENDED_TIMING` is received the `PUDS_MSGTYPE_FLAG_NO_POSITIVE_RESPONSE` flag is ignored as stated in ISO-14229-1.

The parameters of `UDS_WaitFor*_2013` functions have a new order. They do not keep the order of the previous version.

Example

The following example shows the use of the function `UDS_WaitForService_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted (service ECUReset), and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel and the mapping were already initialized.

C/C++

```

uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ECUReset message
result = UDS_SvcECUReset_2013(PCANTP_HANDLE_USBBUS1, config, &request, PUDS_SVC_PARAM_ER_SR);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
{
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
    if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    {
        printf("Response was received\n");
        UDS_MsgFree_2013(&response);
        UDS_MsgFree_2013(&request_confirmation);
    }
    else
    {
        printf("An error occurred\n");
    }
    UDS_MsgFree_2013(&request);
}
else
{
    printf("An error occurred, while sending the request.\n");
}
}

```

See also: [UDS_WaitForServiceFunctional_2013](#) on page 659.

Class-method version: [WaitForService_2013](#) on page 248.

3.8.27 UDS_WaitForServiceFunctional_2013

Handles the communication workflow for a UDS service requested with functional addressing, i.e. multiple responses can be expected. The function waits for a transmit confirmation then for responses. Even if the `PUDS_MSGTYPE_FLAG_NO_POSITIVE_RESPONSE` flag is set, the function will still wait for eventual Negative Responses.

Syntax

C/C++

```
uds_status UDS_WaitForServiceFunctional_2013(
    cantp_handle channel,
    uds_msg* msg_request,
    uint32_t max_msg_count,
    bool wait_until_timeout,
    uds_msg* out_msg_responses,
    uint32_t* out_msg_count,
    uds_msg* out_msg_request_confirmation);
```

Parameters

| Parameter | Description |
|------------------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| Msg_request | A <code>uds_msg</code> containing the PUDS request message that was previously sent (see <code>uds_msg</code> on page 21). |
| Max_msg_count | Length of the responses buffer array (maximum messages that can be received). |
| Wait_until_timeout | If false, the function is interrupted if <code>out_msg_count</code> reaches <code>max_msg_count</code> . |
| Out_msg_responses | A <code>uds_msg</code> buffer array to store the responses messages (see <code>uds_msg</code> on page 21). |
| Out_msg_count | Output, number of read messages. |
| Out_msg_request_confirmation | A <code>uds_msg</code> buffer to store the UDS request confirmation message also known as loopback message (see <code>uds_msg</code> on page 21). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---|--|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of initialized channels of the calling application. |
| <code>PUDS_STATUS_OVERFLOW</code> | Output responses buffer is too small. |
| <code>PUDS_STATUS_SERVICE_TX_ERROR</code> | An error occurred while transmitting the request. |
| <code>PUDS_STATUS_SERVICE_TIMEOUT_CONFIRMATION</code> | Timeout while waiting for request confirmation (request message loopback). |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is invalid. |
| <code>PUDS_STATUS_NO_MESSAGE</code> | Indicates that no matching message was received in the given time. |
| <code>PUDS_STATUS_NETWORK_ERROR</code> | A network error occurred. |
| <code>PUDS_STATUS_SERVICE_RX_OVERFLOW</code> | Service received more messages than input buffer expected. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

The `UDS_WaitForServiceFunctional_2013` function is a utility function that calls other PCAN-UDS 2.x API functions to simplify UDS communication workflow when requests involve functional addressing:


1. The function gets the defined timeouts (global API request and response timeouts and timeouts defined by the current session information with the ECU),
2. Waits for the confirmation of request's transmission,

3. On success, it waits for the responses.
4. The function fills the output messages array with received responses.

The parameters of `UDS_WaitFor*_2013` functions have a new order. They do not keep the order of the previous version.

Example

The following example shows the use of the function `UDS_WaitForServiceFunctional_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS functional service request is transmitted (service ECUReset), and the `UDS_WaitForServiceFunctional_2013` function is called to get the responses. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uint32_t response_array_length = 5;
uds_msg response_array[5];
uint32_t count;
uds_msgconfig config;
int i;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
for (i = 0; i < response_array_length; i++)
    memset(&response_array[i], 0, sizeof(uds_msg));

// Set request message configuration
memset(&config, 0, sizeof(config));
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_OBD_FUNCTIONAL;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_FUNCTIONAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ECUReset message
result = UDS_SvcECUReset_2013(PCANTP_HANDLE_USBBUS1, config, &request, PUDS_SVC_PARAM_ER_SR);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
{
    result = UDS_WaitForServiceFunctional_2013(PCANTP_HANDLE_USBBUS1, &request,
        response_array_length, true, response_array, &count, &request_confirmation);
    if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    {
        if (count > 0)
        {
            printf("Responses were received\n");
            for (uint32_t i = 0; i < count; i++)
            {
                UDS_MsgFree_2013(&(response_array[i]));
            }
        }
        else
        {
            printf("No response was received\n");
        }
    }
}
```

```

        }
        UDS_MsgFree_2013(&request_confirmation);
    }
    else
    {
        printf("An error occurred\n");
    }
    UDS_MsgFree_2013(&request);
}
else
{
    printf("An error occurred, while sending the request.\n");
}

```

See also: [UDS_WaitForService_2013](#) on page 656.

Class-method version: [WaitForServiceFunctional_2013](#) on page 252.

3.8.28 UDS_SvcDiagnosticSessionControl_2013

Writes a UDS request according to the DiagnosticSessionControl service's specifications. The DiagnosticSessionControl service is used to enable different diagnostic sessions in the server.

Syntax

C/C++

```

uds_status UDS_SvcDiagnosticSessionControl_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg* out_msg_request,
    uint8_t session_type);

```

Parameters

| Parameter | Description |
|-----------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |
| session_type | Subfunction parameter: type of the session (see uds_svc_param_dsc on page 71). |

Returns

The return value is a [uds_status](#) code. [PUDS_STATUS_OK](#) is returned on success. The typical errors in case of failure are:

| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with UDS_Reset_2013 (see UDS_Reset_2013 on page 651). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see UDS_MsgFree_2013 on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see [UDS_MsgFree_2013](#) on page 645).

If this service is called with the [PUDS_MSGTYPE_FLAG_NO_POSITIVE_RESPONSE](#) parameter set to ignore responses, the API will automatically change the current session to the new one. Else the session information will be updated when the response is received (only if [UDS_WaitForService_2013](#) or [UDS_WaitForServiceFunctional_2013](#) is used).

Example

The following example shows the use of the service function [UDS_SvcDiagnosticSessionControl_2013](#) on the channel [PCANTP_HANDLE_USBBUS1](#). A UDS physical service request is transmitted, and the [UDS_WaitForService_2013](#) function is called to get the response. Depending on the result, a message will be shown to the user.



Note: It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical DiagnosticSessionControl message
result = UDS_SvcDiagnosticSessionControl_2013(PCANTP_HANDLE_USBBUS1, config, &request,
    PUDS_SVC_PARAM_DSC_DS);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);
```

See also: [UDS_WaitForService_2013](#) on page 656.

Class-method version: `SvcDiagnosticSessionControl_2013` on page 258.

3.8.29 UDS_SvcECUReset_2013

Writes a UDS request according to the ECUReset service's specifications. The ECUReset service is used by the client to request a server (ECU) reset.

Syntax

C/C++

```
uds_status UDS_SvcECUReset_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t reset_type);
```

Parameters

| Parameter | Description |
|-----------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| reset_type | Subfunction parameter: type of reset (see <code>uds_svc_param_er</code> on page 72). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |


Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

After receiving a positive response to UDS request ECUReset with the help of function `UDS_WaitForService_2013`, the API will revert the session information of that ECU to the default diagnostic session.

Example

The following example shows the use of the service function `UDS_SvcECUReset_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```

uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ECUReset message
result = UDS_SvcECUReset_2013(PCANTP_HANDLE_USBBUS1, config, &request, PUDS_SVC_PARAM_ER_SR);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
    &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: [UDS_WaitForService_2013](#) on page 656.

Class-method version: [SvcECUReset_2013](#) on page 262.

3.8.30 UDS_SvcSecurityAccess_2013

Writes a UDS request according to the SecurityAccess service's specifications. SecurityAccess service provides a mean to access data and/or diagnostic services which have restricted access for security, emissions, or safety reasons.

Syntax**C/C++**

```

uds_status UDS_SvcSecurityAccess_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t security_access_type,
    uint8_t * security_access_data,
    uint32_t security_access_data_size);

```


Parameters

| Parameter | Description |
|---------------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| security_access_type | Subfunction parameter: type of security access (see SecurityAccess Type Definitions on page 765) |
| security_access_data | If Requesting Seed, buffer is the optional data to transmit to a server/ECU (like identification). If Sending Key, data holds the value generated by the security algorithm corresponding to a specific "seed" value. |
| security_access_data_size | Size in bytes of the buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_OVERFLOW</code> | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data size. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcSecurityAccess_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
```

```

config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical SecurityAccess message
uint8_t data = 0x42;
result = UDS_SvcSecurityAccess_2013(PCANTP_HANDLE_USBBUS1, config, &request,
    PUDS_SVC_PARAM_SA_RSD_3, &data, 1);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: [UDS_WaitForService_2013](#) on page 656, [SecurityAccess Type Definitions](#) on page 765.

Class-method version: [SvcSecurityAccess_2013](#) on page 267.

3.8.31 UDS_SvcCommunicationControl_2013

Writes a UDS request according to the CommunicationControl service's specifications. CommunicationControl service's purpose is to switch on/off the transmission and/or the reception of certain messages of (a) server(s)/ECU(s).

Syntax

C

```

uds_status UDS_SvcCommunicationControl_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t control_type,
    uint8_t communication_type,
    uint16_t node_identification_number);

```

C++

```

uds_status UDS_SvcCommunicationControl_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t control_type,
    uint8_t communication_type,
    uint16_t node_identification_number = 0);

```

Parameters

| Parameter | Description |
|-----------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |

| Parameter | Description |
|----------------------------|---|
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| control_type | Subfunction parameter: type of communication control (see <code>uds_svc_param_cc</code> on page 74). |
| communication_type | A bit-code value to reference the kind of communication to be controlled (see CommunicationControl Communication Type Definitions on page 766). |
| node_identification_number | A two bytes value, identify a node on a sub-network, only used with <code>PUDS_SVC_PARAM_CC_ERXDTXWEAI</code> or <code>PUDS_SVC_PARAM_CC_ERXTXWEAI</code> control type) |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcCommunicationControl_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
```

```

config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical CommunicationControl message
result = UDS_SvcCommunicationControl_2013(PCANTP_HANDLE_USBBUS1, config, &request,
    PUDS_SVC_PARAM_CC_ERXTX, PUDS_SVC_PARAM_CC_FLAG_APPL | PUDS_SVC_PARAM_CC_FLAG_NWM |
    PUDS_SVC_PARAM_CC_FLAG_DENWRIRO, 0);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: [UDS_WaitForService_2013](#) on page 656, [CommunicationControl Communication Type Definitions](#) on page 766.
Class-method version: [SvcCommunicationControl_2013](#) on page 276.

3.8.32 UDS_SvcTesterPresent_2013

Writes a UDS request according to the TesterPresent service's specifications. TesterPresent service indicates to the server(s)/ECU(s) that a client is still connected to the vehicle and that certain diagnostic services and/or communications that have been previously activated are to remain active.

Syntax

C

```

uds_status UDS_SvcTesterPresent_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t testerpresent_type);

```

C++

```

uds_status UDS_SvcTesterPresent_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t testerpresent_type = PUDS_SVC_PARAM_TP_ZSUBF);

```

Parameters

| Parameter | Description |
|--------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |
| testerpresent_type | No Subfunction parameter by default (see uds_svc_param_tp on page 75). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcTesterPresent_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request)) ;
memset(&request_confirmation, 0, sizeof(request_confirmation)) ;
memset(&response, 0, sizeof(response)) ;
memset(&config, 0, sizeof(config)) ;

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;

// Sends a physical TesterPresent message with no positive response
config.type = PUDS_MSGTYPE_USDT | PUDS_MSGTYPE_FLAG_NO_POSITIVE_RESPONSE;
result = UDS_SvcTesterPresent_2013(PCANTP_HANDLE_USBBUS1, config, &request,
    PUDS_SVC_PARAM_TP_ZSUBF);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
```

```

        printf("Response was received\n");
else if (UDS_StatusIsOk_2013(result, PUDS_STATUS_NO_MESSAGE, false))
    printf("No error response\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: `UDS_WaitForService_2013` on page 656, `uds_svc_param_tp` on page 75.

Class-method version: `SvcTesterPresent_2013` on page 285.

3.8.33 UDS_SvcSecuredDataTransmission_2013

Writes a UDS request according to the SecuredDataTransmission service's specifications(ISO-14229-1:2013). SecuredDataTransmission service's purpose is to transmit data that is protected against attacks from third parties, which could endanger data security.

Syntax

C/C++

```

uds_status UDS_SvcSecuredDataTransmission_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t * security_data_request_record,
    uint32_t security_data_request_record_size);

```

Parameters

| Parameter | Description |
|-----------------------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| Request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| Out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| Security_data_request_record | Buffer containing the data as processed by the security sub-layer (See ISO-15764). |
| Security_data_request_record_size | Size in bytes of the buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_OVERFLOW</code> | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data size. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

When using SecuredDataTransmission service, user may need to construct a `security_data_request_record` that is equal to another UDS service request definition. That is why `UDS_Svc*` functions can be called with `PUDS_ONLY_PREPARE_REQUEST` as channel identifier (instead of using `cantp_handle`): it prepares the `uds_msg` structure without sending it.

Example

The following example shows the use of the service function `UDS_SvcSecuredDataTransmission_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized. The example shows the use of a numeric buffer and points out the fact that Windows uses Little Endian Byte order and the `UDS_SvcXXX` functions expect data in Big Endian Byte Order.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uint32_t dw_buffer;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
```

```

config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Note: next a function is called to set MSB as 1st byte in the buffer
// (Win32 uses little endian format, UDS expects big endian)
uint32_t value_little_endian = 0xF0A1B2C3;
dw_buffer = Reverse32(&value_little_endian);

// Sends a physical SecuredDataTransmission message
result = UDS_SvcSecuredDataTransmission_2013(PCANTP_HANDLE_USBBUS1, config, &request,
    (uint8_t*)&dw_buffer, 4);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: `UDS_WaitForService_2013` on page 656.

Class-method version: `SvcSecuredDataTransmission_2013` on page 293.

3.8.34 UDS_SvcSecuredDataTransmission_2020

Writes a UDS request according to the SecuredDataTransmission service's specifications (ISO-14229-1:2020). SecuredDataTransmission service's purpose is to transmit data that is protected against attacks from third parties, which could endanger data security.

Syntax

C/C++

```

uds_status UDS_SvcSecuredDataTransmission_2020(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint16_t administrative_parameter,
    uint8_t signature_encryption_calculation,
    uint16_t anti_replay_counter,
    uint8_t internal_service_identfier,
    uint8_t *service_specific_parameters,
    uint32_t service_specific_parameters_size,
    uint8_t *signature_mac,
    uint16_t signature_size);

```

Parameters

| Parameter | Description |
|-----------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |

| Parameter | Description |
|----------------------------------|--|
| administrative_parameter | Security features used in the message (see SecuredDataTransmission Administrative Parameter Flags Definitions on page 767) |
| signature_encryption_calculation | Signature or encryption algorithm identifier. |
| anti_replay_counter | Anti-replay counter value. |
| internal_service_identifier | Internal message service request identifier (see uds_service on page 53). |
| service_specific_parameters | Buffer that contains internal message service request data. |
| service_specific_parameters_size | Internal message service request data size (in bytes). |
| signature_mac | Buffer that contains signature used to verify the message. |
| signature_size | Size in bytes of the signature. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_OVERFLOW</code> | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data size. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |


Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

When using `SecuredDataTransmission` service, user may need to construct a `security_data_request_record` that is equal to another UDS service request definition. That is why `UDS_Svc*` functions can be called with `PUDS_ONLY_PREPARE_REQUEST` as channel identifier (instead of using `cantp_handle`): it prepares the `uds_msg` structure without sending it.

Example

The following example shows the use of the service function `UDS_SvcSecuredDataTransmission_2020` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;
uint16_t administrative_parameter;
uint8_t signature_encryption_calculation;
```

```

uint16_t anti_replay_counter;
uint8_t internal_service_identifier;
uint8_t service_specific_parameters[4];
uint32_t service_specific_parameters_size;
uint8_t signature_mac[6];
uint16_t signature_size;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical SecuredDataTransmission message
administrative_parameter = PUDS_SVC_PARAM_APAR_REQUEST_MSG_FLAG |
    PUDS_SVC_PARAM_APAR_REQUEST_RESPONSE_SIGNATURE_FLAG |
    PUDS_SVC_PARAM_APAR_SIGNED_MSG_FLAG;
signature_encryption_calculation = 0x0;
anti_replay_counter = 0x0124;
internal_service_identifier = 0x2E;
service_specific_parameters[0] = 0xF1;
service_specific_parameters[1] = 0x23;
service_specific_parameters[2] = 0xAA;
service_specific_parameters[3] = 0x55;
service_specific_parameters_size = 4;
signature_mac[0] = 0xDB;
signature_mac[1] = 0xD1;
signature_mac[2] = 0x0E;
signature_mac[3] = 0xDC;
signature_mac[4] = 0x55;
signature_mac[5] = 0xAA;
signature_size = 0x0006;
result = UDS_SvcSecuredDataTransmission_2020(PCANTP_HANDLE_USBBUS1, config, &request,
    administrative_parameter, signature_encryption_calculation, anti_replay_counter,
    internal_service_identifier, service_specific_parameters,
    service_specific_parameters_size, signature_mac, signature_size);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: SecuredDataTransmission Administrative Parameter Flags Definitions on page 767,
[UDS_WaitForService_2013](#) on page 656.

Class-method version: [SvcSecuredDataTransmission_2020](#) on page 298.

3.8.35 UDS_SvcControlDTCSetting_2013

Writes a UDS request according to the ControlDTCSetting service's specifications. ControlDTCSetting service shall be used by a client to stop or resume the setting of diagnostic trouble codes (DTCs) in the server(s).

Syntax

C/C++

```
uds_status UDS_SvcControlDTCSetting_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t dtc_setting_type,
    uint8_t * dtc_setting_control_option_record,
    uint32_t dtc_setting_control_option_record_size);
```

Parameters

| Parameter | Description |
|--|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| Request_config | Message request configuration (see uds_msgconfig on page 27). |
| Out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |
| Dtc_setting_type | Subfunction parameter (see uds_svc_param_cdtcs on page 76). |
| Dtc_setting_control_option_record | This parameter record is user-optional and transmits data to a server (ECU) when controlling the DTC setting. It can contain a list of DTCs to be turned on or off. |
| Dtc_setting_control_option_record_size | Size in bytes of the buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is not valid. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | The mapping given in the message configuration is unknown. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_OVERFLOW | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data size. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcControlDTCSetting_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized. The example shows the use of a numeric buffer and points out the fact that Windows uses Little Endian Byte order and the `UDS_SvcXXX` functions expect data in Big Endian Byte Order.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uint32_t dw_buffer;
uds_msgconfig config;

memset(&request, 0, sizeof(request)) ;
memset(&request_confirmation, 0, sizeof(request_confirmation)) ;
memset(&response, 0, sizeof(response)) ;
memset(&config, 0, sizeof(config)) ;

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Note: next a function is called to set MSB as 1st byte in the buffer
// (Win32 uses little endian format, UDS expects big endian)
uint32_t value_little_endian = 0xF0A1B2C3;
dw_buffer = Reverse32(&value_little_endian);
// Sends a physical ControlDTCSetting message
result = UDS_SvcControlDTCSetting_2013(PCANTP_HANDLE_USBBUS1, config, &request,
    PUDS_SVC_PARAM_CDTCS_OFF, (uint8_t*)&dw_buffer, 3);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);
```

See also: `UDS_WaitForService_2013` on page 656, `uds_svc_param_cdtcs` on page 76.

Class-method version: `SvcControlDTCSetting_2013` on page 305.

3.8.36 UDS_SvcResponseOnEvent_2013

Writes a UDS request according to the ResponseOnEvent service's specifications. The ResponseOnEvent service requests a server (ECU) to start or stop transmission of responses on a specified event.

Syntax

C

```
uds_status UDS_SvcResponseOnEvent_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t event_type,
    bool store_event,
    uint8_t event_window_time,
    uint8_t * event_type_record,
    uint32_t event_type_record_size,
    uint8_t * service_to_respond_to_record,
    uint32_t service_to_respond_to_record_size);
```

C++

```
uds_status UDS_SvcResponseOnEvent_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t event_type,
    bool store_event,
    uint8_t event_window_time,
    uint8_t * event_type_record = nullptr,
    uint32_t event_type_record_size = 0,
    uint8_t * service_to_respond_to_record = nullptr,
    uint32_t service_to_respond_to_record_size = 0);
```

Parameters

| Parameter | Description |
|-----------------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |
| event_type | Subfunction parameter: event type (see uds_svc_param_roe on page 77). |
| store_event | Storage state (true to store event, false to do not store event). |
| event_window_time | Specify a window for the event logic to be active in the server/ECU (see ResponseOnEvent Service Definitions on page 766). |
| event_type_record | Additional parameters for the specified event type. |
| event_type_record_size | Size in bytes of the event type record (see ResponseOnEvent Service Definitions on page 766). |
| service_to_respond_to_record | Service parameters, with first byte as service identifier (see uds_svc_param_roe_recommended_service_id on page 78) |
| service_to_respond_to_record_size | Size in bytes of the service to respond to record. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is not valid. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | The mapping given in the message configuration is unknown. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with UDS_Reset_2013 (see UDS_Reset_2013 on page 651). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_OVERFLOW | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data size. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see UDS_MsgFree_2013 on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see [UDS_MsgFree_2013](#) on page 645).

Example

The following example shows the use of the service function [UDS_SvcResponseOnEvent_2013](#) on the channel [PCANTP_HANDLE_USBBUS1](#). A UDS physical service request is transmitted, and the [UDS_WaitForService_2013](#) function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uint8_t event_type_record[5];
uint8_t service_to_respond_to_record[5];
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ResponseOnEvent message
event_type_record[0] = PUDS_SVC_PARAM_RDTCTI_RNODTCBSM;
event_type_record[1] = 0x01;
service_to_respond_to_record[0] = 0x08;
service_to_respond_to_record[1] = PUDS_SI_ReadDTCInformation;
result = UDS_SvcResponseOnEvent_2013(PCANTP_HANDLE_USBBUS1, config, &request,
                                     PUDS_SVC_PARAM_ROE_ONDTCS, false, 0x08, event_type_record, 2,
```

```

        service_to_respond_to_record, 2);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: [UDS_WaitForService_2013](#) on page 656, [ResponseOnEvent Service Definitions](#) on page 766.

Class-method version: [SvcResponseOnEvent_2013](#) on page 314.

3.8.37 UDS_SvcLinkControl_2013

Writes a UDS request according to the LinkControl service's specifications. The LinkControl service is used to control the communication link baud rate between the client and the server(s)/ECU(s) for the exchange of diagnostic data.

Syntax

C

```

uds_status UDS_SvcLinkControl_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t link_control_type,
    uint8_t baudrate_identifier,
    uint32_t link_baudrate);

```

C++

```

uds_status UDS_SvcLinkControl_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t link_control_type,
    uint8_t baudrate_identifier,
    uint32_t link_baudrate = 0);

```

Parameters

| Parameter | Description |
|---------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |
| link_control_type | Subfunction parameter: link control type (see uds_svc_param_lc on page 79). |
| baudrate_identifier | Defined baud rate identifier (see uds_svc_param_lc_baudrate_identifier on page 80). |
| link_baudrate | Used only with PUDS_SVC_PARAM_LC_VBTWSBR parameter: a three-byte value baud rate (baud rate High, Middle and Low Bytes). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcLinkControl_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical LinkControl message (Verify Fixed Baudrate)
result = UDS_SvcLinkControl_2013(PCANTP_HANDLE_USBBUS1, config, &request,
    PUDS_SVC_PARAM_LC_VBTWFBF, PUDS_SVC_PARAM_LC_BAUDRATE_CAN_250K, 0);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
```



```
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);
```

See also: [UDS_WaitForService_2013](#) on page 656.

Class-method version: [SvcLinkControl_2013](#) on page 328.

3.8.38 UDS_SvcReadDataByIdentifier_2013

Writes a UDS request according to the ReadDataByIdentifier service's specifications. The ReadDataByIdentifier service allows the client to request data record values from the server (ECU) identified by one or more data identifiers.

Syntax

C/C++

```
uds_status UDS_SvcReadDataByIdentifier_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint16_t * data_identifier,
    uint32_t data_identifier_length);
```

Parameters

| Parameter | Description |
|------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |
| data_identifier | Buffer containing a list of two-byte data identifiers (see uds_svc_param_di on page 82). |
| data_identifier_length | Number of elements in the buffer (size in UInt16 of the buffer). |

Returns

The return value is a [uds_status](#) code. [PUDS_STATUS_OK](#) is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is not valid. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | The mapping given in the message configuration is unknown. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with UDS_Reset_2013 (see UDS_Reset_2013 on page 651). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_OVERFLOW | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data length. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see UDS_MsgFree_2013 on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see [UDS_MsgFree_2013](#) on page 645).

Example

The following example shows the use of the service function [UDS_SvcReadDataByIdentifier_2013](#) on the channel [PCANTP_HANDLE_USBBUS1](#). A UDS physical service request is transmitted, and the [UDS_WaitForService_2013](#) function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDataByIdentifier message
uint16_t buffer[2] = { PUDS_SVC_PARAM_DI_ADSDID, PUDS_SVC_PARAM_DI_ECUMDDID };
result = UDS_SvcReadDataByIdentifier_2013(PCANTP_HANDLE_USBBUS1, config, &request, buffer, 2);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
    &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);
```

See also: [UDS_WaitForService_2013](#) on page 656, [uds_svc_param_di](#) on page 82.

Class-method version: [SvcReadDataByIdentifier_2013](#) on page 337.

3.8.39 UDS_SvcReadMemoryByAddress_2013

Writes a UDS request according to the ReadMemoryByAddress service's specifications. The ReadMemoryByAddress service allows the client to request memory data from the server (ECU) via a provided starting address and to specify the size of memory to be read.

Syntax

C/C++

```
uds_status UDS_SvcReadMemoryByAddress_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t * memory_address_buffer,
    uint8_t memory_address_size,
    uint8_t * memory_size_buffer,
    uint8_t memory_size_size);
```

Parameters

| Parameter | Description |
|-----------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |
| memory_address_buffer | Starting address of server (ECU) memory from which data is to be retrieved. |
| memory_address_size | Size in bytes of the memory address buffer (max.: 0xF). |
| memory_size_buffer | Number of bytes to be read starting at the address specified by memory address. |
| memory_size_size | Size in bytes of the memory size buffer (max.: 0xF). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is not valid. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | The mapping given in the message configuration is unknown. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with UDS_Reset_2013 (see UDS_Reset_2013 on page 651). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see UDS_MsgFree_2013 on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcReadMemoryByAddress_2013` on the channel `PCANTP_HANDLE_USBUS1`. A UDS physical service request is transmitted, and the

`UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.



Note: It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uint8_t memory_address_buffer[10];
uint8_t memory_size_buffer[10];
uint8_t memory_address_size = 10;
uint8_t memory_size_size = 3;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Fill data
for (int i = 0; i < memory_address_size; i++)
{
    memory_address_buffer[i] = 'A' + i;
    memory_size_buffer[i] = '1' + i;
}

// Sends a physical ReadMemoryByAddress message
result = UDS_SvcReadMemoryByAddress_2013(PCANTP_HANDLE_USBBUS1, config, &request,
    memory_address_buffer, memory_address_size, memory_size_buffer, memory_size_size);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);
```

See also: `UDS_WaitForService_2013` on page 656.

Class-method version: `SvcReadMemoryByAddress_2013` on page 342.

3.8.40 UDS_SvcReadScalingDataByIdentifier_2013

Writes a UDS request according to the ReadScalingDataByIdentifier service's specifications. The ReadScalingDataByIdentifier service allows the client to request scaling data record information from the server (ECU) identified by a data identifier.

Syntax

C/C++

```
uds_status UDS_SvcReadScalingDataByIdentifier_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint16_t data_identifier);
```

Parameters

| Parameter | Description |
|-----------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |
| data_identifier | A two-byte data identifier (see uds_svc_param_di on page 82). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with UDS_Reset_2013 (see UDS_Reset_2013 on page 651). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see UDS_MsgFree_2013 on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcReadScalingDataByIdentifier_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```

uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadScalingDataByIdentifier message
result = UDS_SvcReadScalingDataByIdentifier_2013(PCANTP_HANDLE_USBBUS1, config, &request,
    PUDS_SVC_PARAM_DI_BSFPDID);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: [UDS_WaitForService_2013](#) on page 656, [uds_svc_param_di](#) on page 82.

Class-method version: [SvcReadScalingDataByIdentifier_2013](#) on page 347.

3.8.41 UDS_SvcReadDataByPeriodicIdentifier_2013

Writes a UDS request according to the ReadDataByPeriodicIdentifier service's specifications. The ReadDataByPeriodicIdentifier service allows the client to request the periodic transmission of data record values from the server (ECU) identified by one or more periodic data identifiers.

Syntax**C/C++**

```

uds_status UDS_SvcReadDataByPeriodicIdentifier_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t transmission_mode,
    uint8_t *periodic_data_identifier,
    uint32_t periodic_data_identifier_length);

```

Parameters

| Parameter | Description |
|---------------------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| transmission_mode | Transmission rate mode (see <code>uds_svc_param_rdbpi</code> on page 86). |
| periodic_data_identifier | Buffer containing a list of periodic data identifiers (see <code>uds_svc_param_di</code> on page 82). |
| periodic_data_identifier_length | Number of elements in the buffer (size in bytes of the buffer). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_OVERFLOW</code> | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data length. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcReadDataByPeriodicIdentifier_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uint8_t periodic_data_identifier[10];
uint16_t periodic_data_identifier_size = 10;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
```

```

config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Fill Data
for (int i = 0; i < periodic_data_identifier_size; i++)
{
    periodic_data_identifier[i] = 'A' + i;
}

// Sends a physical ReadDataByPeriodicIdentifier message
result = UDS_SvcReadDataByPeriodicIdentifier_2013(PCANTP_HANDLE_USBBUS1, config, &request,
    PUDS_SVC_PARAM_RDBPI_SAMR, periodic_data_identifier, periodic_data_identifier_size);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: `UDS_WaitForService_2013` on page 656, `uds_svc_param_di` on page 82.

Class-method version: `SvcReadDataByPeriodicIdentifier_2013` on page 351.

3.8.42 UDS_SvcDynamicallyDefineDataIdentifierDBID_2013

Writes a UDS request according to the DynamicallyDefineDataIdentifier service's specifications. The DynamicallyDefineDataIdentifier service allows the client to dynamically define in a server (ECU) a data identifier that can be read via the ReadDataByIdentifier service later. The Define by identifier subfunction specifies that the definition of the dynamic data identifier shall occur via a data identifier reference.

Syntax

C/C++

```

uds_status UDS_SvcDynamicallyDefineDataIdentifierDBID_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint16_t dynamically_defined_data_identifier,
    uint16_t * source_data_identifier,
    uint8_t * memory_size,
    uint8_t * position_in_source_data_record,
    uint32_t number_of_elements);

```

Parameters

| Parameter | Description |
|-----------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |

| Parameter | Description |
|-------------------------------------|---|
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| dynamically_defined_data_identifier | A two-byte data identifier (see <code>uds_svc_param_di</code> on page 82). |
| source_data_identifier | Buffer containing the sources of information to be included into the dynamic data record. |
| memory_size | Buffer containing the total numbers of bytes from the source data record address. |
| position_in_source_data_record | Buffer containing the starting byte positions of the excerpt of the source data record. |
| number_of_elements | Number of <code>source_data_identifier</code> / <code>position_in_source_data_record</code> / <code>memory_size</code> triplet. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_OVERFLOW</code> | The total buffer length is too big. The resulting UDS message data size is bigger than the maximum data length. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcDynamicallyDefineDataIdentifierDBID_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uint16_t source_data_identifier[10];
uint8_t memory_size[10];
uint8_t position_in_source_data_record[10];
uint16_t number_of_elements = 10;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
```

```

config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Fill Data
for (int i = 0; i < number_of_elements; i++)
{
    source_data_identifier[i] = ((0xF0 + i) << 8) + ('A' + i);
    memory_size[i] = i + 1;
    position_in_source_data_record[i] = 100 + i;
}

// Sends a physical DynamicallyDefineDataIdentifierDBID message
result = UDS_SvcDynamicallyDefineDataIdentifierDBID_2013(PCANTP_HANDLE_USBBUS1, config,
    &request, PUDS_SVC_PARAM_DI_CDDID, source_data_identifier, memory_size,
    position_in_source_data_record, number_of_elements);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: [UDS_WaitForService_2013](#) on page 656.

Class-method version: [SvcDynamicallyDefineDataIdentifierDBID_2013](#) on page 356.

3.8.43 UDS_SvcDynamicallyDefineDataIdentifierDBMA_2013

Writes a UDS request according to the DynamicallyDefineDataIdentifier service's specifications. The DynamicallyDefineDataIdentifier service allows the client to dynamically define in a server (ECU) a data identifier that can be read via the ReadDataByIdentifier service later. The define by memory Address subfunction specifies that the definition of the dynamic data identifier shall occur via an address reference.

Syntax

C/C++

```

uds_status UDS_SvcDynamicallyDefineDataIdentifierDBMA_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint16_t dynamically_defined_data_identifier,
    uint8_t memory_address_size,
    uint8_t memory_size_size,
    uint8_t * memory_address_buffer,
    uint8_t * Memory_size_buffer,
    uint32_t number_of_elements);

```

Parameters

| Parameter | Description |
|-------------------------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| dynamically_defined_data_identifier | A two-byte data identifier (see <code>uds_svc_param_di</code> on page 82). |
| memory_address_size | Size in bytes of the memory address items in the memory address buffer (max.: 0xF). |
| memory_size_size | Size in bytes of the memory size items in the memory size buffer (max.: 0xF). |
| memory_address_buffer | buffer containing the MemoryAddress buffer, must be an array of 'number_of_elements' items whose size is 'memory_address_size' (size is 'number_of_elements * memory_address_size' bytes) |
| memory_size_buffer | buffer containing the MemorySize buffer, must be an array of 'number_of_elements' items whose size is 'memory_size_size' (size is 'number_of_elements * memory_size_size' bytes) |
| number_of_elements | Number of memory address/memory_size couple. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_OVERFLOW</code> | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data length. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcDynamicallyDefineDataIdentifierDBMA_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uint16_t number_of_elements = 3;
uint8_t memory_address_buffer[15];
uint8_t memory_size_buffer[9];
uint8_t memory_address_size = 5;
uint8_t memory_size_size = 3;
```

```

uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Fill data
for (int j = 0; j < number_of_elements; j++)
{
    for (int i = 0; i < memory_address_size; i++)
    {
        memory_address_buffer[memory_address_size*j + i] = (10 * j) + i + 1;
    }
    for (int i = 0; i < memory_size_size; i++)
    {
        memory_size_buffer[memory_size_size*j + i] = 100 + (10 * j) + i + 1;
    }
}

// Sends a physical DynamicallyDefineDataIdentifierDBMA message
result = UDS_SvcDynamicallyDefineDataIdentifierDBMA_2013(PCANTP_HANDLE_USBBUS1, config,
    &request, PUDS_SVC_PARAM_DI_CESWNDID, memory_address_size, memory_size_size,
    memory_address_buffer, memory_size_buffer, number_of_elements);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: [UDS_WaitForService_2013](#) on page 656, [uds_svc_param_di](#) on page 82.

Class-method version: [SvcDynamicallyDefineDataIdentifierDBMA_2013](#) on page 362.

3.8.44 UDS_SvcDynamicallyDefineDataIdentifierCDDDI_2013

Writes a UDS request according to the Clear Dynamically Defined Data Identifier service's specifications. The Clear Dynamically Defined Data Identifier subfunction shall be used to clear the specified dynamic data identifier.

Syntax

C/C++

```
uds_status UDS_SvcDynamicallyDefineDataIdentifierCDDDI_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint16_t dynamically_defined_data_identifier);
```

Parameters

| Parameter | Description |
|-------------------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |
| dynamically_defined_data_identifier | A two-byte data identifier (see uds_svc_param_di on page 82). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcDynamicallyDefineDataIdentifierCDDDI_2013` on the channel `PCANTP_HANDLE_USBUSB1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
```

```

uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical DynamicallyDefineDataIdentifierCDDDI message
result = UDS_SvcDynamicallyDefineDataIdentifierCDDDI_2013(PCANTP_HANDLE_USBBUS1, config,
    &request, PUDS_SVC_PARAM_DI_CESWNDID);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: [UDS_WaitForService_2013](#) on page 656, [uds_svc_param_di](#) on page 82.

Class-method version: [SvcDynamicallyDefineDataIdentifierCDDDI_2013](#) on page 368.

3.8.45 UDS_SvcDynamicallyDefineDataIdentifierClearAllDDDI_2013

Writes a UDS request according to the Clear all Dynamically Defined Data Identifier service's specifications. The Clear Dynamically Defined Data Identifier subfunction (without data identifier parameter) shall be used to clear all the specified dynamic data identifier in the server.

Syntax

C/C++

```

uds_status UDS_SvcDynamicallyDefineDataIdentifierClearAllDDDI_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg* out_msg_request);

```

Parameters

| Parameter | Description |
|----------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |

| Parameter | Description |
|-----------------|--|
| out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcDynamicallyDefineDataIdentifierClearAllDDDI_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical DynamicallyDefineDataIdentifierClearAllDDDI message
result = UDS_SvcDynamicallyDefineDataIdentifierClearAllDDDI_2013(PCANTP_HANDLE_USBBUS1, config,
    &request);
```

```

if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
    &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);

UDS_MsgFree_2013(&request_confirmation);

```

See also: `UDS_WaitForService_2013` on page 656.

Class-method version: `SvcDynamicallyDefineDataIdentifierClearAllDDDI_2013` on page 372.

3.8.46 UDS_SvcWriteDataByIdentifier_2013

Writes a UDS request according to the WriteDataByIdentifier service's specifications. The WriteDataByIdentifier service allows the client to write information into the server (ECU) at an internal location specified by the provided data identifier.

Syntax

C/C++

```

uds_status UDS_SvcWriteDataByIdentifier_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint16_t data_identifier,
    uint8_t * data_record,
    uint32_t data_record_size);

```

Parameters

| Parameter | Description |
|------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| data_identifier | A two-byte data identifier (see <code>uds_svc_param_di</code> on page 82). |
| data_record | Buffer containing the data to write. |
| data_record_size | Size in bytes of the buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is not valid. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | The mapping given in the message configuration is unknown. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_OVERFLOW | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data length. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcWriteDataByIdentifier_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uint8_t data_record[10];
uint16_t data_record_size = 10;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Fill Data
for (int i = 0; i < data_record_size; i++)
{
    data_record[i] = 'A' + i;
}

// Sends a physical WriteDataByIdentifier message
```

```

result = UDS_SvcWriteDataByIdentifier_2013(PCANTP_HANDLE_USBBUS1, config, &request,
    PUDS_SVC_PARAM_DI_ASFDPID, data_record, data_record_size);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: [UDS_WaitForService_2013](#) on page 656, [uds_svc_param_di](#) on page 82.

Class-method version: [SvcWriteDataByIdentifier_2013](#) on page 376.

3.8.47 UDS_SvcWriteMemoryByAddress_2013

Writes a UDS request according to the WriteMemoryByAddress service's specifications. The WriteMemoryByAddress service allows the client to write information into the server (ECU) at one or more contiguous memory locations.

Syntax

C/C++

```

uds_status UDS_SvcWriteMemoryByAddress_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t * memory_address_buffer,
    uint8_t memory_address_size,
    uint8_t * memory_size_buffer,
    uint8_t memory_size_size,
    uint8_t * data_record,
    uint32_t data_record_size);

```

Parameters

| Parameter | Description |
|-----------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |
| memory_address_buffer | Starting address buffer of server (ECU) memory to which data is to be written. |
| memory_address_size | Size in bytes of the memory address buffer (max.: 0xF). |
| memory_size_buffer | Number of bytes to be written starting at the address specified by memory address. |
| memory_size_size | Size in bytes of the memory size buffer (max.: 0xF). |
| data_record | Buffer containing the data to write. |
| data_record_size | Size in bytes of the buffer. |

Returns

The return value is a [uds_status](#) code. [PUDS_STATUS_OK](#) is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is not valid. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | The mapping given in the message configuration is unknown. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with UDS_Reset_2013 (see UDS_Reset_2013 on page 651). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_OVERFLOW | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data length. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see UDS_MsgFree_2013 on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see [UDS_MsgFree_2013](#) on page 645).

Example

The following example shows the use of the service function [UDS_SvcWriteMemoryByAddress_2013](#) on the channel [PCANTP_HANDLE_USBBUS1](#). A UDS physical service request is transmitted, and the [UDS_WaitForService_2013](#) function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uint8_t data_record[50];
uint8_t memory_address_buffer[50];
uint8_t memory_size_buffer[50];
uint16_t data_record_size = 50;
uint8_t memory_address_size = 5;
uint8_t memory_size_size = 3;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Fill Data
for (int i = 0; i < data_record_size; i++)
{
```

```

    data_record[i] = i + 1;
    memory_address_buffer[i] = 'A' + i;
    memory_size_buffer[i] = 10 + i;
}

// Sends a physical WriteMemoryByAddress message
result = UDS_SvcWriteMemoryByAddress_2013(PCANTP_HANDLE_USBBUS1, config, &request,
    memory_address_buffer, memory_address_size, memory_size_buffer, memory_size_size,
    data_record, data_record_size);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: [UDS_WaitForService_2013](#) on page 656.

Class-method version: [SvcWriteMemoryByAddress_2013](#) on page 381.

3.8.48 UDS_SvcClearDiagnosticInformation_2013

Writes a UDS request according to the ClearDiagnosticInformation service's specifications. The ClearDiagnosticInformation service is used by the client to clear diagnostic information in one server's or multiple servers' memory.

Syntax

C/C++

```

uds_status UDS_SvcClearDiagnosticInformation_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint32_t group_of_dtc);

```

Parameters

| Parameter | Description |
|-----------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |
| group_of_dtc | A three-byte value indicating the group of DTCs (e.g. powertrain, body, chassis) or the particular DTC to be cleared (see ClearDiagnosticInformation Group of DTC Definitions on page 767). |

Returns

The return value is a [uds_status](#) code. [PUDS_STATUS_OK](#) is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcClearDiagnosticInformation_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ClearDiagnosticInformation message
result = UDS_SvcClearDiagnosticInformation_2013(PCANTP_HANDLE_USBBUS1, config, &request,
0xF1A2B3);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
&request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");
```

```
// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);
```

See also: `UDS_WaitForService_2013` on page 656, `ClearDiagnosticInformation` Group of DTC Definitions on page 767, `UDS_SvcClearDiagnosticInformation_2020` on page 702.

Class-method version: `SvcClearDiagnosticInformation_2013` on page 387.

3.8.49 UDS_SvcClearDiagnosticInformation_2020

Writes a UDS request according to the `ClearDiagnosticInformation` service's specifications with memory selection parameter (ISO-14229-1:2020). The `ClearDiagnosticInformation` service is used by the client to clear diagnostic information in one server's or multiple servers' memory.

Syntax

C/C++

```
uds_status UDS_SvcClearDiagnosticInformation_2020(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg* out_msg_request,
    uint32_t group_of_dtc,
    uint8_t memory_selection);
```

Parameters

| Parameter | Description |
|------------------|--|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| group_of_dtc | A three-byte value indicating the group of DTCs (e.g. powertrain, body, chassis) or the particular DTC to be cleared (see <code>ClearDiagnosticInformation</code> Group of DTC Definitions on page 767). |
| memory_selection | User defined DTC memory. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcClearDiagnosticInformation_2020` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ClearDiagnosticInformation message
result = UDS_SvcClearDiagnosticInformation_2020(PCANTP_HANDLE_USBBUS1, config, &request,
0xF1A2B3, 0x42);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
&request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);
```

See also: `UDS_WaitForService_2013` on page 656, `ClearDiagnosticInformation Group of DTC Definitions` on page 767, `UDS_SvcClearDiagnosticInformation_2013` on page 700.

Class-method version: `SvcClearDiagnosticInformation_2020` on page 391.

3.8.50 UDS_SvcReadDTCInformation_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information.

Syntax

C/C++

```
uds_status UDS_SvcReadDTCInformation_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t PUDS_SVC_PARAM_RDTCI_Type,
    uint8_t dtc_status_mask);
```

Parameters

| Parameter | Description |
|---------------------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |
| PUDS_SVC_PARAM_RDTCI_Type | Subfunction parameter, ReadDTCInformation type, use one of the following: PUDS_SVC_PARAM_RDTCI_RNODTCBSM, PUDS_SVC_PARAM_RDTCI_RDTCSM, PUDS_SVC_PARAM_RDTCI_RMMDTCSM, PUDS_SVC_PARAM_RDTCI_RNOMMDTCBSM, PUDS_SVC_PARAM_RDTCI_RNOOBDDTCBSM, PUDS_SVC_PARAM_RDTCI_ROBDDTCBSM. See also uds_svc_param_rdtci on page 88. |
| dtc_status_mask | Contains eight DTC status bits. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with UDS_Reset_2013 (see UDS_Reset_2013 on page 651). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see UDS_MsgFree_2013 on page 645). |


Remarks

- This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).
- Only the following subfunctions are allowed:
 - reportNumberOfDTCByStatusMask,
 - reportDTCByStatusMask,
 - reportMirrorMemoryDTCByStatusMask,
 - reportNumberOfMirrorMemoryDTCByStatusMask,

- reportNumberOfEmissionsRelatedOBDDTCByStatusMask,
- reportEmissionsRelatedOBDDTCByStatusMask.

Example

The following example shows the use of the service function `UDS_SvcReadDTCInformation_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformation message
result = UDS_SvcReadDTCInformation_2013(PCANTP_HANDLE_USBBUS1, config, &request,
    PUDS_SVC_PARAM_RDTCTI_RDTCSBM, 0xF0);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);
```

See also: `UDS_WaitForService_2013` on page 656, `uds_svc_param_rdtcti` on page 88.

Class-method version: `SvcReadDTCInformation_2013` on page 396.

3.8.51 UDS_SvcReadDTCInformationRDTCSBDTC_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. The subfunction `reportDTCSnapshotRecordByDTCNumber` (`PUDS_SVC_PARAM_RDTCL_RDTCSBDTC`) is implicit.

Syntax

C/C++

```
uds_status UDS_SvcReadDTCInformationRDTCSBDTC_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint32_t dtc_mask,
    uint8_t dtc_snapshot_record_number);
```

Parameters

| Parameter | Description |
|----------------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| dtc_mask | A unique identification number (three-byte value) for a specific diagnostic trouble code. |
| dtc_snapshot_record_number | The number of the specific DTCSnapshot data records. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcReadDTCInformationRDTCSBDTC_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```

uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRDTCSBDTC message
result = UDS_SvcReadDTCInformationRDTCSBDTC_2013(PCANTP_HANDLE_USBBUS1, config, &request,
0x00A1B2B3, 0x12);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
&request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: [UDS_WaitForService_2013](#) on page 656.

Class-method version: [SvcReadDTCInformationRDTCSBDTC_2013](#) on page 400.

3.8.52 UDS_SvcReadDTCInformationRDTCSBRN_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. The subfunction reportDTCSnapshotByRecordNumber ([PUDS_SVC_PARAM_RDTCI_RDTCSBRN](#)) is implicit.

Syntax

C/C++

```

uds_status UDS_SvcReadDTCInformationRDTCSBRN_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t dtc_snapshot_record_number);

```

Parameters

| Parameter | Description |
|----------------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| dtc_snapshot_record_number | The number of the specific DTCSnapshot data records. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcReadDTCInformationRDTCSBURN_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
```

```

config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRDTCSB RN message
result = UDS_SvcReadDTCInformationRDTCSB RN_2013(PCANTP_HANDLE_USBBUS1, config, &request,
0x12);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
&request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: [UDS_WaitForService_2013](#) on page 656.

Class-method version: [SvcReadDTCInformationRDTCSB RN_2013](#) on page 404.

3.8.53 UDS_SvcReadDTCInformationReportExtended_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. Only [reportDTCExtendedDataRecordByDTCNumber](#) and [reportMirrorMemoryDTCExtendedDataRecordByDTCNumber](#) subfunctions are allowed.

Syntax

C/C++

```

uds_status UDS_SvcReadDTCInformationReportExtended_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t PUDS_SVC_PARAM_RDTCI_Type,
    uint32_t dtc_mask,
    uint8_t dtc_extended_data_record_number);

```

Parameters

| Parameter | Description |
|---------------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |
| PUDS_SVC_PARAM_RDTCI_Type | Subfunction parameter, ReadDTCInformation type, use one of the following: PUDS_SVC_PARAM_RDTCI_RDTCEDRBDN, PUDS_SVC_PARAM_RDTCI_RMMDEDRBDN. See also uds_svc_param_rdtci on page 88. |
| dtc_mask | A unique identification number (three-byte value) for a specific diagnostic trouble code. |
| dtc_extended_data_record_number | The number of the specific DTCExtendedData record requested. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcReadDTCInformationReportExtended_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationReportExtended message
result = UDS_SvcReadDTCInformationReportExtended_2013(PCANTP_HANDLE_USBBUS1, config, &request,
    PUDS_SVC_PARAM_RDTCI_RDTCEDRBDN, 0x00A1B2B3, 0x12);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
```

```

if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: `UDS_WaitForService_2013` on page 656, `uds_svc_param_rdtci` on page 88.

Class-method version: `SvcReadDTCInformationReportExtended_2013` on page 408.

3.8.54 UDS_SvcReadDTCInformationReportSeverity_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. Only `reportNumberOfDTCBySeverityMaskRecord` and `reportDTCSeverityInformation` subfunctions are allowed.

Syntax

C/C++

```

uds_status UDS_SvcReadDTCInformationReportSeverity_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t PUDS_SVC_PARAM_RDTCI_Type,
    uint8_t dtc_severity_mask,
    uint8_t dtc_status_mask);

```

Parameters

| Parameter | Description |
|---------------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| PUDS_SVC_PARAM_RDTCI_Type | Subfunction parameter, ReadDTCInformation type, use one of the following: PUDS_SVC_PARAM_RDTCI_RNODTCBSMR, PUDS_SVC_PARAM_RDTCI_RDTCBSMR See also <code>uds_svc_param_rdtci</code> on page 88. |
| dtc_severity_mask | A mask of eight (8) DTC severity bits (see <code>uds_svc_param_rdtci_dtcsvm</code> on page 91). |
| dtc_status_mask | A mask of eight (8) DTC status bits. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---------------------------------|--|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |


| | |
|---|---|
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with UDS_Reset_2013 (see UDS_Reset_2013 on page 651). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see UDS_MsgFree_2013 on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see [UDS_MsgFree_2013](#) on page 645).

Example

The following example shows the use of the service function [UDS_SvcReadDTCInformationReportSeverity_2013](#) on the channel [PCANTP_HANDLE_USBBUS1](#). A UDS physical service request is transmitted, and the [UDS_WaitForService_2013](#) function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationReportSeverity message
result = UDS_SvcReadDTCInformationReportSeverity_2013(PCANTP_HANDLE_USBBUS1, config, &request,
PUDS_SVC_PARAM_RDTCI_RNODTCBSMR, 0xF1, 0x12);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
&request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
```



```
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);
```

See also: `UDS_WaitForService_2013` on page 656, `uds_svc_param_rdtci` on page 88, `uds_svc_param_rdtci_dtcsvm` on page 91.

Class-method version: `SvcReadDTCInformationReportSeverity_2013` on page 413.

3.8.55 UDS_SvcReadDTCInformationRSIODTC_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. The subfunction `reportSeverityInformationOfDTC` (`PUDS_SVC_PARAM_RDTCI_RSIODTC`) is implicit.

Syntax

C/C++

```
uds_status UDS_SvcReadDTCInformationRSIODTC_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint32_t dtc_mask);
```

Parameters

| Parameter | Description |
|-----------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| dtc_mask | A unique identification number for a specific diagnostic trouble code. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcReadDTCInformationRSIODTC_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the

`UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRSIODTC message
result = UDS_SvcReadDTCInformationRSIODTC_2013(PCANTP_HANDLE_USBBUS1, config, &request,
0xF1A1B2B3);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
&request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);
```

See also: `UDS_WaitForService_2013` on page 656.

Class-method version: `SvcReadDTCInformationRSIODTC_2013` on page 418.

3.8.56 UDS_SvcReadDTCInformationNoParam_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. Only theses following subfunctions are allowed:

- reportSupportedDTC
- reportFirstTestFailedDTC
- reportFirstConfirmedDTC
- reportMostRecentTestFailedDTC
- reportMostRecentConfirmedDTC

- reportDTCFaultDetectionCounter
- reportDTCWithPermanentStatus
- reportDTCSnapshotIdentification

Syntax

C/C++

```
uds_status UDS_SvcReadDTCInformationNoParam_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t PUDS_SVC_PARAM_RDTCI_Type);
```

Parameters

| Parameter | Description |
|---------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |
| PUDS_SVC_PARAM_RDTCI_Type | Subfunction parameter, ReadDTCInformation type, use one of the following: PUDS_SVC_PARAM_RDTCI_RFTFDTC, PUDS_SVC_PARAM_RDTCI_RFCDDTC, PUDS_SVC_PARAM_RDTCI_RMRTFDTC, PUDS_SVC_PARAM_RDTCI_RMRCDDTC, PUDS_SVC_PARAM_RDTCI_RSUPDTC, PUDS_SVC_PARAM_RDTCI_RDTWCPS, PUDS_SVC_PARAM_RDTCI_RDTCSSI , PUDS_SVC_PARAM_RDTCI_RDTCFDC. See also uds_svc_param_rdtci on page 88. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with UDS_Reset_2013 (see UDS_Reset_2013 on page 651). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see UDS_MsgFree_2013 on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcReadDTCInformationNoParam_2013` on the channel `PCANTP_HANDLE_USBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationNoParam message
result = UDS_SvcReadDTCInformationNoParam_2013(PCANTP_HANDLE_USBBUS1, config, &request,
        PUDS_SVC_PARAM_RDTCI_RSUPDTC);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);
```

See also: [UDS_WaitForService_2013](#) on page 656, [uds_svc_param_rdtci](#) on page 88.

Class-method version: [SvcReadDTCInformationNoParam_2013](#) on page 422.

3.8.57 UDS_SvcReadDTCInformationRDTCEDBR_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. The subfunction [reportDTCExtDataRecordByRecordNumber](#) ([PUDS_SVC_PARAM_RDTCI_RDTCEDBR](#)) is implicit.

Syntax

C/C++

```
uds_status UDS_SvcReadDTCInformationRDTCEDBR_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg* out_msg_request,
    uint8_t dtc_extended_data_record_number);
```

Parameters

| Parameter | Description |
|---------------------------------|---|
| channel | The handle of a PUDS channel (see <code>can_tp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| dtc_extended_data_record_number | DTC extended data record number. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcReadDTCInformationRDTCEDBR_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;
uint8_t dtc_extended_data_record_number = 0x12;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
```

```

config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical SvcReadDTCInformationRDTCEDBR message
result = UDS_SvcReadDTCInformationRDTCEDBR_2013(PCANTP_HANDLE_USBBUS1, config, &request,
    dtc_extended_data_record_number);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: `UDS_WaitForService_2013` on page 656.

Class-method version: `SvcReadDTCInformationRDTCEDBR_2013` on page 426.

3.8.58 UDS_SvcReadDTCInformationRUDMDTCBSM_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. The subfunction `reportUserDefMemoryDTCByStatusMask` (`PUDS_SVC_PARAM_RDTCI_RUDMDTCBSM`) is implicit.

Syntax

C/C++

```

uds_status UDS_SvcReadDTCInformationRUDMDTCBSM_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg* out_msg_request,
    uint8_t dtc_status_mask,
    uint8_t memory_selection);

```

Parameters

| Parameter | Description |
|------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| Request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| Out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| Dtc_status_mask | A mask of eight (8) DTC status bits. |
| Memory_selection | Memory selection. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcReadDTCInformationRUDMDTCBSM_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;
uint8_t dtc_status_mask = 0x12;
uint8_t memory_selection = 0x34;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRUDMDTCBSM message
result = UDS_SvcReadDTCInformationRUDMDTCBSM_2013(PCANTP_HANDLE_USBBUS1, config, &request,
    dtc_status_mask, memory_selection);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
```

```

        result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
            &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: `UDS_WaitForService_2013` on page 656.

Class-method version: `SvcReadDTCInformationRUDMDTCBSM_2013` on page 431.

3.8.59 UDS_SvcReadDTCInformationRUDMDTCSSBDTC_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. The subfunction `reportUserDefMemoryDTCsSnapshotRecordByDTCNumber` (`PUDS_SVC_PARAM_RDTCL_RUDMDTCSSBDTC`) is implicit.

Syntax

C/C++

```

uds_status UDS_SvcReadDTCInformationRUDMDTCSSBDTC_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg* out_msg_request,
    uint32_t dtc_mask,
    uint8_t user_def_dtc_snapshot_record_number,
    uint8_t memory_selection);

```

Parameters

| Parameter | Description |
|-------------------------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| dtc_mask | A unique identification number (three-byte value) for a specific diagnostic trouble code. |
| user_def_dtc_snapshot_record_number | User DTC snapshot record number. |
| memory_selection | Memory selection. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcReadDTCInformationRUDMDTCSSBDTC_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;
uint32_t dtc_mask = 0x12345678;
uint8_t user_def_dtc_snapshot_record_number = 0x9A;
uint8_t memory_selection = 0xBC;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRUDMDTCSSBDTC message
result = UDS_SvcReadDTCInformationRUDMDTCSSBDTC_2013(PCANTP_HANDLE_USBBUS1, config, &request,
    dtc_mask, user_def_dtc_snapshot_record_number, memory_selection);
```

```

if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
    &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: `UDS_WaitForService_2013` on page 656.

Class-method version: `SvcReadDTCInformationRUDMDTCSSBDTC_2013` on page 435.

3.8.60 UDS_SvcReadDTCInformationRUDMDTCEDRBDN_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. The subfunction `reportUserDefMemoryDTCExtDataRecordByDTCNumber` (`PUDS_SVC_PARAM_RDTCI_RUDMDTCEDRBDN`) is implicit.

Syntax

C/C++

```

uds_status UDS_SvcReadDTCInformationRUDMDTCEDRBDN_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg* out_msg_request,
    uint32_t dtc_mask,
    uint8_t dtc_extended_data_record_number,
    uint8_t memory_selection);

```

Parameters

| Parameter | Description |
|---------------------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| dtc_mask | A unique identification number (three-byte value) for a specific diagnostic trouble code. |
| dtc_extended_data_record_number | DTC extended data record number. |
| memory_selection | Memory selection. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |


| | |
|---|---|
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see UDS_MsgFree_2013 on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see [UDS_MsgFree_2013](#) on page 645).

Example

The following example shows the use of the service function [UDS_SvcReadDTCInformationRUDMDTCEDRBDN_2013](#) on the channel [PCANTP_HANDLE_USBBUS1](#). A UDS physical service request is transmitted, and the [UDS_WaitForService_2013](#) function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;
uint32_t dtc_mask = 0x12345678;
uint8_t dtc_extended_data_record_number = 0x9A;
uint8_t memory_selection = 0xBC;

memset(&request, 0, sizeof(request)) ;
memset(&request_confirmation, 0, sizeof(request_confirmation)) ;
memset(&response, 0, sizeof(response)) ;
memset(&config, 0, sizeof(config)) ;

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRUDMDTCEDRBDN message
result = UDS_SvcReadDTCInformationRUDMDTCEDRBDN_2013(PCANTP_HANDLE_USBBUS1, config, &request,
    dtc_mask, dtc_extended_data_record_number, memory_selection);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);
```

See also: [UDS_WaitForService_2013](#) on page 656.

Class-method version: [SvcReadDTCInformationRUDMDTCEDRBDN_2013](#) on page 440.

3.8.61 UDS_SvcReadDTCInformationRDTCEdi_2020

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. The subfunction [reportSupportedDTCExtDataRecord](#) ([PUDS_SVC_PARAM_RDTCEdi](#)) is implicit.

Syntax

C/C++

```
uds_status UDS_SvcReadDTCInformationRDTCEdi_2020(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg* out_msg_request,
    uint8_t dtc_extended_data_record_number);
```

Parameters

| Parameter | Description |
|---------------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |
| dtc_extended_data_record_number | DTC extended data record number. |

Returns

The return value is a [uds_status](#) code. [PUDS_STATUS_OK](#) is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with UDS_Reset_2013 (see UDS_Reset_2013 on page 651). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see UDS_MsgFree_2013 on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see [UDS_MsgFree_2013](#) on page 645).

Example

The following example shows the use of the service function [UDS_SvcReadDTCInformationRDTCEdi_2020](#) on the channel [PCANTP_HANDLE_USBBUS1](#). A UDS physical service request is transmitted, and the [UDS_WaitForService_2013](#) function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```

uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;
uint8_t dtc_extended_data_record_number = 0x12;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRDTCEdi message
result = UDS_SvcReadDTCInformationRDTCEdi_2020(PCANTP_HANDLE_USBBUS1, config, &request,
    dtc_extended_data_record_number);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: [UDS_WaitForService_2013](#) on page 656.

Class-method version: [SvcReadDTCInformationRDTCEdi_2020](#) on page 444.

3.8.62 UDS_SvcReadDTCInformationRWWHOBDDTCBMR_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. The subfunction [reportRWWHOBDDTCByMaskRecord](#) ([PUDS_SVC_PARAM_RDTCl_RWWHOBDDTCBMR](#)) is implicit.

Syntax**C/C++**

```

uds_status UDS_SvcReadDTCInformationRWWHOBDDTCBMR_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg* out_msg_request,

```

```
uint8_t functional_group_identifier,
uint8_t dtc_status_mask,
uint8_t dtc_severity_mask);
```

Parameters

| Parameter | Description |
|-----------------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| functional_group_identifier | Functional group identifier. |
| dtc_status_mask | A mask of eight (8) DTC status bits. |
| dtc_severity_mask | A mask of eight (8) DTC severity bits (see <code>uds_svc_param_rdtci_dtcsvm</code> on page 91). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see <code>uds_msgconfig</code> on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcReadDTCInformationRWWHOBDDTCBMR_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;
uint8_t functional_group_identifier = 0x12;
uint8_t dtc_status_mask = 0x34;
uint8_t dtc_severity_mask = 0x78;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
```

```
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRWWHOBDDTCBMR message
result = UDS_SvcReadDTCInformationRWWHOBDDTCBMR_2013(PCANTP_HANDLE_USBBUS1, config, &request,
    functional_group_identifier, dtc_status_mask, dtc_severity_mask);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);
```

See also: `UDS_WaitForService_2013` on page 656.

Class-method version: `SvcReadDTCInformationRWWHOBDDTCBMR_2013` on page 449.

3.8.63 UDS_SvcReadDTCInformationRWWHOBDDTCWPS_2013

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. The subfunction `reportWWHOBDDTCWithPermanentStatus` (`PUDS_SVC_PARAM_RDTCI_RWWHOBDDTCWPS`) is implicit.

Syntax

C/C++

```
uds_status UDS_SvcReadDTCInformationRWWHOBDDTCWPS_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg* out_msg_request,
    uint8_t functional_group_identifier);
```

Parameters

| Parameter | Description |
|-----------------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| functional_group_identifier | Functional group identifier. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with UDS_Reset_2013 (see UDS_Reset_2013 on page 651). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see UDS_MsgFree_2013 on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see [UDS_MsgFree_2013](#) on page 645).

Example

The following example shows the use of the service function [UDS_SvcReadDTCInformationRWWHOBDDTCWPS_2013](#) on the channel [PCANTP_HANDLE_USBBUS1](#). A UDS physical service request is transmitted, and the [UDS_WaitForService_2013](#) function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;
uint8_t functional_group_identifier = 0x12;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRWWHOBDDTCWPS message
result = UDS_SvcReadDTCInformationRWWHOBDDTCWPS_2013(PCANTP_HANDLE_USBBUS1, config, &request,
functional_group_identifier);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
&request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
```



```
printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);
```

See also: [UDS_WaitForService_2013](#) on page 656.

Class-method version: [SvcReadDTCInformationRWWHOBDDTCWPS_2013](#) on page 453.

3.8.64 UDS_SvcReadDTCInformationRDTCBRI_2020

Writes a UDS request according to the ReadDTCInformation service's specifications. This service allows a client to read the status of server-resident Diagnostic Trouble Code (DTC) information. The subfunction [reportDTCInformationByDTCReadinessGroupIdentifier](#) ([PUDS_SVC_PARAM_RDTCI_RDTCBRI](#)) is implicit.

Syntax

C/C++

```
uds_status UDS_SvcReadDTCInformationRDTCBRI_2020(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg* out_msg_request,
    uint8_t functional_group_identifier,
    uint8_t dtc_readiness_group_identifier);
```

Parameters

| Parameter | Description |
|--------------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |
| functional_group_identifier | Functional group identifier. |
| dtc_readiness_group_identifier | DTC readiness group identifier. |

Returns

The return value is a [uds_status](#) code. [PUDS_STATUS_OK](#) is returned on success. The typical errors in case of failure are:

| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with UDS_Reset_2013 (see UDS_Reset_2013 on page 651). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see UDS_MsgFree_2013 on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see [UDS_MsgFree_2013](#) on page 645).

Example

The following example shows the use of the service function `UDS_SvcReadDTCInformationRDTCBRGI_2020` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

Note: It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;
uint8_t functional_group_identifier = 0x12;
uint8_t dtc_readiness_group_identifier = 0x34;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical ReadDTCInformationRDTCBRGI message
result = UDS_SvcReadDTCInformationRDTCBRGI_2020(PCANTP_HANDLE_USBBUS1, config, &request,
    functional_group_identifier, dtc_readiness_group_identifier);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);
```

See also: `UDS_WaitForService_2013` on page 656.

Class-method version: `SvcReadDTCInformationRDTCBRGI_2020` on page 458.

3.8.65 UDS_SvcInputOutputControlByIdentifier_2013

Writes a UDS request according to the InputOutputControlByIdentifier service's specifications. The InputOutputControlByIdentifier service is used by the client to substitute a value for an input signal, internal server (ECU) function and/or control an output (actuator) of an electronic system.

Syntax

C

```
uds_status UDS_SvcInputOutputControlByIdentifier_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint16_t data_identifier,
    uint8_t * control_option_record,
    uint32_t control_option_record_size,
    uint8_t * control_enable_mask_record,
    uint32_t control_enable_mask_record_size);
```

C++

```
uds_status UDS_SvcInputOutputControlByIdentifier_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint16_t data_identifier,
    uint8_t * control_option_record,
    uint32_t control_option_record_size,
    uint8_t * control_enable_mask_record = nullptr,
    uint32_t control_enable_mask_record_size = 0);
```

Parameters

| Parameter | Description |
|---------------------------------|---|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |
| data_identifier | A two-byte data identifier (see uds_svc_param_di on page 82). |
| control_option_record | First byte can be used as either an input output control parameter that describes how the server (ECU) shall control its inputs or outputs (see uds_svc_param_iocbi on page 92), or as an additional control state byte. |
| control_option_record_size | Size in bytes of the control option record buffer. |
| control_enable_mask_record | The control enable mask shall only be supported when the input output control parameter is used and the data identifier to be controlled consists of more than one parameter (i.e. the data identifier is bit-mapped or packed by definition). There shall be one bit in the control enable mask corresponding to each individual parameter defined within the data identifier. |
| control_enable_mask_record_size | Size in bytes of the control enable mask record buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is not valid. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | The mapping given in the message configuration is unknown. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with UDS_Reset_2013 (see UDS_Reset_2013 on page 651). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_OVERFLOW | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data size. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see UDS_MsgFree_2013 on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see [UDS_MsgFree_2013](#) on page 645).

Example

The following example shows the use of the service function [UDS_SvcInputOutputControlByIdentifier_2013](#) on the channel [PCANTP_HANDLE_USBBUS1](#). A UDS physical service request is transmitted, and the [UDS_WaitForService_2013](#) function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uint8_t control_option_record[10];
uint8_t control_enable_mask_record[10];
uint16_t control_option_record_size = 10;
uint16_t control_enable_mask_record_size = 5;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Fill Data
for (int i = 0; i < control_option_record_size; i++)
{
    control_option_record[i] = 'A' + i;
    control_enable_mask_record[i] = 10 + i;
}
```

```

}

// Sends a physical InputOutputControlByIdentifier message
result = UDS_SvcInputOutputControlByIdentifier_2013(PCANTP_HANDLE_USBBUS1, config, &request,
    PUDS_SVC_PARAM_DI_SSECUSWVNDID, control_option_record, control_option_record_size,
    control_enable_mask_record, control_enable_mask_record_size);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: [UDS_WaitForService_2013](#) on page 656, [uds_svc_param_iocbi](#) on page 92.

Class-method version: [SvcInputOutputControlByIdentifier_2013](#) on page 462.

3.8.66 UDS_SvcRoutineControl_2013

Writes a UDS request according to the RoutineControl service's specifications. The RoutineControl service is used by the client to start/stop a routine and request routine results.

Syntax

C

```

uds_status UDS_SvcRoutineControl_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t routine_control_type,
    uint16_t routine_identifier,
    uint8_t * routine_control_option_record,
    uint32_t routine_control_option_record_size);

```

C++

```

uds_status UDS_SvcRoutineControl_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t routine_control_type,
    uint16_t routine_identifier,
    uint8_t * routine_control_option_record = nullptr,
    uint32_t routine_control_option_record_size = 0);

```

Parameters

| Parameter | Description |
|------------------------------------|---|
| channel | The handle of a PUDS channel (see <code>canntp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| routine_control_type | Subfunction parameter: routine control type (see <code>uds_svc_param_rc</code> on page 93). |
| routine_identifier | Server local routine identifier (see <code>uds_svc_param_rc_rid</code> on page 94). |
| routine_control_option_record | Buffer containing the routine control options (only with start and stop routine subfunctions). |
| routine_control_option_record_size | Size in bytes of the buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_OVERFLOW</code> | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data size. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcRoutineControl_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uint8_t routine_control_option_record[10];
uint16_t routine_control_option_record_size = 10;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));
```

```
// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Fill data
for (int i = 0; i < routine_control_option_record_size; i++)
{
    routine_control_option_record[i] = 'A' + i;
}

// Sends a physical RoutineControl message
result = UDS_SvcRoutineControl_2013(PCANTP_HANDLE_USBBUS1, config, &request,
    PUDS_SVC_PARAM_RC_RRR, 0xF1A2, routine_control_option_record,
    routine_control_option_record_size);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);
```

See also: [UDS_WaitForService_2013](#) on page 656, [uds_svc_param_rc](#) on page 93, [uds_svc_param_rc_rid](#) on page 94.
Class-method version: [SvcRoutineControl_2013](#) on page 473.

3.8.67 UDS_SvcRequestDownload_2013

Writes a UDS request according to the RequestDownload service's specifications. The RequestDownload service is used by the client to initiate a data transfer from the client to the server/ECU (download).

Syntax

C/C++

```
uds_status UDS_SvcRequestDownload_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t compression_method,
    uint8_t encrypting_method,
    uint8_t * memory_address_buffer,
    uint8_t memory_address_size,
    uint8_t * memory_size_buffer,
    uint8_t memory_size_size);
```

Parameters

| Parameter | Description |
|-----------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |

| Parameter | Description |
|-----------------------|---|
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| compression_method | A nibble-value that specifies the compression method, the value 0x0 specifies that no compression method is used. |
| encrypting_method | A nibble-value that specifies the encrypting method, the value 0x0 specifies that no encrypting method is used. |
| memory_address_buffer | Starting address of server (ECU) memory to which data is to be written. |
| memory_address_size | Size in bytes of the memory address buffer (max.: 0xF). |
| memory_size_buffer | Used by the server (ECU) to compare the uncompressed memory size with the total amount of data transferred during the TransferData service. |
| memory_size_size | Size in bytes of the memory size buffer (max.: 0xF). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcRequestDownload_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uint8_t memory_address_buffer[8];
uint8_t memory_size_buffer[8];
uint8_t memory_address_size = 8;
uint8_t memory_size_size = 8;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));
```



```
// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Fill data
for (int i = 0; i < memory_address_size; i++)
{
    memory_address_buffer[i] = 'A' + i;
    memory_size_buffer[i] = 10 + i;
}

// Sends a physical RequestDownload message
result = UDS_SvcRequestDownload_2013(PCANTP_HANDLE_USBBUS1, config, &request, 0x01, 0x02,
    memory_address_buffer, memory_address_size, memory_size_buffer, memory_size_size);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);
```

See also: [UDS_WaitForService_2013](#) on page 656.

Class-method version: [SvcRequestDownload_2013](#) on page 483.

3.8.68 UDS_SvcRequestUpload_2013

Writes a UDS request according to the RequestUpload service's specifications. The RequestUpload service is used by the client to initiate a data transfer from the server/ECU to the client (upload).

Syntax

C/C++

```
uds_status UDS_SvcRequestUpload_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t compression_method,
    uint8_t encrypting_method,
    uint8_t * memory_address_buffer,
    uint8_t memory_address_size,
    uint8_t * memory_size_buffer,
    uint8_t memory_size_size);
```

Parameters

| Parameter | Description |
|-----------------------|---|
| channel | The handle of a PUDS channel (see <code>can_tp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| compression_method | A nibble-value that specifies the compression method, the value 0x0 specifies that no compression method is used. |
| encrypting_method | A nibble-value that specifies the encrypting method, the value 0x0 specifies that no encrypting method is used. |
| memory_address_buffer | Starting address of server (ECU) memory from which data is to be retrieved. |
| memory_address_size | Size in bytes of the memory address buffer (max.: 0xF). |
| memory_size_buffer | Used by the server (ECU) to compare the uncompressed memory size with the total amount of data transferred during the TransferData service. |
| memory_size_size | Size in bytes of the memory size buffer (max.: 0xF). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcRequestUpload_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uint8_t memory_address_buffer[4];
uint8_t memory_size_buffer[4];
uint8_t memory_address_size = 4;
uint8_t memory_size_size = 4;
uds_msgconfig config;
int i;
```

```
memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));
for (i = 0; i < 4; i++) {
    memory_address_buffer[i] = 0;
    memory_size_buffer[i] = 0;
}

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical RequestUpload message
result = UDS_SvcRequestUpload_2013(PCANTP_HANDLE_USBBUS1, config, &request, 0x01, 0x02,
    memory_address_buffer, memory_address_size, memory_size_buffer, memory_size_size);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);
```

See also: [UDS_WaitForService_2013](#) on page 656.

Class-method version: [SvcRequestUpload_2013](#) on page 489.

3.8.69 UDS_SvcTransferData_2013

Writes a UDS request according to the TransferData service's specifications. The TransferData service is used by the client to transfer data either from the client to the server/ECU (download) or from the server/ECU to the client (upload).

Syntax

C

```
uds_status UDS_SvcTransferData_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t block_sequence_counter,
    uint8_t * transfer_request_parameter_record,
    uint32_t transfer_request_parameter_record_size);
```

C++

```
uds_status UDS_SvcTransferData_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t block_sequence_counter,
    uint8_t * transfer_request_parameter_record = nullptr,
    uint32_t transfer_request_parameter_record_size = 0);
```

Parameters

| Parameter | Description |
|--|--|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| block_sequence_counter | The <code>block_sequence_counter</code> parameter value starts at 0x01 with the first <code>TransferData</code> request that follows the <code>RequestDownload</code> (0x34) or <code>RequestUpload</code> (0x35) service. Its value is incremented by 1 for each subsequent <code>TransferData</code> request. At the value of 0xFF, the <code>block_sequence_counter</code> rolls over and starts at 0x00 with the next <code>TransferData</code> request message. |
| transfer_request_parameter_record | Buffer containing the required transfer parameters. |
| transfer_request_parameter_record_size | Size in bytes of the buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_OVERFLOW</code> | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data size. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcTransferData_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```

uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uint8_t record[50];
uint8_t record_size = 50;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDES_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDES_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDES_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDES_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDES_MSGTYPE_USDT;

// Fill data
for (int i = 0; i < record_size; i++)
{
    record[i] = 'A' + i;
}

// Sends a physical TransferData message
result = UDS_SvcTransferData_2013(PCANTP_HANDLE_USBBUS1, config, &request, 0x01, record,
    record_size);
if (UDS_StatusIsOk_2013(result, PUDES_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDES_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: `UDS_WaitForService_2013` on page 656.

Class-method version: `SvcTransferData_2013` on page 494.

3.8.70 UDS_SvcRequestTransferExit_2013

Writes a UDS request according to the RequestTransferExit service's specifications. The RequestTransferExit service is used by the client to terminate a data transfer between the client and server/ECU (upload or download).

Syntax

C

```
uds_status UDS_SvcRequestTransferExit_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t * transfer_request_parameter_record,
    uint32_t transfer_request_parameter_record_size);
```

C++

```
uds_status UDS_SvcRequestTransferExit_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t * transfer_request_parameter_record = nullptr,
    uint32_t transfer_request_parameter_record_size = 0);
```

Parameters

| Parameter | Description |
|--|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |
| transfer_request_parameter_record | Buffer containing the required transfer parameters. |
| transfer_request_parameter_record_size | Size in bytes of the buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is not valid. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | The mapping given in the message configuration is unknown. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with UDS_Reset_2013 (see UDS_Reset_2013 on page 651). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_OVERFLOW | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data size. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see UDS_MsgFree_2013 on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcRequestTransferExit_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uint8_t transfer_request_parameter_record[20];
uint8_t transfer_request_parameter_record_size = 20;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Fill data
for (int i = 0; i < transfer_request_parameter_record_size; i++)
{
    transfer_request_parameter_record[i] = 'A' + i;
}

// Sends a physical RequestTransferExit message
result = UDS_SvcRequestTransferExit_2013(PCANTP_HANDLE_USBBUS1, config, &request,
    transfer_request_parameter_record, transfer_request_parameter_record_size);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);
```

See also: `UDS_WaitForService_2013` on page 656.

Class-method version: `SvcRequestTransferExit_2013` on page 503.

3.8.71 UDS_SvcAccessTimingParameter_2013

Writes a UDS request according to the AccessTimingParameter service's specifications.

Syntax

C/C++

```
uds_status UDS_SvcAccessTimingParameter_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t access_type,
    uint8_t* request_record,
    uint32_t request_record_size);
```

Parameters

| Parameter | Description |
|---------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |
| access_type | Access type (see uds_svc_param_atp on page 95). |
| request_record | Timing parameter request record. |
| request_record_size | Size in bytes of the request record. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is not valid. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | The mapping given in the message configuration is unknown. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_OVERFLOW | The given buffer size is too big, the resulting UDS message data size is bigger than the maximum data size. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcAccessTimingParameter_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++


```

uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical SvcAccessTimingParameter message
uint8_t request_record[2] = { 0xAB, 0xCD };
uint32_t request_record_length = 2;
result = UDS_SvcAccessTimingParameter_2013(PCANTP_HANDLE_USBBUS1, config, &request,
    PUDS_SVC_PARAM_ATP_RCATP, request_record, request_record_length);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: [uds_svc_param_atp](#) on page 95, [UDS_WaitForService_2013](#) on page 656.

Class-method version: [SvcAccessTimingParameter_2013](#) on page 512.

3.8.72 UDS_SvcRequestFileTransfer_2013

Writes a UDS request according to the SvcRequestFileTransfer service's specifications.

Syntax

C

```

uds_status UDS_SvcRequestFileTransfer_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t mode_of_operation,
    uint16_t file_path_and_name_size,
    uint8_t *file_path_and_name,
    uint8_t compression_method,
    uint8_t encrypting_method,
    uint8_t file_size_parameter_size,
    uint8_t *file_size_uncompressed,

```

```
uint8_t *file_size_compressed);
```

C++

```
uds_status UDS_SvcRequestFileTransfer_2013(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t mode_of_operation,
    uint16_t file_path_and_name_size,
    uint8_t *file_path_and_name,
    uint8_t compression_method = 0,
    uint8_t encrypting_method = 0,
    uint8_t file_size_parameter_size = 0,
    uint8_t *file_size_uncompressed = 0,
    uint8_t *file_size_compressed = 0);
```

Parameters

| Parameter | Description |
|--------------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| mode_of_operation | Mode of operation (see <code>uds_svc_param_rft_moop</code> on page 96). |
| file_path_and_name_size | Size in bytes of <code>file_path_and_name</code> buffer |
| file_path_and_name | File path and name string. |
| compression_method | A nibble-value that specifies the compression method, the value 0x0 specifies that no compression method is used. |
| encrypting_method | A nibble-value that specifies the encrypting method, the value 0x0 specifies that no encrypting method is used. |
| file_size_parameter_size | Size in byte of <code>file_size_uncompressed</code> and <code>file_size_compressed</code> parameters |
| file_size_uncompressed | Uncompressed file size. |
| file_size_compressed | Compressed file size. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:


| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |

Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

Example

The following example shows the use of the service function `UDS_SvcRequestFileTransfer_2013` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical RequestFileTransfer message
uint8_t file_name[9] = "toto.txt";
uint16_t file_size_uncompressed = 0xD;
uint16_t file_size_compressed = 0xA;
result = UDS_SvcRequestFileTransfer_2013(PCANTP_HANDLE_USBBUS1, config, &request,
    PUDS_SVC_PARAM_RFT_MOOP_ADDFILE, 8, file_name, 0, 0, 2,
    (uint8_t*)&file_size_uncompressed, (uint8_t*)&file_size_compressed);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);
```

See also: `uds_svc_param_rft_moop` on page 96, `UDS_WaitForService_2013` on page 656.

Class-method version: `SvcRequestFileTransfer_2013` on page 517.

3.8.73 UDS_SvcAuthenticationDA_2020

Writes a UDS request according to the Authentication service's specifications (ISO-14229-1:2020). The subfunction deAuthenticate is implicit.

Syntax

C/C++

```
uds_status UDS_SvcAuthenticationDA_2020(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request);
```

Parameters

| Parameter | Description |
|-----------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |


Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

The PCAN-UDS 2.x API provides `uds_svc_authentication_subfunction` (see on page 98) and `uds_svc_authentication_return_parameter` (see on page 99) enumerations to help user to decode Authentication service responses.

Example

The following example shows the use of the service function `UDS_SvcAuthenticationDA_2020` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```

uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/deAuthenticate message
result = UDS_SvcAuthenticationDA_2020(PCANTP_HANDLE_USBBUS1, config, &request);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
    &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: [UDS_WaitForService_2013](#) on page 656.

Class-method version: [SvcAuthenticationDA_2020](#) on page 545.

3.8.74 UDS_SvcAuthenticationVCU_2020

Writes a UDS request according to the Authentication service's specifications (ISO-14229-1:2020). The subfunction verifyCertificateUnidirectional is implicit.

Syntax**C**

```

uds_status UDS_SvcAuthenticationVCU_2020(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t communication_configuration,
    uint8_t *certificate_client,
    uint16_t certificate_client_size,
    uint8_t *challenge_client,
    uint16_t challenge_client_size);

```

C++

```
uds_status UDS_SvcAuthenticationVCU_2020(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t communication_configuration,
    uint8_t *certificate_client,
    uint16_t certificate_client_size,
    uint8_t *challenge_client = nullptr,
    uint16_t challenge_client_size = 0);
```

Parameters

| Parameter | Description |
|-----------------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| communication_configuration | Configuration information about communication. |
| certificate_client | Buffer containing the certificate of the client. |
| certificate_client_size | Size in bytes of the certificate buffer. |
| challenge_client | Buffer containing the challenge of the client. |
| challenge_client_size | Size in bytes of the challenge buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |


Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

The PCAN-UDS 2.x API provides `uds_svc_authentication_subfunction` (see on page 98) and `uds_svc_authentication_return_parameter` (see on page 99) enumerations to help user to decode Authentication service responses.

Example

The following example shows the use of the service function `UDS_SvcAuthenticationVCU_2020` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```

uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/verifyCertificateUnidirectional message
uint8_t communication_configuration = 0x00;
uint8_t certificate_client[2] = { 0x12, 0x34 };
uint16_t certificate_client_size = 2;
uint8_t challenge_client[2] = { 0x56, 0x78 };
uint16_t challenge_client_size = 2;
result = UDS_SvcAuthenticationVCU_2020(PCANTP_HANDLE_USBBUS1, config, &request,
    communication_configuration, certificate_client, certificate_client_size,
    challenge_client, challenge_client_size);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: [UDS_WaitForService_2013](#) on page 656.

Class-method version: [SvcAuthenticationVCU_2020](#) on page 549.

3.8.75 UDS_SvcAuthenticationVCB_2020

Writes a UDS request according to the Authentication service's specifications (ISO-14229-1:2020). The subfunction `verifyCertificateBidirectional` is implicit.

Syntax

C/C++

```
uds_status UDS_SvcAuthenticationVCB_2020(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t communication_configuration,
    uint8_t *certificate_client,
    uint16_t certificate_client_size,
    uint8_t *challenge_client,
    uint16_t challenge_client_size) ;
```

Parameters

| Parameter | Description |
|-----------------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| Request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| Out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| Communication_configuration | Configuration information about communication. |
| Certificate_client | Buffer containing the certificate of the client. |
| Certificate_client_size | Size in bytes of the certificate buffer. |
| Challenge_client | Buffer containing the challenge of the client. |
| Challenge_client_size | Size in bytes of the challenge buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |


Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

The PCAN-UDS 2.x API provides `uds_svc_authentication_subfunction` (see on page 98) and `uds_svc_authentication_return_parameter` (see on page 99) enumerations to help user to decode Authentication service responses.

Example

The following example shows the use of the service function `UDS_SvcAuthenticationVCB_2020` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/verifyCertificateBidirectional message
uint8_t communication_configuration = 0x00;
uint8_t certificate_client[2] = { 0x12, 0x34 };
uint16_t certificate_client_size = 2;
uint8_t challenge_client[2] = { 0x56, 0x78 };
uint16_t challenge_client_size = 2;
result = UDS_SvcAuthenticationVCB_2020(PCANTP_HANDLE_USBBUS1, config, &request,
    communication_configuration, certificate_client, certificate_client_size,
    challenge_client, challenge_client_size);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);
```

See also: `UDS_WaitForService_2013` on page 656.

Class-method version: `SvcAuthenticationVCB_2020` on page 559.

3.8.76 UDS_SvcAuthenticationPOWN_2020

Writes a UDS request according to the Authentication service's specifications (ISO-14229-1:2020). The subfunction proofOfOwnership is implicit.

Syntax

C

```
uds_status UDS_SvcAuthenticationPOWN_2020(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t *proof_of_ownership_client,
    uint16_t proof_of_ownership_client_size,
    uint8_t *ephemeral_public_key_client,
    uint16_t ephemeral_public_key_client_size);
```

C++

```
uds_status UDS_SvcAuthenticationPOWN_2020(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t *proof_of_ownership_client,
    uint16_t proof_of_ownership_client_size,
    uint8_t *ephemeral_public_key_client = nullptr,
    uint16_t ephemeral_public_key_client_size = 0);
```

Parameters

| Parameter | Description |
|----------------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |
| proof_of_ownership_client | Buffer containing the proof of ownership of the client. |
| proof_of_ownership_client_size | Size in bytes of the proof of ownership buffer. |
| ephemeral_public_key_client | Buffer containing the ephemeral public key of the client. |
| ephemeral_public_key_client_size | Size in bytes of the ephemeral public key buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is not valid. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | The mapping given in the message configuration is unknown. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |


Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see [UDS_MsgFree_2013](#) on page 645).

The PCAN-UDS 2.x API provides [uds_svc_authentication_subfunction](#) (see on page 98) and [uds_svc_authentication_return_parameter](#) (see on page 99) enumerations to help user to decode Authentication service responses.

Example

The following example shows the use of the service function [UDS_SvcAuthenticationPOWN_2020](#) on the channel [PCANTP_HANDLE_USBBUS1](#). A UDS physical service request is transmitted, and the [UDS_WaitForService_2013](#) function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/proofOfOwnership message
uint8_t proof_of_ownership_client[2] = { 0x12, 0x34 };
uint16_t proof_of_ownership_client_size = 2;
uint8_t ephemeral_public_key_client[2] = { 0x56, 0x78 };
uint16_t ephemeral_public_key_client_size = 2;
result = UDS_SvcAuthenticationPOWN_2020(PCANTP_HANDLE_USBBUS1, config, &request,
    proof_of_ownership_client, proof_of_ownership_client_size, ephemeral_public_key_client,
    ephemeral_public_key_client_size);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);
```

See also: [UDS_WaitForService_2013](#) on page 656.

Class-method version: [SvcAuthenticationPOWN_2020](#) on page 565.

3.8.77 UDS_SvcAuthenticationRCFA_2020

Writes a UDS request according to the Authentication service's specifications (ISO-14229-1:2020). The subfunction requestChallengeForAuthentication is implicit.

Syntax

C/C++

```
uds_status UDS_SvcAuthenticationRCFA_2020(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t communication_configuration,
    uint8_t *algorithm_indicator);
```

Parameters

| Parameter | Description |
|-----------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| request_config | Message request configuration (see uds_msgconfig on page 27). |
| out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |
| communication_configuration | Configuration information about communication. |
| algorithm_indicator | Buffer of 16 bytes containing the algorithm indicator. |

Returns

The return value is a [uds_status](#) code. [PUDS_STATUS_OK](#) is returned on success. The typical errors in case of failure are:

| | |
|---|---|
| PUDS_STATUS_NOT_INITIALIZED | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| PUDS_STATUS_PARAM_INVALID_VALUE | One of the given parameters is not valid. |
| PUDS_STATUS_MAPPING_NOT_INITIALIZED | The mapping given in the message configuration is unknown. |
| PUDS_STATUS_SERVICE_ALREADY_PENDING | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with UDS_Reset_2013 (see UDS_Reset_2013 on page 651). |
| PUDS_STATUS_NO_MEMORY | Failed to allocate memory and copy the new message in the transmission queue. |
| PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED | The given message buffer is already allocated, user must release the buffer before reusing it (see UDS_MsgFree_2013 on page 645). |


Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see [UDS_MsgFree_2013](#) on page 645).

The PCAN-UDS 2.x API provides [uds_svc_authentication_subfunction](#) (see on page 98) and [uds_svc_authentication_return_parameter](#) (see on page 99) enumerations to help user to decode Authentication service responses.

Example

The following example shows the use of the service function `UDS_SvcAuthenticationRCFA_2020` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result ;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/requestChallengeForAuthentication message
uint8_t communication_configuration = 0x00;
uint8_t algorithm_indicator[16] = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
    0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F };
result = UDS_SvcAuthenticationRCFA_2020(PCANTP_HANDLE_USBBUS1, config, &request,
    communication_configuration, algorithm_indicator);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);
```

See also: `UDS_WaitForService_2013` on page 656.

Class-method version: `SvcAuthenticationRCFA_2020` on page 574.

3.8.78 UDS_SvcAuthenticationVPOWNU_2020

Writes a UDS request according to the Authentication service's specifications (ISO-14229-1:2020). The subfunction verifyProofOfOwnershipUnidirectional is implicit.

Syntax

C

```
uds_status UDS_SvcAuthenticationVPOWNU_2020(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t *algorithm_indicator,
    uint8_t *proof_of_ownership_client,
    uint16_t proof_of_ownership_client_size,
    uint8_t *challenge_client,
    uint16_t challenge_client_size,
    uint8_t *additional_parameter,
    uint16_t additional_parameter_size);
```

C++

```
uds_status UDS_SvcAuthenticationVPOWNU_2020(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t *algorithm_indicator,
    uint8_t *proof_of_ownership_client,
    uint16_t proof_of_ownership_client_size,
    uint8_t *challenge_client = nullptr,
    uint16_t challenge_client_size = 0,
    uint8_t *additional_parameter = nullptr,
    uint16_t additional_parameter_size = 0);
```

Parameters

| Parameter | Description |
|--------------------------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| Request_config | Message request configuration (see uds_msgconfig on page 27). |
| Out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |
| Algorithm_indicator | Buffer of 16 bytes containing the algorithm indicator |
| proof_of_ownership_client | Buffer containing the proof of ownership of the client. |
| Proof_of_ownership_client_size | Size in bytes of the proof of ownership buffer. |
| Challenge_client | Buffer containing the challenge of the client. |
| Challenge_client_size | Size in bytes of the challenge buffer. |
| Additional_parameter | Buffer containing additional parameters. |
| Additional_parameter_size | Size in bytes of the additional parameter buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |


Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

The PCAN-UDS 2.x API provides `uds_svc_authentication_subfunction` (see on page 98) and `uds_svc_authentication_return_parameter` (see on page 99) enumerations to help user to decode Authentication service responses.

Example

The following example shows the use of the service function `UDS_SvcAuthenticationVPOWNU_2020` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```
uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/verifyProofOfOwnershipUnidirectional message
```

```
uint8_t algorithm_indicator[16] = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
    0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F };
uint8_t proof_of_ownership_client[2] = { 0x12, 0x34 };
uint16_t proof_of_ownership_client_size = 2;
uint8_t challenge_client[2] = { 0x56, 0x78 };
uint16_t challenge_client_size = 2;
uint8_t additional_parameter[2] = { 0x9A, 0xBC };
uint16_t additional_parameter_size = 2;
result = UDS_SvcAuthenticationVPOWNU_2020(PCANTP_HANDLE_USBBUS1, config, &request,
    algorithm_indicator, proof_of_ownership_client, proof_of_ownership_client_size,
    challenge_client, challenge_client_size, additional_parameter,
    additional_parameter_size);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);
```

See also: [UDS_WaitForService_2013](#) on page 656.

Class-method version: [SvcAuthenticationVPOWNU_2020](#) on page 579.

3.8.79 UDS_SvcAuthenticationVPOWNB_2020

Writes a UDS request according to the Authentication service's specifications (ISO-14229-1:2020). The subfunction verifyProofOfOwnershipBidirectional is implicit.

Syntax

C

```
uds_status UDS_SvcAuthenticationVPOWNB_2020(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t *algorithm_indicator,
    uint8_t *proof_of_ownership_client,
    uint16_t proof_of_ownership_client_size,
    uint8_t *challenge_client,
    uint16_t challenge_client_size,
    uint8_t *additional_parameter,
    uint16_t additional_parameter_size);
```

C++

```
uds_status UDS_SvcAuthenticationVPOWNB_2020(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request,
    uint8_t *algorithm_indicator,
    uint8_t *proof_of_ownership_client,
    uint16_t proof_of_ownership_client_size,
    uint8_t *challenge_client,
```



```
uint16_t challenge_client_size,
uint8_t *additional_parameter = nullptr,
uint16_t additional_parameter_size = 0);
```

Parameters

| Parameter | Description |
|--------------------------------|---|
| channel | The handle of a PUDS channel (see <code>cantp_handle</code> on page 105). |
| request_config | Message request configuration (see <code>uds_msgconfig</code> on page 27). |
| out_msg_request | Output, request message created and sent by the function (see <code>uds_msg</code> on page 21). |
| algorithm_indicator | Buffer of 16 bytes containing the algorithm indicator. |
| proof_of_ownership_client | Buffer containing the proof of ownership of the client. |
| proof_of_ownership_client_size | Size in bytes of the proof of ownership buffer. |
| challenge_client | Buffer containing the challenge of the client. |
| challenge_client_size | Size in bytes of the challenge buffer. |
| additional_parameter | Buffer containing additional parameters. |
| additional_parameter | Size in bytes of the additional parameter buffer. |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | One of the given parameters is not valid. |
| <code>PUDS_STATUS_MAPPING_NOT_INITIALIZED</code> | The mapping given in the message configuration is unknown. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |


Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

The PCAN-UDS 2.x API provides `uds_svc_authentication_subfunction` (see on page 98) and `uds_svc_authentication_return_parameter` (see on page 99) enumerations to help user to decode Authentication service responses.

Example

The following example shows the use of the service function `UDS_SvcAuthenticationVPOWNB_2020` on the channel `PCANTP_HANDLE_USBUSB1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```

uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/verifyProofOfOwnershipBidirectional message
uint8_t algorithm_indicator[16] = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
    0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F };
uint8_t proof_of_ownership_client[2] = { 0x12, 0x34 };
uint16_t proof_of_ownership_client_size = 2;
uint8_t challenge_client[2] = { 0x56, 0x78 };
uint16_t challenge_client_size = 2;
uint8_t additional_parameter[2] = { 0x9A, 0xBC };
uint16_t additional_parameter_size = 2;
result = UDS_SvcAuthenticationVPOWNB_2020(PCANTP_HANDLE_USBBUS1, config, &request,
    algorithm_indicator, proof_of_ownership_client, proof_of_ownership_client_size,
    challenge_client, challenge_client_size, additional_parameter,
    additional_parameter_size);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
        &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: [UDS_WaitForService_2013](#) on page 656.

Class-method version: [SvcAuthenticationVPOWNB_2020](#) on page 596.

3.8.80 UDS_SvcAuthenticationAC_2020

Writes a UDS request according to the Authentication service's specifications (ISO-14229-1:2020). The subfunction authenticationConfiguration is implicit.

Syntax

C/C++

```
uds_status UDS_SvcAuthenticationAC_2020(
    cantp_handle channel,
    uds_msgconfig request_config,
    uds_msg *out_msg_request) ;
```

Parameters

| Parameter | Description |
|-----------------|--|
| channel | The handle of a PUDS channel (see cantp_handle on page 105). |
| Request_config | Message request configuration (see uds_msgconfig on page 27). |
| Out_msg_request | Output, request message created and sent by the function (see uds_msg on page 21). |

Returns

The return value is a `uds_status` code. `PUDS_STATUS_OK` is returned on success. The typical errors in case of failure are:

| | |
|--|---|
| <code>PUDS_STATUS_NOT_INITIALIZED</code> | Indicates that the given PUDS channel was not found in the list of reserved channels of the calling application. |
| <code>PUDS_STATUS_PARAM_INVALID_VALUE</code> | The request configuration is not valid (see uds_msgconfig on page 27), or the given message buffer is null. |
| <code>PUDS_STATUS_SERVICE_ALREADY_PENDING</code> | A message with the same service identifier is already pending in the reception queue, the user must read a response for his previous request before or clear the reception queues with <code>UDS_Reset_2013</code> (see <code>UDS_Reset_2013</code> on page 651). |
| <code>PUDS_STATUS_NO_MEMORY</code> | Failed to allocate memory and copy the new message in the transmission queue. |
| <code>PUDS_STATUS_MESSAGE_BUFFER_ALREADY_USED</code> | The given message buffer is already allocated, user must release the buffer before reusing it (see <code>UDS_MsgFree_2013</code> on page 645). |


Remarks

This function creates a new message using the given message configuration and sets the given data according to the service's specifications. It then writes the message to the transmit queue. Once processed, this request message should be released (see `UDS_MsgFree_2013` on page 645).

The PCAN-UDS 2.x API provides `uds_svc_authentication_subfunction` (see on page 98) and `uds_svc_authentication_return_parameter` (see on page 99) enumerations to help user to decode Authentication service responses.

Example

The following example shows the use of the service function `UDS_SvcAuthenticationAC_2020` on the channel `PCANTP_HANDLE_USBBUS1`. A UDS physical service request is transmitted, and the `UDS_WaitForService_2013` function is called to get the response. Depending on the result, a message will be shown to the user.

 **Note:** It is assumed that the channel was already initialized.

C/C++

```

uds_status result;
uds_msg request;
uds_msg request_confirmation;
uds_msg response;
uds_msgconfig config;

memset(&request, 0, sizeof(request));
memset(&request_confirmation, 0, sizeof(request_confirmation));
memset(&response, 0, sizeof(response));
memset(&config, 0, sizeof(config));

// Set request message configuration
config.can_id = PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.nai.extension_addr = 0x0;
config.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
config.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
config.type = PUDS_MSGTYPE_USDT;

// Sends a physical Authentication/authenticationConfiguration message
result = UDS_SvcAuthenticationAC_2020(PCANTP_HANDLE_USBBUS1, config, &request);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    result = UDS_WaitForService_2013(PCANTP_HANDLE_USBBUS1, &request, &response,
    &request_confirmation);
if (UDS_StatusIsOk_2013(result, PUDS_STATUS_OK, false))
    printf("Response was received\n");
else
    // An error occurred
    printf("An error occurred\n");

// Free structures
UDS_MsgFree_2013(&request);
UDS_MsgFree_2013(&response);
UDS_MsgFree_2013(&request_confirmation);

```

See also: [UDS_WaitForService_2013](#) on page 656.

Class-method version: [SvcAuthenticationAC_2020](#) on page 607.

3.9 Definitions





The PCAN-UDS 2.x API defines the following values:

| Name | Description |
|--|---|
| Messages Related Definitions | Defines values and constants for the members of the <code>uds_msg</code> structure. |
| PCAN-UDS 2.x Service Parameter Definitions | Defines constants to be used with some UDS service functions. |



3.9.1 Messages Related Definitions

The following definitions are related to messages.

Data information:

| | Type | Constant | Value | Description |
|---|--------|--------------------------------|------------|--|
|  | Int32 | PCANTP_MAX_LENGTH_CAN_STANDARD | 0x8 | Maximum size of a CAN (non-FD) frame (8 bytes) |
|  | Int32 | PCANTP_MAX_LENGTH_CAN_FD | 0x40 | Maximum size of a CAN FD frame (64 bytes) |
|  | Int32 | PCANTP_MAX_LENGTH_ISOTP2004 | 0xFFFF | Maximum size of an ISO-TP rev. 2004 frame (4095 bytes) |
|  | UInt32 | PCANTP_MAX_LENGTH_ISOTP2016 | 0xFFFFFFFF | Maximum size of an ISO-TP rev. 2016 frame (4294967295 bytes) |

Other values:


| | Type | Constant | Value | Description |
|---|-------|---------------------------|------------|---|
|  | Int32 | PUDS_NRC_EXTENDED_TIMING | 0x78 (120) | Negative UDS response code stating that the server/ECU requests more time to transmit a response. |
|  | Int32 | PUDS_SI_POSITIVE_RESPONSE | 0x40 (64) | Service Identifier offset for positive response message (i.e.: Service Identifier for a positive response to a UDS request is 0x40 + Service Request Identifier). |











See also: `uds_msg` on page 21.

3.9.2 PCAN-UDS 2.x Service Parameter Definitions

SecurityAccess Type Definitions

The following constants are a reminder of some Request Seed and Send Key values.

 **Note:** Ranges for system-supplier-specific use and ISO/SAE reserved values have been discarded.








| | Type | Constant | Value | Description |
|---|------|---------------------------|-----------|----------------------------|
|  | Byte | PUDS_SVC_PARAM_SA_RSD_1 | 0x01 (1) | Request seed (odd numbers) |
|  | Byte | PUDS_SVC_PARAM_SA_RSD_3 | 0x03 (3) | Request seed (odd numbers) |
|  | Byte | PUDS_SVC_PARAM_SA_RSD_5 | 0x05 (5) | Request seed (odd numbers) |
|  | Byte | PUDS_SVC_PARAM_SA_RSD_MIN | 0x07 (7) | Request seed (odd numbers) |
|  | Byte | PUDS_SVC_PARAM_SA_RSD_MAX | 0x5F (95) | Request seed (odd numbers) |
|  | Byte | PUDS_SVC_PARAM_SA_SK_2 | 0x02 (2) | Send Key (even numbers) |
|  | Byte | PUDS_SVC_PARAM_SA_SK_4 | 0x04 (4) | Send Key (even numbers) |
|  | Byte | PUDS_SVC_PARAM_SA_SK_6 | 0x06 (6) | Send Key (even numbers) |
|  | Byte | PUDS_SVC_PARAM_SA_SK_MIN | 0x08 (8) | Send Key (even numbers) |
|  | Byte | PUDS_SVC_PARAM_SA_SK_MAX | 0x60 (96) | Send Key (even numbers) |

See also: `UDS_SvcSecurityAccess_2013` on page 664, `SvcSecurityAccess_2013` on page 267.

CommunicationControl Communication Type Definitions

The communication type parameter is a bit-code value that allows control of multiple communication types at the same time. The following table lists the coding of the communication type data parameter:










- the bit-encoded low nibble of this byte represents the communication types,
- the high nibble defines which of the subnets connected to the receiving node shall be disabled/enabled

| | Type | Constant | Value | Description |
|---|------|--------------------------------------|------------|---|
|  | Byte | PUDS_SVC_PARAM_CC_FLAG_APPL | 0x01 (1) | CommunicationType Flag: Application (01b) |
|  | Byte | PUDS_SVC_PARAM_CC_FLAG_NWM | 0x02 (2) | CommunicationType Flag: NetworkManagement (10b) |
|  | Byte | PUDS_SVC_PARAM_CC_FLAG_DESCIRNCN | 0x00 (0) | CommunicationType Flag: Disable/Enable specified communicationType (see Flags APPL/NMW) |
|  | Byte | PUDS_SVC_PARAM_CC_FLAG_DENWRIRO | 0xF0 (240) | CommunicationType Flag: Disable/Enable network which request is received on. |
|  | Byte | PUDS_SVC_PARAM_CC_FLAG_DESNIBNN_MIN | 0x10 (16) | CommunicationType Flag: Disable/Enable specific network identified by network number (minimum value). |
|  | Byte | PUDS_SVC_PARAM_CC_FLAG_DESNIBNN_MAX | 0xE0 (224) | CommunicationType Flag: Disable/Enable specific network identified by network number (maximum value). |
|  | Byte | PUDS_SVC_PARAM_CC_FLAG_DESNIBNN_MASK | 0xF0 (240) | CommunicationType Flag: Mask for DESNIBNN bits. |








See also: [UDS_SvcCommunicationControl_2013](#) on page 666, [SvcCommunicationControl_2013](#) on page 276.

ResponseOnEvent Service Definitions




The following table defines the expected size of the [EventTypeRecord](#) based on the [EventType](#) (see [uds_svc_param_roe](#) on page 77):

| | Type | Constant | Value | Description |
|---|------|--------------------------------|-------|---|
|  | Byte | PUDS_SVC_PARAM_ROE_STPROE_LEN | 0 | Expected size of EventTypeRecord for ROE_STPROE. |
|  | Byte | PUDS_SVC_PARAM_ROE_ONDTCS_LEN | 1 | Expected size of EventTypeRecord for ROE_ONDTCS. |
|  | Byte | PUDS_SVC_PARAM_ROE_OTI_LEN | 1 | Expected size of EventTypeRecord for ROE_OTI. |
|  | Byte | PUDS_SVC_PARAM_ROE_OCODID_LEN | 2 | Expected size of EventTypeRecord for ROE_OCODID. |
|  | Byte | PUDS_SVC_PARAM_ROE_RAE_LEN | 0 | Expected size of EventTypeRecord for ROE_RAE. |
|  | Byte | PUDS_SVC_PARAM_ROE_STRTROE_LEN | 0 | Expected size of EventTypeRecord for ROE_STRTROE. |
|  | Byte | PUDS_SVC_PARAM_ROE_CLRROE_LEN | 0 | Expected size of EventTypeRecord for ROE_CLRROE. |
|  | Byte | PUDS_SVC_PARAM_ROE_OCOV_LEN | 10 | Expected size of EventTypeRecord for ROE_OCOV. |
|  | Byte | PUDS_SVC_PARAM_ROE_RMRDOSC_LEN | 1 | Expected size of EventTypeRecord for ROE_RMRDOSC. |

The following table defines the [EventWindowTime](#) parameters to be used with ResponseOnEvent service:

| | Type | Constant | Value | Description |
|---|------|------------------------------|-------|---|
|  | Byte | PUDS_SVC_PARAM_ROE_EWT_ITTR | 2 | Infinite Time to Response (eventWindowTime parameter). |
|  | Byte | PUDS_SVC_PARAM_ROE_EWT_SEWT | 3 | Short event window time (eventWindowTime parameter). |
|  | Byte | PUDS_SVC_PARAM_ROE_EWT_MEWT | 4 | Medium event window time (eventWindowTime parameter). |
|  | Byte | PUDS_SVC_PARAM_ROE_EWT_LEWT | 5 | Long event window time (eventWindowTime parameter). |
|  | Byte | PUDS_SVC_PARAM_ROE_EWT_PWT | 6 | Power window time (eventWindowTime parameter). |
|  | Byte | PUDS_SVC_PARAM_ROE_EWT_IWT | 7 | Ignition window time (eventWindowTime parameter). |
|  | Byte | PUDS_SVC_PARAM_ROE_EWT_MTEWT | 8 | Manufacturer trigger event window time (eventWindowTime parameter). |



The following table defines the `onTimerInterrupt` timer parameters to be used with `ResponseOnEvent` service:

| | Type | Constant | Value | Description |
|---|------|------------------------------------|-------|---|
|  | Byte | PUDS_SVC_PARAM_ROE_OTI_SLOW_RATE | 1 | Slow rate (onTimerInterrupt parameter). |
|  | Byte | PUDS_SVC_PARAM_ROE_OTI_MEDIUM_RATE | 2 | Medium rate (onTimerInterrupt parameter). |
|  | Byte | PUDS_SVC_PARAM_ROE_OTI_FAST_RATE | 3 | Fast rate (onTimerInterrupt parameter). |

See also: `UDS_SvcResponseOnEvent_2013` on page 677, `SvcResponseOnEvent_2013` on page 314.

ClearDiagnosticInformation Group of DTC Definitions






The following table lists constants to be used as the `GroupOfDTC` parameter with the `ClearDiagnosticInformation` service:

| | Type | Constant | Value | Description |
|---|--------|---------------------------|------------|--|
|  | UInt32 | PUDS_SVC_PARAM_CDI_ERS | 0x000000 | Emissions-related systems group of DTCs. |
|  | UInt32 | PUDS_SVC_PARAM_CDI_AGDTCT | 0xFFFFFFFF | All Groups of DTCs. |

See also: `UDS_SvcClearDiagnosticInformation_2013` on page 700, `SvcClearDiagnosticInformation_2013` on page 387.

SecuredDataTransmission Administrative Parameter Flags Definitions

The following table lists constants to be used as flags for the administrative parameter with the `SecuredDataTransmission` service(ISO-14229-1:2020):

| | Type | Constant | Value | Description |
|---|------|---|-------|---|
|  | Byte | PUDS_SVC_PARAM_APAR_REQUEST_MSG_FLAG | 0x01 | The message is a request message. |
|  | Byte | PUDS_SVC_PARAM_APAR_PRE_ESTABLISHED_KEY_FLAG | 0x08 | A pre - established key is used. |
|  | Byte | PUDS_SVC_PARAM_APAR_ENCRYPTED_MSG_FLAG | 0x10 | Message is encrypted. |
|  | Byte | PUDS_SVC_PARAM_APAR_SIGNED_MSG_FLAG | 0x20 | Message is signed. |
|  | Byte | PUDS_SVC_PARAM_APAR_REQUEST_RESPONSE_SIGNATURE_FLAG | 0x40 | Signature on the response is requested. |

See also: `UDS_SvcSecuredDataTransmission_2020` on page 672, `SvcSecuredDataTransmission_2020` on page 298.

4 Additional Information

PCAN is the platform for PCAN-OBDI, PCAN-UDS 2.x, and PCAN-Basic. In the following, topics there is an overview of PCAN and the fundamental practice with the interface DLL CanApi2 (PCAN-API).

| Topics | Description |
|---|--|
| PCAN Fundamentals | This section contains an introduction to PCAN |
| PCAN-Basic | This section contains general information about the PCAN-Basic API |
| UDS and ISO-TP Network Addressing Information | This section contains general information about the ISO-TP network addressing format |

4.1 PCAN Fundamentals

PCAN is a synonym for PEAK CAN APPLICATIONS and is a flexible system for planning, developing, and using a CAN Bus System. Developers as well as end-users are getting a helpful and powerful product.

The basis for the communication between PCs and external hardware via CAN is a series of Windows Kernel-Mode Drivers (Virtual Device Drivers) e.g. PCAN_USB.SYS, PCAN_PCI.SYS, PCAN_XXX.SYS. These drivers are the core of a complete CAN environment on a PC running Windows and work as interfaces between CAN software and PC-based CAN hardware. The drivers manage the entire data flow of every CAN device connected to the PC.

A user or administrator of a CAN installation gets access via the PCAN-Clients (short: Clients). Several parameters of processes can be visualized and changed with their help. The drivers allow the connection of several Clients at the same time.

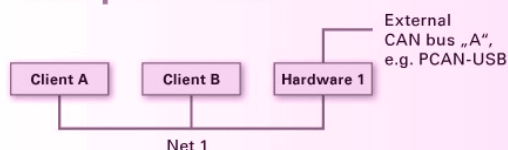
Furthermore, several hardware components based on the SJA1000 CAN controller are supported by a PCAN driver. So-called Nets provide the logical structure for CAN buses, which are virtually extended into the PC. On the hardware side, several Clients can be connected, too. The following figures demonstrate different possibilities of Net configurations (also realizable at the same time).

Following rules apply to PCAN clients, nets, and hardware:

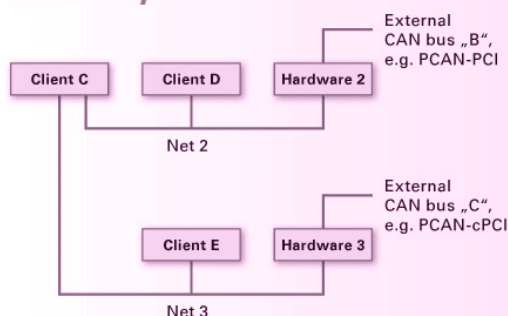
- └ One Client can be connected to several Nets
- └ One Net provides several Clients
- └ One piece of hardware belongs to one Net
- └ One Net can include none or one piece of hardware
- └ A message from a transmitting Client is carried on to every other connected Client, and to the external bus via the connected CAN hardware
- └ A message received by the CAN hardware is received by every connected Client. However, Clients react only on those messages that pass their acceptance filter

Users of PCAN-View 3 do not have to define and manage Nets. If PCAN-View is instructed to connect directly to a PCAN hardware, the application automatically creates a Net for the selected hardware and automatically establishes a connection with this Net.

Example network



Gateway



Internal network



See also: PCAN-Basic on page 769, ISO-TP Network Addressing Format on page 773.

4.2 PCAN-Basic

PCAN-Basic is a programming application interface (API) for CAN communication using the PCAN system of the company PEAK-System Technik GmbH. It comprises of a collection of Windows Device Drivers which allow the real-time connection of Windows applications to all CAN busses physically connected to a PC. PCAN-Basic supports connecting several CAN channels simultaneously, from the same or different types of devices. The following list shows the PCAN-Channels that can be connected per PCAN-Device:

| | PCAN-ISA | PCAN-Dongle | PCAN-PCI | PCAN-USB | PCAN-PC-Card | PCAN-LAN |
|--------------------|----------|-------------|----------|----------|--------------|----------|
| Number of channels | 8 | 1 | 16 | 16 | 2 | 16 |

Using PCAN-Basic

PCAN-Basic offers the possibility to use several PCAN channels within the same application easily. The communication process is divided in 3 phases: initialization, interaction, and finalization of a PCAN channel.

Initialization: To do CAN communication using a channel, it is necessary to first initialize it. This is done making a call to the function `CAN_Initialize` (**class-method:** `Initialize`) or `CAN_InitializeFD` (**class-method:** `InitializeFD`) in case of FD communication.

Interaction: After successful initialization, a channel is ready to communicate with the connected CAN bus. Further configuration is not needed. The functions `CAN_Read` and `CAN_Write` (**class-methods:** `Read` and `Write`) can then be used to read and write CAN messages. If the used channel is FD-capable and has been initialized using `CAN_InitializeFD`, then the functions to use are `CAN_ReadFD` and `CAN_WriteFD` (**class-methods:** `ReadFD` and `WriteFD`). An extra configuration can be made to improve a communication session, like changing the message filter to target-specific messages.

Finalization: When the communication is finished, the function `CAN_Uninitialize` (**class-method:** `Uninitialize`) should be called to release the PCAN channel and the resources allocated to it. In this way, the channel is marked as "Free" and can be used by other applications.

Hardware and Drivers

Overview of the current PCAN hardware and device drivers:

| Hardware | Plug and Play Hardware | Driver |
|------------------------|------------------------|--------------|
| PCAN-Dongle | No | Pcan_dng.sys |
| PCAN-ISA | No | Pcan_isa.sys |
| PCAN-PC/104 | No | Pcan_isa.sys |
| PCAN-PCI | Yes | Pcan_pci.sys |
| PCAN-PCI Express | Yes | Pcan_pci.sys |
| PCAN-PCI Express FD | Yes | Pcan_pci.sys |
| PCAN-cPCI | Yes | Pcan_pci.sys |
| PCAN-miniPCI | Yes | Pcan_pci.sys |
| PCAN-miniPCle | Yes | Pcan_pci.sys |
| PCAN-miniPCle FD | Yes | Pcan_pci.sys |
| PCAN-M.2 | Yes | Pcan_pci.sys |
| PCAN-Chip PCle FD | Yes | Pcan_pci.sys |
| PCAN-PC/104-Plus | Yes | Pcan_pci.sys |
| PCAN-PC/104-Plus Quad | Yes | Pcan_pci.sys |
| PCAN-PC/104-Express | Yes | Pcan_pci.sys |
| PCAN-PC/104-Express FD | Yes | Pcan_pci.sys |
| PCAN-ExpressCard | Yes | Pcan_pci.sys |

| Hardware | Plug and Play Hardware | Driver |
|----------------------------------|------------------------|--------------|
| PCAN-ExpressCard 34 | Yes | Pcan_pci.sys |
| PCAN-USB | Yes | Pcan_usb.sys |
| PCAN-USB FD | Yes | Pcan_usb.sys |
| PCAN-USB Pro | Yes | Pcan_usb.sys |
| PCAN-USB Pro FD | Yes | Pcan_usb.sys |
| PCAN-USB Hub | Yes | Pcan_usb.sys |
| PCAN-USB X6 | Yes | Pcan_usb.sys |
| PCAN-Chip USB | Yes | Pcan_usb.sys |
| PCAN-PC Card | Yes | Pcan_pcc.sys |
| PCAN-Ethernet Gateway DR | Yes | Pcan_lan.sys |
| PCAN-Wireless Gateway DR | Yes | Pcan_lan.sys |
| PCAN-Wireless Gateway | Yes | Pcan_lan.sys |
| PCAN-Wireless Automotive Gateway | Yes | Pcan_lan.sys |

See also: PCAN Fundamentals on page 768, ISO-TP Network Addressing Format on page 773.

4.3 UDS and ISO-TP Network Addressing Information

The UDS API makes use of the PCAN-ISO-TP 3.x API to receive and transmit UDS messages. When a PUDS channel is initialized, the PCAN-ISO-TP 3.x API is configured to allow the following communications:

- └ Functional request using 11 bits CAN identifier and normal addressing, from External Test Equipment address (`PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT`) to OBD functional address (`PUDS_ISO_15765_4_ADDR_OBD_FUNCTIONAL`):
 - CAN ID `0x7DF` (`PUDS_ISO_15765_4_CAN_ID_FUNCTIONAL_REQUEST`) from Source `0xF1` to Target `0x33`
- └ Physical requests and responses using 11 bits CAN identifier and normal addressing, between the External Test Equipment address (`PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT`) and standard ECU addresses (ECU #1 to #8):
 - ECU #1
 - Request: CAN ID `0x7E8` (`PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_1`) from Source `0xF1` to Target `0x01`
 - Response: CAN ID `0x7E0` (`PUDS_ISO_15765_4_CAN_ID_PHYSICAL_RESPONSE_1`) from Source `0x01` to Target `0xF1`
 - ECU #2:
 - Request: CAN ID `0x7E9` (`PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_2`) from Source `0xF1` to Target `0x01`
 - Response: CAN ID `0x7E1` (`PUDS_ISO_15765_4_CAN_ID_PHYSICAL_RESPONSE_2`) from Source `0x01` to Target `0xF1`
 - ECU #3:
 - Request: CAN ID `0x7EA` (`PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_3`) from Source `0xF1` to Target `0x01`
 - Response: CAN ID `0x7E2` (`PUDS_ISO_15765_4_CAN_ID_PHYSICAL_RESPONSE_3`) from Source `0x01` to Target `0xF1`

- ECU #4:
 - Request: CAN ID 0x7EB (PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_4) from Source 0xF1 to Target 0x01
 - Response: CAN ID 0x7E3 (PUDS_ISO_15765_4_CAN_ID_PHYSICAL_RESPONSE_4) from Source 0x01 to Target 0xF1
- ECU #5:
 - Request: CAN ID 0x7EC (PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_5) from Source 0xF1 to Target 0x01
 - Response: CAN ID 0x7E4 (PUDS_ISO_15765_4_CAN_ID_PHYSICAL_RESPONSE_5) from Source 0x01 to Target 0xF1
- ECU #6:
 - Request: CAN ID 0x7ED (PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_6) from Source 0xF1 to Target 0x01
 - Response: CAN ID 0x7E5 (PUDS_ISO_15765_4_CAN_ID_PHYSICAL_RESPONSE_6) from Source 0x01 to Target 0xF1
- ECU #7:
 - Request: CAN ID 0x7EE (PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_7) from Source 0xF1 to Target 0x01
 - Response: CAN ID 0x7E6 (PUDS_ISO_15765_4_CAN_ID_PHYSICAL_RESPONSE_7) from Source 0x01 to Target 0xF1
- ECU #8:
 - Request: CAN ID 0x7EF (PUDS_ISO_15765_4_CAN_ID_PHYSICAL_REQUEST_8) from Source 0xF1 to Target 0x01
 - Response: CAN ID 0x7E7 (PUDS_ISO_15765_4_CAN_ID_PHYSICAL_RESPONSE_8) from Source 0x01 to Target 0xF1

← Communications with 29 bits CAN identifier and FIXED NORMAL addressing format (where XX is Target Address and YY Source address and respectively physical/functional addressing):

- CAN ID 0x00DAXXYY/0x00DBXXYY (data link layer priority 0)
- CAN ID 0x04DAXXYY/0x04DBXXYY (data link layer priority 1)
- CAN ID 0x08DAXXYY/0x08DBXXYY (data link layer priority 2)
- CAN ID 0x0CDAXXYY/0x0CDBXXYY (data link layer priority 3)
- CAN ID 0x10DAXXYY/0x10DBXXYY (data link layer priority 4)
- CAN ID 0x14DAXXYY/0x14DBXXYY (data link layer priority 5)
- CAN ID 0x18DAXXYY/0x18DBXXYY (data link layer priority 6)
- CAN ID 0x1CDAXXYY/0x1CDBXXYY (data link layer priority 7)

← Communications with 29 bits CAN identifier and MIXED addressing format (where XX is Target Address and YY Source address and respectively physical/functional addressing):

- CAN ID 0x00CEXXYY/0x00CDXXYY (data link layer priority 0)
- CAN ID 0x04CEXXYY/0x04CDXXYY (data link layer priority 1)
- CAN ID 0x08CEXXYY/0x08CDXXYY (data link layer priority 2)
- CAN ID 0x0CCEXXYY/0x0CCDXXYY (data link layer priority 3)
- CAN ID 0x10CEXXYY/0x10CDXXYY (data link layer priority 4)

- CAN ID 0x14CEXXYY/0x14CDXXYY (data link layer priority 5)
 - CAN ID 0x18CEXXYY/0x18CDXXYY (data link layer priority 6)
 - CAN ID 0x1CCEXXYY/0x1CCDXXYY (data link layer priority 7)
- └ Communications with 29 bits CAN identifier and ENHANCED addressing format (where YYY is Target Address and XXX Source address, addresses are encoded on 11 bits):
- CAN ID 0x03XXXXYY (data link layer priority 0)
 - CAN ID 0x07XXXXYY (data link layer priority 1)
 - CAN ID 0x0BXXXXYY (data link layer priority 2)
 - CAN ID 0x0FXXXXYY (data link layer priority 3)
 - CAN ID 0x13XXXXYY (data link layer priority 4)
 - CAN ID 0x17XXXXYY (data link layer priority 5)
 - CAN ID 0x1BXXXXYY (data link layer priority 6)
- └ CAN ID 0x1FXXXXYY (data link layer priority 7)

If an application requires other communication settings, it will have to be set with through the function `UDS_AddMapping_2013` and the method `AddMapping_2013`. See also `uds_mapping` on page 25.

Alternatively, it is also possible to directly use PCAN-ISO-TP 3.x API. Once a PUDS channel is initialized, PCAN-ISO-TP 3.x specific functions (like `CANTP_Write_2016` and `CANTP_Read_2016`) can be called.

4.3.1 Usage in a Non-Standardized Context

Default Source Address

When a UDS channel is initialized, the default source address for this new node is the standardized “Test Equipment” address: 0xF1 (see `uds_address` on page 56). This means that all UDS messages received by this node whose target address does not match this source address will be discarded. If your application makes communications with a different source address, you need to specify that address to the API by using the parameter `PUDS_PARAMETER_SERVER_ADDRESS` (see on page 46):

```
cantp_handle channel;

// ...

// Define server address
uint8_t param = 0xA1;
uds_status status = UDS_SetValue_2013(channel, PUDS_PARAMETER_SERVER_ADDRESS, &param,
    sizeof(param));
// check status and proceed...
```

Alternatively, it is possible to listen to multiple addresses via the parameter `PUDS_PARAMETER_ADD_LISTENED_ADDRESS` (see on page 48):

```
cantp_handle channel;

// ...

// Listen to address 0x11
uint16_t param = 0x11;
uds_status status = UDS_SetValue_2013(channel, PUDS_PARAMETER_ADD_LISTENED_ADDRESS, &param,
    sizeof(param));
// check status and proceed...
```

Removing Default Mappings

- If you need to set different network addressing information to the already defined mappings (for instance to use the standardized CAN identifiers with a different Source Address), you first need to remove the existing mapping(s).

```
cantp_handle channel = PCANTP_HANDLE_USBBUS1;
uds_mapping mappings[256];
uint16_t count;
uds_status status;
uint16_t i;

status = UDS_GetMappings_2013(channel, mappings, 256, &count);
if(!UDS_StatusIsOk_2013(status, PUDS_STATUS_OK, false))
    printf("Get mappings failed.\n");
for (i = 0; i < count; i++)
{
    status = UDS_RemoveMapping_2013(channel, mappings[i]);
    if (!UDS_StatusIsOk_2013(status, PUDS_STATUS_OK, false))
        printf("Remove mapping failed.\n");
}
```

- Standardized 29 bits CAN identifiers do not use mappings, if you want to override those CAN identifiers, simply use that value when adding a mapping.

If you need to disable completely the handling of standardized ISO-TP 29 bits CAN identifiers, consider disabling their support directly in PCAN-ISO-TP API using the following parameters:

- `PCANTP_PARAMETER_SUPPORT_29B_ENHANCED`
- `PCANTP_PARAMETER_SUPPORT_29B_FIXED_NORMAL`
- `PCANTP_PARAMETER_SUPPORT_29B_MIXED`

4.3.2 ISO-TP Network Addressing Format

ISO-TP specifies three addressing formats to exchange data: normal, extended, and mixed addressing. Each addressing requires a different number of CAN frame data bytes to encapsulate the addressing information associated with the data to be exchanged.

The following table sums up the mandatory configuration to the PCAN-ISO-TP 3.x API for each addressing format:

| Addressing format | CAN identifier length | Mandatory configuration steps |
|---|-----------------------|---|
| Normal addressing PCANTP_FORMAT_NORMAL | 11 bits | Define mappings with UDS_AddMapping_2013 |
| | 29 bits | Define mappings with UDS_AddMapping_2013 |
| Normal fixed addressing PCANTP_FORMAT_FIXED_NORMAL | 11 bits | Addressing is invalid |
| | 29 bits | - |
| Extended addressing PCANTP_FORMAT_EXTENDED | 11 bits | Define mappings with UDS_AddMapping_2013 |
| | 29 bits | Define mappings with UDS_AddMapping_2013 |
| Mixed addressing PCANTP_FORMAT_MIXED | 11 bits | Define mappings with UDS_AddMapping_2013 |
| | 29 bits | - |
| Enhanced addressing PCANTP_ISOTP_FORMAT_ENHANCED | 11 bits | Addressing is invalid. |
| | 29 bits | - Note: With ISO-15765:2016, this addressing is considered deprecated and disabled by default. See PCAN-ISO-TP documentation on parameter: PCANTP_PARAMETER_SUPPORT_29B_ENHANCED. |

A mapping allows an PCANTP node to identify and decode CAN Identifiers, it binds a CAN identifier to an PCANTP network address information. CAN messages that cannot be identified are ignored by the API.

Mappings involving physically addressed communication are most usually defined in pairs: the first mapping defines outgoing communication (i.e. request messages from node A to node B) and the second to match incoming communication (i.e. responses from node B to node A).

Functionally addressed communication requires one mapping to transmit functionally addressed messages (i.e. request messages from node A to any node) and as many mappings as responding nodes (i.e. responses from nodes B, C, etc. to node A).

4.3.3 PCAN-UDS 2.x Example

Configuration of Mappings

The following C/C++ example shows how to define 4 mappings with the PCAN-UDS 2.x API on the Tester Client side to:

- Transmit physical message to ECU #1 with CAN identifier 0x326
- Receive physical message from ECU #1 with CAN identifier 0x626
- Transmit functional message on CAN ID 0x200
- Receive UUDT message from the ECU #1 with CAN identifier 0x526

```
cantp_handle channel;
uds_status status;
uds_mapping mapping;

// Note: Channel is properly initialized with UDS_Initialize_2013 (..)
// [...]

// Add mapping for physical request from external equipment to ECU_1: ID=0x326
memset(&mapping, 0, sizeof(mapping));
mapping.can_id = 0x326;
mapping.can_id_flow_ctrl = 0x626;
mapping.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
mapping.can_tx_dlc = 8;
mapping.nai.extension_addr = 0;
mapping.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
mapping.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
mapping.nai.target_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
mapping.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
status = UDS_AddMapping_2013(channel, &mapping);
if (!UDS_StatusIsOk_2013(status, PUDS_STATUS_OK, false))
    printf("Add mapping failed.\n");

// Add mapping for physical response from ECU_#X to external equipment: ID=0x626
memset(&mapping, 0, sizeof(mapping));
mapping.can_id = 0x626;
mapping.can_id_flow_ctrl = 0x326;
mapping.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
mapping.can_tx_dlc = 8;
mapping.nai.extension_addr = 0;
mapping.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
mapping.nai.source_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
mapping.nai.target_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
mapping.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
status = UDS_AddMapping_2013(channel, &mapping);
if (!UDS_StatusIsOk_2013(status, PUDS_STATUS_OK, false))
    printf("Add mapping failed.\n");

// Add mapping for functional request from external equipment: ID=0x200
memset(&mapping, 0, sizeof(mapping));
mapping.can_id = 0x200;
mapping.can_id_flow_ctrl = (uint32_t)-1;
```

```

mapping.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
mapping.can_tx_dlc = 8;
mapping.nai.extension_addr = 0;
mapping.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
mapping.nai.source_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
mapping.nai.target_addr = PUDS_ISO_15765_4_ADDR_OBD_FUNCTIONAL;
mapping.nai.target_type = PCANTP_ISOTP_ADDRESSING_FUNCTIONAL;
status = UDS_AddMapping_2013(channel, &mapping);
if (!UDS_StatusIsOk_2013(status, PUDS_STATUS_OK, false))
    printf("Add mapping failed.\n");

// Unacknowledged Unsegmented Data Transfer (UUDT) support:
// standard CAN ID without UDS Protocol Data Unit can be sent by ECU with service
readDataByPeriodicdataIdentifier
// Add mapping for UUDT physical response from ECU_1 to external equipment: ID=0x526
memset(&mapping, 0, sizeof(mapping));
mapping.can_id = 0x526;
mapping.can_id_flow_ctrl = (uint32_t)-1;
mapping.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
mapping.can_tx_dlc = 4;
mapping.nai.extension_addr = 0;
mapping.nai.protocol = PUDS_MSGPROTOCOL_ISO_15765_2_11B_NORMAL;
mapping.nai.source_addr = PUDS_ISO_15765_4_ADDR_ECU_1;
mapping.nai.target_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
mapping.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;
status = UDS_AddMapping_2013(channel, &mapping);
if (!UDS_StatusIsOk_2013(status, PUDS_STATUS_OK, false))
    printf("Add mapping failed.\n");
status = UDS_AddCanIdFilter_2013(channel, 0x526);
if (!UDS_StatusIsOk_2013(status, PUDS_STATUS_OK, false))
    printf("Add can identifier failed.\n");

```

UUDT Read/Write Example

The following C/C++ example shows Unacknowledged Unsegmented Data Transfer (UUDT), it writes from USB 1 UDS channel and reads the message from USB2 UDS channel.

```

uds_status status;
int tmp_buffer;
int count;
uds_msgconfig config;
uds_msg tx_msg;
uds_msg rx_msg;

cantp_handle channel_tx = PCANTP_HANDLE_USBBUS1;
cantp_handle channel_rx = PCANTP_HANDLE_USBBUS2;

memset(&config, 0, sizeof(config));
memset(&tx_msg, 0, sizeof(tx_msg));
memset(&rx_msg, 0, sizeof(rx_msg));

// Initializes UDS Communication for the transmitting channel
status = UDS_Initialize_2013(channel_tx, PCANTP_BAUDRATE_500K, (cantp_hwtype)0, 0, 0);
printf("Initialize UDS: %i\n", (int)status);

// Initializes UDS Communication for the receiving channel
status = UDS_Initialize_2013(channel_rx, PCANTP_BAUDRATE_500K, (cantp_hwtype)0, 0, 0);
printf("Initialize channel_rx: %i\n", (int)status);

// Define "channel_tx" Address as ECU #9
tmp_buffer = 0xF9;
status = UDS_SetValue_2013(channel_tx, PUDS_PARAMETER_SERVER_ADDRESS, &tmp_buffer, 1);
printf(" Set server address: %i (0x%02x)\n", (int)status, tmp_buffer);

```



```

// Define "channel_rx" Address as external equipment
tmp_buffer = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
status = UDS_SetValue_2013(channel_rx, PUDS_PARAMETER_SERVER_ADDRESS, &tmp_buffer, 1);
printf(" Set server address: %i (0x%02x)\n", (int)status, tmp_buffer);

// Prepare mapping configuration:
// UUDT physical response from ECU_#9 to external equipment: ID=0x526
config.can_id = 0x526;
config.can_msgtype = PCANTP_CAN_MSGTYPE_STANDARD;
config.type = PUDS_MSGTYPE_UUDT;
config.nai.extension_addr = 0;
config.nai.protocol = PUDS_MSGPROTOCOL_NONE;
config.nai.source_addr = 0xF9;
config.nai.target_addr = PUDS_ISO_15765_4_ADDR_TEST_EQUIPMENT;
config.nai.target_type = PCANTP_ISOTP_ADDRESSING_PHYSICAL;

// Add "channel_tx" can identifier filter (in order to send message)
// for UUDT physical response from ECU_#9 to external equipment: ID=0x526
status = UDS_AddCanIdFilter_2013(channel_tx, 0x526);
printf(" Add channel_tx can identifier filter: 0x%x\n", status);

// Add "channel_rx" can identifier filter (in order to receive message)
// for UUDT physical response from ECU_#9 to external equipment: ID=0x526
status = UDS_AddCanIdFilter_2013(channel_rx, 0x526);
printf(" Add channel_rx can identifier filter: 0x%x\n", status);

// Write a message from "channel_tx" to "channel_rx"
status = UDS_MsgAlloc_2013(&tx_msg, config, 6);
printf(" Allocate tx message: 0x%x\n", status);
if (UDS_StatusIsOk_2013)
{
    tx_msg.msg.msgdata.any->data[4] = 0xCA;
    tx_msg.msg.msgdata.any->data[5] = 0xB1;
    status = UDS_Write_2013(channel_tx, &tx_msg);
    printf(" UDS_Write_2013 UUDT: 0x%x\n", status);

    // Read message on channel Rx
    printf(" Reading message on channel_rx...\n");
    count = 0;
    do
    {
        count++;
        Sleep(100);
        status = UDS_Read_2013(channel_rx, &rx_msg, NULL, NULL);
    } while (status == PUDS_STATUS_NO_MESSAGE && count < 10);
    if (status == PUDS_STATUS_NO_MESSAGE)
        printf("Failed to read message on Channel RX !");
    else
    {
        // Received message will hold Network Address Information
        // as defined by the mapping.
        // The CAN ID information is removed from the "data" field.
        printf("\n --> Received data:");
        for (uint32_t i = 0; i < rx_msg.msg.msgdata.any->length; i++)
            printf(" 0x%02x", rx_msg.msg.msgdata.any->data[i]);
        printf("\n\n");
        UDS_MsgFree_2013(&rx_msg);
    }
}
UDS_MsgFree_2013(&tx_msg);
UDS_Uninitialize_2013(PCANTP_HANDLE_NONEBUS);

```


4.4 Using Events

Event objects can be used to automatically notify a client on reception of a UDS message. This has following advantages:

- └ The client program does not need to check periodically for received messages any longer
- └ The response time on received messages is reduced

To use events, the client application must call the `UDS_SetValue_2013` function (class-method: `SetValue_2013`) to set the parameter `PUDS_PARAMETER_RECEIVE_EVENT`. This parameter sets the handle for the event object. When receiving a message, the API sets this event to the **Signaled** state.

Another thread must be started in the client application, which waits for the event to be signaled, using one of the Win32 synchronization functions (e.g. `WaitForSingleObject`) without increasing the processor load. After the event is signaled, available messages can be read with the `UDS_Read_2013` function (class method: `Read_2013`), and the UDS messages can be processed.

Remarks

Be careful, it is not recommended to use both event-handler (with a reading thread) and the UDS `WaitFor` functions and methods:

- └ `UDS_WaitForSingleMessage_2013` / `WaitForSingleMessage_2013`
- └ `UDS_WaitForFunctionalResponses_2013` / `WaitForFunctionalResponses_2013`
- └ `UDS_WaitForService_2013` / `WaitForService_2013`
- └ `UDS_WaitForServiceFunctional_2013` / `WaitForServiceFunctional_2013`

Indeed, both mechanisms would read messages at the same time, the result is that one will not receive any (or some) messages. If one of the previous functions is called and a thread is waiting for events to call the read UDS message function, then the user will have to temporarily prevent the thread from reading messages.

Tips for the creation of the event object:

- └ Creation of the event as **auto-reset**
 - Trigger mode **set (default)**: After the first waiting thread has been released, the event object's state changes to non-signaled. Other waiting threads are not released. If no threads are waiting, the event object's state remains signaled
 - Trigger mode **pulse**: After the first waiting thread has been released, the event object's state changes to non-signaled. Other waiting threads are not released. If no threads are waiting, or if no thread can be released immediately, the event object's state is simply set to non-signaled
- └ Creation of the event as **manual-reset**
 - Trigger mode **set (default)**: The state of the event object remains signaled until it is set explicitly to the non-signaled state by the Win32 `ResetEvent` function. Any number of waiting threads, or threads that subsequently begin wait operations, can be released while the object's state remains signaled
 - Trigger mode **pulse**: All waiting threads that can be released immediately are released. The event object's state is then reset to the non-signaled state. If no threads are waiting, or if no thread can be released immediately, the event object's state is simply set to non-signaled

See also: `UDS_SetValue_2013` (class-method: `SetValue_2013`), `UDS_Read_2013` (class-method: `Read_2013`)

5 License Information

The use of this software is subject to the terms of the End User License Agreement of PEAK-System Technik GmbH.

The APIs PCAN-UDS 2.x, PCAN-ISO-TP 3.x, and PCAN-Basic are property of the PEAK-System Technik GmbH and may be used only in connection with a hardware component purchased from PEAK-System or one of its partners. If CAN hardware of third-party suppliers should be compatible to that of PEAK-System, then you are not allowed to use the mentioned APIs with those components.

If a third-party supplier develops software based on the mentioned APIs and problems occur during the use of this software, consult that third-party supplier.