



Nucleus® Connectivity Guide

Release 2013.08

August 2013

**© 2007-2013 Mentor Graphics Corporation
All rights reserved.**

This document contains information that is proprietary to Mentor Graphics Corporation. The original recipient of this document may duplicate this document in whole or in part for internal business purposes only, provided that this entire notice appears in all copies. In duplicating any part of this document, the recipient agrees to make every reasonable effort to prevent the unauthorized use and distribution of the proprietary information.

This document is for information and instruction purposes. Mentor Graphics reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Mentor Graphics to determine whether any changes have been made.

The terms and conditions governing the sale and licensing of Mentor Graphics products are set forth in written agreements between Mentor Graphics and its customers. No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Mentor Graphics whatsoever.

MENTOR GRAPHICS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

MENTOR GRAPHICS SHALL NOT BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF MENTOR GRAPHICS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

U.S. GOVERNMENT LICENSE RIGHTS: The software and documentation were developed entirely at private expense and are commercial computer software and commercial computer software documentation within the meaning of the applicable acquisition regulations. Accordingly, pursuant to FAR 48 CFR 12.212 and DFARS 48 CFR 227.7202, use, duplication and disclosure by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in the license agreement provided with the software, except for provisions which are contrary to applicable mandatory federal laws.

TRADEMARKS: The trademarks, logos and service marks ("Marks") used herein are the property of Mentor Graphics Corporation or other parties. No one is permitted to use these Marks without the prior written consent of Mentor Graphics or the owner of the Mark, as applicable. The use herein of a third-party Mark is not an attempt to indicate Mentor Graphics as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A current list of Mentor Graphics' trademarks may be viewed at: www.mentor.com/trademarks.

Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: www.mentor.com
SupportNet: supportnet.mentor.com/

Send Feedback on Documentation: supportnet.mentor.com/doc_feedback_form

Table of Contents

Chapter 1

Nucleus Connectivity Components	25
Build Configurations	25
Required Include File	25
Connectivity Components	26
Nucleus Connectivity Examples	27

Chapter 2

Controller Area Network (CAN)	29
CAN Module Overview	29
CAN Driver	29
Enabling CAN	30
CAN Function Reference	31
NU_CAN_Start	32
NU_CAN_Assign_RTR_Response	35
NU_CAN_Clear_RTR_Response	37
NU_CAN_Close_Driver	39
NU_CAN_Get_Baud_Rate	41
NU_CAN_Get_Node_State	42
NU_CAN_Receive_Data	44
NU_CAN_Request_Data_Transfer	46
NU_CAN_Request_Remote_Transfer	48
NU_CAN_Set_Acceptance_Mask	50
NU_CAN_Set_Baud_Rate	53
NU_CAN_Sleep	55
NU_CAN_Wakeup	57
CAN Configuration Options	58
Driver Selection (Loopback/Hardware)	58
Remote Transmission Request (RTR)	60
Queue Size	61
Maximum CAN Driver	61
CAN Baud Rate	61
Self Transmitted Frame Handling	61
Debug Mode	62
Optimization	62
Time Stamp	62
Multiple Port	62
MMU Support	63
CAN Error Codes	63

Chapter 3

Inter Integrated Circuit (I2C)	67
I2C Module Overview	67
Standard I2C Model	68
I2C Driver	69
Enabling I2C	69
I2C Function Reference	70
I2C Common API Functions	71
NU_I2C_Close	72
NU_I2C_Get_Node_State	74
NU_I2C_Master_Get_Baudrate	76
NU_I2C_Master_Get_Callbacks	78
NU_I2C_Master_Get_Mode	79
NU_I2C_Master_Set_Baudrate	80
NU_I2C_Master_Set_Callbacks	81
NU_I2C_Master_Remove_Callbacks	83
NU_I2C_Open	84
NU_I2C_Receive_Data	87
NU_I2C_Slave_Get_Address	89
NU_I2C_Slave_Set_Address	91
I2C Automatic API Functions	93
NU_I2C_Master_Config_HW_Master	94
NU_I2C_Master_Multi_Transfer	96
NU_I2C_Master_Read	99
NU_I2C_Master_Set_Slave_Address	101
NU_I2C_Master_Write	103
NU_I2C_Slave_Response_To_Read	105
I2C Fine Control API Functions	107
NU_I2C_Check_Ack	108
NU_I2C_Ioctl_Driver	110
NU_I2C_Master_Free_Bus	112
NU_I2C_Master_Read_Byte	113
NU_I2C_Master_Restart	114
NU_I2C_Master_Start_Transfer	116
NU_I2C_Master_Stop_Transfer	118
NU_I2C_Master_Write_Byte	119
NU_I2C_Send_Ack	120
I2C Configuration Options	121
Acknowledgment Wait	121
Slave Address of the Node	122
I2C Baud Rate	122
I2C Error Checking	122
I2C Device Selection	122
Node Type	122
Support for Polling Mode Selection	123
I2C Error Codes	123

Chapter 4

Serial Peripheral Interface (SPI).....	127
SPI Module Overview	127
Standard SPI Model	128
SPI Driver	129
SPI Configuration Options.....	130
Configuration in SPI Metadata File	130
Configuration in nu_connectivity.h File.....	130
SPI Error Checking.....	131
Support for Polling Mode	131
Support for Internal Buffering	131
Optimize Buffer Copy for Speed	132
Number of SPI Devices	132
SPI Pin Naming	132
SPI Function Reference	133
Initialization and Shutdown	134
NU_SPI_Close	135
NU_SPI_Start	136
Transfer Attribute Control	139
NU_SPI_Master_Get_Configuration	140
NU_SPI_Master_Set_Baud_Rate.....	142
NU_SPI_Master_Set_Bit_Order.....	144
NU_SPI_Master_Set_Configuration	146
NU_SPI_Master_Set_SPI_Mode	148
NU_SPI_Master_Set_Transfer_Size	150
NU_SPI_Slave_Get_Configuration	152
NU_SPI_Slave_Set_Bit_Order.....	153
NU_SPI_Slave_Set_Configuration.....	155
NU_SPI_Slave_Set_SPI_Mode	157
NU_SPI_Slave_Set_Transfer_Size.....	159
Data Transfer	160
NU_SPI_Master_Duplex_8Bit.....	162
NU_SPI_Master_Duplex_16Bit.....	164
NU_SPI_Master_Duplex_32Bit.....	166
NU_SPI_Master_Receive_8Bit	168
NU_SPI_Master_Receive_16Bit	170
NU_SPI_Master_Receive_32Bit	172
NU_SPI_Master_Transmit_8Bit.....	174
NU_SPI_Master_Transmit_16Bit.....	176
NU_SPI_Master_Transmit_32Bit.....	178
NU_SPI_Slave_8Bit_Duplex_Response	180
NU_SPI_Slave_16Bit_Duplex_Response	182
NU_SPI_Slave_32Bit_Duplex_Response	184
NU_SPI_Slave_8Bit_Receive_Response.....	186
NU_SPI_Slave_16Bit_Receive_Response.....	188
NU_SPI_Slave_32Bit_Receive_Response.....	190
NU_SPI_Slave_8Bit_Transmit_Response	192
NU_SPI_Slave_16Bit_Transmit_Response	194

NU_SPI_Slave_32Bit_Transmit_Response	196
Miscellaneous	198
NU_SPI_Ioctl_Driver	199
NU_SPI_Master_Get_Callbacks	201
NU_SPI_Master_Get_Driver_Mode	203
NU_SPI_Master_Set_Callbacks	204
NU_SPI_Master_Set_SS_Polarity	206
SPI Error Codes	207
Lightweight SPI	209
Theory of Operation	210
Lightweight SPI Configuration	210
Lightweight SPI Function Reference	211
NU_SPI_Register_Slave	212
NU_SPI_Unregister_Slave	214
NU_SPI_Read	215
NU_SPI_Write	217
NU_SPI_Write_Read	219
Chapter 5	
Serial Driver	221
Serial Driver Module Overview	221
Driver Functions	222
NU_SD_Data_Ready	223
NU_SD_Put_Char	224
NU_SD_Put_String	226
NU_SD_Put_Stringn	227
I/O Services (SIO) Functions	228
NU_SIO_Get_Port	229
NU_SIO_Getchar	230
NU_SIO_Putchar	231
NU_SIO_Puts	232
Middleware Functions	233
NU_Serial_Close	234
NU_Serial_Getchar	235
NU_Serial_Get_Configuration	236
NU_Serial_Get_Read_Mode	238
NU_Serial_Get_Write_Mode	239
NU_Serial_Open	240
NU_Serial_Putchar	242
NU_Serial_Puts	243
NU_Serial_Set_Configuration	244
NU_Serial_Set_Read_Mode	246
NU_Serial_Set_Write_Mode	247
Nucleus Serial Driver Examples	248
Serial Demo	249

Chapter 6

DMA Device Driver	251
DMA Device Driver Module Overview	251
DMA Device Driver Operation	251
DMA Initialization	251
DMA Channel Acquisition	252
DMA Data Transfer	253
DMA Device Driver Data Structures	255
DMA_ADDRESS_TYPE	255
DMA_REQUEST_TYPE	255
DMA_REQ	256
DMA Device Driver APIs	258
NU_DMA_Open	259
NU_DMA_Close	260
NU_DMA_Acquire_Channel	261
NU_DMA_Release_Channel	263
NU_DMA_Reset_Channel	264
NU_DMA_Data_Transfer	265

Chapter 7

USB Communications	267
USB Communication Class Module Overview	267
USB Host Component	267
Host Component Control Block	268
Host Component Initialization	268
Host Component Hierarchy	268
Host Comm Class Match Spec	269
Communications Class Function Reference	271
NU_USBH_COM_Clear_Comm_Feature	272
NU_USBH_COM_Create	273
NU_USBH_COM_Data_Transfer	275
NU_USBH_COM_Get_Comm_Feature	277
NU_USBH_COM_Get_Encap_Resp	278
NU_USBH_COM_Send_Encap_Cmd	279
NU_USBH_COM_Set_Comm_Feature	280
_NU_USBH_COM_Delete	282
ACM Function Reference	283
NU_USBH_COM_Get_Line_Coding	284
NU_USBH_COM_Send_Break	286
NU_USBH_COM_Set_Ctrl_LS	287
NU_USBH_COM_Set_Line_Coding	288
ECM Function Reference	289
NU_USBH_COM_Get_ETH_Power_Filter	290
NU_USBH_COM_Get_ETH_Static	292
NU_USBH_COM_Set_ETH_Mul_Filter	293
NU_USBH_COM_Set_ETH_Packet_Filter	294
NU_USBH_COM_Set_ETH_Power_Filter	295
Host Communications User Driver	297

Writing a User Driver for NU_USBH_COM	297
USB Host Communications User Base Driver	299
NU_USBH_COM_USER	299
Host Communications User Base Driver Functions	300
_NU_USBH_COM_USER_Create	301
_NU_USBH_COM_USER_Delete	302
NU_USBH_COM_USER_Create_Polling	303
NU_USBH_COM_USER_Delete_Polling	304
NU_USBH_COM_USER_Get_Data	305
NU_USBH_COM_USER_HDL	306
NU_USBH_COM_USER_Register	307
NU_USBH_COM_USER_Send_Data	308
NU_USBH_COM_USER_Start_Polling	309
NU_USBH_COM_USER_Stop_Polling	310
NU_USBH_COM_USER_Wait	311
*Connect_Handler	313
*Data_Handler	314
*Disconnect_Handler	316
*Event_Handler	317
USB Function Component	319
Function Component Control block	320
Function Component Initialization	320
Function Component Hierarchy	320
Function Component Match Spec	320
Class Driver Configuration Parameters	321
NU_USBF_COMM_DATA	322
Comm Class Component Hierarchy	322
Comm Class Match Spec	322
Nucleus USB Function Communications User Base Driver	323
NU_USBF_USER_COMM	323
Communications Interface Function Reference	324
NU_USBF_COMM_Cancel_Io	325
NU_USBF_COMM_Create	326
NU_USBF_COMM_Send_Notification	327
Data Interface Function Reference	329
NU_USBF_COMM_DATA_Cancel_Io	330
NU_USBF_COMM_DATA_Config_Xfers	331
NU_USBF_COMM_DATA_Create	333
NU_USBF_COMM_DATA_Dis_Reception	334
NU_USBF_COMM_DATA_Get_Rcvd	335
NU_USBF_COMM_DATA_Rbg_Create	336
NU_USBF_COMM_DATA_Reg_Rx_Buffer	337
NU_USBF_COMM_DATA_Send	338
Nucleus USB Function Communications User Function Reference	339
_NU_USBF_USER_COMM_Create	340
NU_USBF_USER_COMM_DATA_Connect	341
NU_USBF_USER_COMM_DATA_Discon	342
NU_USBF_USER_COMM_Delete	343
NU_USBF_USER_COMM_Wait	344

Table of Contents

Developing a Communication User Driver	345
Step 1: Creating Control Block	345
Step 2: Defining a new Dispatch Table	345
Step 3: Implementing the User Driver	345
Step 4: Populating the Dispatch Table	346
Step 5: Create Function	346
Step 6: Activating the User Driver	347
Nucleus USB Communications Class Driver Callback Functions	348
_NU_USBF_XYZ_USER_Connect	349
_NU_USBF_XYZ_USER_Delete	350
_NU_USBF_XYZ_USER_Disconnect	351
_NU_USBF_XYZ_USER_New_Command	352
_NU_USBF_XYZ_USER_New_Transfer	354
_NU_USBF_XYZ_USER_Notify	355
_NU_USBF_XYZ_USER_Tx_Done	356
 Chapter 8	
USB Mass Storage.....	359
USB Host Driver	359
NU_USBH_MS	359
Mass Storage Host Match Spec	360
Logical Units	361
Command Block Specification	361
Writing a User Driver for NU_USBH_MS	363
Transport Service	365
USB Host Driver Functions	366
NU_USBH_MS_Create	367
NU_USBH_MS_Transport	369
USB Host FILE Driver	371
Architecture	371
Nucleus FILE 3.1 Using USB Host FILE Driver Subproduct	372
USB Host Driver and the Device Manager	372
USB Function Driver	373
NU_USBF_MS	373
Function Mass Storage Component Hierarchy	373
Mass Storage Function Match Spec	374
USB Function Driver Functions	375
NU_USBF_MS_Create	376
NU_USBF_MS_Create_Task_Mode	377
NU_USBF_MS_Start_Cmd_Processing	378
Function Subclass Drivers	379
USB Function User Driver Functions	380
_NU_USBF_USER_MS_Create	381
_NU_USBF_USER_MS_Delete	382
NU_USBF_USER_MS_Get_Max_LUN	383
NU_USBF_USER_MS_Reset	384
USB Function Device Manager	385
NU_USBF_MS_CALLBACKS	385

Nucleus USB Function User SCSI Driver	389
Function SCSI Component Hierarchy	389
USB Function User SCSI Driver Functions	389
NU_USBF_USER_SCSI_Create	390
NU_USBF_USER_SCSI_Delete	391
NU_USBF_USER_SCSI_Dereg_Media	392
NU_USBF_USER_SCSI_Get_Class_Handl	393
NU_USBF_USER_SCSI_Reg_Media	394
Media Drivers and NU_USBF_USER_SCSI	395
SCSI Container Configuration Parameters	395
Function Media Drivers	395
Developing SCSI Media Drivers	401
Creating Components	402
Registering Media with SCSI Container	402
Providing Custom Response to Commands	403
Reading and Writing to Media	404
USB Function SCSI Media Functions	405
NU_USBF_SCSI_MEDIA_Capacity	407
NU_USBF_SCSI_MEDIA_Connect	408
NU_USBF_SCSI_MEDIA_Disconnect	409
NU_USBF_SCSI_MEDIA_Format	410
NU_USBF_SCSI_MEDIA_Inquiry	412
NU_USBF_SCSI_MEDIA_Mode_Sel6	413
NU_USBF_SCSI_MEDIA_Mode_Sel_10	414
NU_USBF_SCSI_MEDIA_Mode_Sense_6	415
NU_USBF_SCSI_MEDIA_Mode_Sense_10	416
NU_USBF_SCSI_MEDIA_New_Transfer	417
NU_USBF_SCSI_MEDIA_Prevent_Allow	418
NU_USBF_SCSI_MEDIA_Read	419
NU_USBF_SCSI_MEDIA_Read_6	420
NU_USBF_SCSI_MEDIA_Read_10	421
NU_USBF_SCSI_MEDIA_Read_12	422
NU_USBF_SCSI_MEDIA_Ready	423
NU_USBF_SCSI_MEDIA_Release_Unit	424
NU_USBF_SCSI_MEDIA_Reserve_Unit	425
NU_USBF_SCSI_MEDIA_Reset_Device	426
NU_USBF_SCSI_MEDIA_Sense	427
NU_USBF_SCSI_MEDIA_Snd_Diag	428
NU_USBF_SCSI_MEDIA_Start_Stop	429
NU_USBF_SCSI_MEDIA_Tx_Done	430
NU_USBF_SCSI_MEDIA_Unknown_Cmd	432
NU_USBF_SCSI_MEDIA_Verify	433
NU_USBF_SCSI_MEDIA_Write	434
NU_USBF_SCSI_MEDIA_Write_6	435
NU_USBF_SCSI_MEDIA_Write_10	436
NU_USBF_SCSI_MEDIA_Write_12	437
_NU_USBF_SCSI_MEDIA_Create	438
_NU_USBF_SCSI_MEDIA_Delete	439
_NU_USBF_SCSI_MEDIA_Insert	440

Table of Contents

_NU_USBFS SCSI_MEDIA_Remove	441
Chapter 9	
USB Human Interface Design (HID).....	443
HID Overview	443
USB Host Mouse Driver	443
USB Host Mouse Driver and Device Manager.....	444
USB Host Keyboard Driver.....	449
USB Host Keyboard Driver and Device Manager	449
USB Function Mouse Driver.....	454
Function Mouse Driver Configuration Macros.....	455
Function Mouse Data Structures.....	455
Values of Characters and LED States in the Application.....	455
Function Mouse Driver and Device Manager.....	457
USBFS_MSE_WAIT_DATA.....	458
USB Function Keyboard Driver	461
Function Keyboard Configuration Macros	461
Mouse Driver Data Structures	461
Function Keyboard Driver and Device Manager	462
Chapter 10	
USB Audio Driver.....	467
USB Host Audio Driver Overview	467
Nucleus USB Host Audio Driver Operation	468
Configuration	468
Initialization	469
Device Connection and Disconnection Handling.....	470
Nucleus USB Host Audio Driver Application Interface	472
Nucleus USB Host Audio Driver IOCTL Data Structures.....	472
Nucleus USB Host Audio Driver IOCTLs	479
USBH_AUDIO_OPEN_PLAY_SESSION	481
USBH_AUDIO_CLOSE_PLAY_SESSION	482
USBH_AUDIO_OPEN_RECORD_SESSION	483
USBH_AUDIO_CLOSE_RECORD_SESSION	484
USBH_AUDIO_REGISTER_RECORD_CALLBACK	485
USBH_AUDIO_REGISTER_PLAY_CALLBACK	486
USBH_AUDIO_PLAY_SOUND.....	487
USBH_AUDIO_RECORD_SOUND.....	488
USBH_AUDIO_GET_FUNC_COUNT.....	489
USBH_AUDIO_GET_FREQ_COUNT.....	490
USBH_AUDIO_GET_FUNC_SETTINGS	491
USBH_AUDIO_REGISTER_NOTIFICATION_CALLBACKS	492
USBH_AUDIO_USER_GET_VOL_STATS.....	493
USBH_AUDIO_USER_ADJUST_VOL	494
USBH_AUDIO_USER_GET_AUD_FUNCS	495
USBH_AUDIO_USER_GET_SUPPORTED_CTRLs	496
USBH_AUDIO_USER_GET_DEV_CB	497

Chapter 11**USB Device Firmware Upgrade Driver 499**

USB Function DFU Driver Overview	499
USB Function DFU Driver Operation	500
USB DFU Mode Switching Sequence	501
Descriptor Switching User-Configurable Parameters	502
USB Function DFU Driver Build Configuration	503
Run level Initialization	503
USB Function DFU Driver Application Interface	504
USB Function DFU Driver IOCTL Data Structures	505
USB Function DFU Driver IOCTLs	505
USBF_DFU_IOCTL_SET_STATUS	507
USBF_DFU_IOCTL_GET_STATE	509
USBF_DFU_IOCTL_GET_STATUS	511
USBF_DFU_IOCTL_REG_CALLBACK	512
USBF_DFU_IOCTL_UNREG_CALLBACK	513

Chapter 12**USB Host and Functions 515**

Nucleus USB Services	517
NU_USB_Delete	518
NU_USB_Get_Name	520
NU_USB_Get_Object_Id	521
Nucleus USB Alternate Setting Services	522
NU_USB_ALT_SETTG_Find_Pipe	523
NU_USB_ALT_SETTG_Get_bAlternateSetting	525
NU_USB_ALT_SETTG_Get_Class	526
NU_USB_ALT_SETTG_Get_Class_Desc	527
NU_USB_ALT_SETTG_Get_Desc	528
NU_USB_ALT_SETTG_Get_Endp	529
NU_USB_ALT_SETTG_Get_Is_Active	530
NU_USB_ALT_SETTG_Get_Num_Endps	531
NU_USB_ALT_SETTG_Get_Protocol	532
NU_USB_ALT_SETTG_Get_String	533
NU_USB_ALT_SETTG_Get_String_Desc	534
NU_USB_ALT_SETTG_Get_String_Num	535
NU_USB_ALT_SETTG_Get_SubClass	536
NU_USB_ALT_SETTG_Set_Active	537
Nucleus USB Binary Object Store (BOS) Services	538
NU_USB_BOS_Get_Num_DevCap	539
NU_USB_BOS_Get_Total_Length	540
NU_USB_DEVCAP_CntnrID_Get_CID	541
NU_USB_DEVCAP_SuprSpd_Get_FS	542
NU_USB_DEVCAP_SuprSpd_Get_Functionality	543
NU_USB_DEVCAP_SuprSpd_Get_HS	544
NU_USB_DEVCAP_SuprSpd_Get_LS	545
NU_USB_DEVCAP_SuprSpd_Get_LTM	546
NU_USB_DEVCAP_SuprSpd_Get_SS	547

Table of Contents

NU_USB_DEVCAP_SuprSpd_Get_U1ExitLat.	548
NU_USB_DEVCAP_SuprSpd_Get_U2ExitLat.	549
NU_USB_DEVCAP_USB2Ext_Get_LPM.	550
Nucleus USB Config Services.	551
NU_USB_CFG_Find_Alt_Setting.	552
NU_USB_CFG_Get_Cfg_Value.	554
NU_USB_CFG_Get_Desc.	555
NU_USB_CFG_Get_Device.	556
NU_USB_CFG_Get_IAD.	557
NU_USB_CFG_Get_Intf.	558
NU_USB_CFG_Get_Is_Active.	559
NU_USB_CFG_Get_Is_Self_Powered.	560
NU_USB_CFG_Get_Is_Wakeup.	561
NU_USB_CFG_Get_Max_Power.	562
NU_USB_CFG_Get_Num_IADs.	563
NU_USB_CFG_Get_Num_Intfs.	564
NU_USB_CFG_Get_String.	565
NU_USB_CFG_Get_String_Desc.	566
NU_USB_CFG_Get_String_Num.	567
NU_USB_CFG_Get_wTotalLength.	568
NU_USB_CFG_Set_Is_Active.	569
Nucleus USB Device Services.	571
NU_USB_DEVICE_Claim.	574
NU_USB_DEVICE_Get_Active_Cfg.	575
NU_USB_DEVICE_Get_Active_Cfg_Num.	576
NU_USB_DEVICE_Get_bcdDevice.	577
NU_USB_DEVICE_Get_bcdUSB.	578
NU_USB_DEVICE_Get_bDeviceClass.	579
NU_USB_DEVICE_Get_bDeviceProtocol.	580
NU_USB_DEVICE_Get_bDeviceSubClass.	581
NU_USB_DEVICE_Get_bMaxPacketSize0.	582
NU_USB_DEVICE_Get_BOS.	583
NU_USB_DEVICE_Get_BOS_Desc.	584
NU_USB_DEVICE_Get_Cfg.	585
NU_USB_DEVICE_Get_CntnrID_Desc.	586
NU_USB_DEVICE_Get_Current_Requirement.	587
NU_USB_DEVICE_Get_Desc.	588
NU_USB_DEVICE_Get_Function_Addr.	589
NU_USB_DEVICE_Get_Hw.	590
NU_USB_DEVICE_Get_idProduct.	591
NU_USB_DEVICE_Get_idVendor.	592
NU_USB_DEVICE_Get_Is_Claimed.	593
NU_USB_DEVICE_Get_Manf_String.	594
NU_USB_DEVICE_Get_Manf_String_Desc.	595
NU_USB_DEVICE_Get_Manf_String_Num.	596
NU_USB_DEVICE_Get_Num_Cfgs.	597
NU_USB_DEVICE_Get_OTG_Desc.	598
NU_USB_DEVICE_Get_OTG_Status.	599
NU_USB_DEVICE_Get_Parent.	600

NU_USB_DEVICE_Get_Port_Number	601
NU_USB_DEVICE_Get_Product_String	602
NU_USB_DEVICE_Get_Product_String_Desc	603
NU_USB_DEVICE_Get_Product_String_Num	604
NU_USB_DEVICE_Get_Serial_Num_String	605
NU_USB_DEVICE_Get_Serial_Num_String_Desc	606
NU_USB_DEVICE_Get_Serial_Num_String_Num	607
NU_USB_DEVICE_Get_Speed	608
NU_USB_DEVICE_Get_Stack	609
NU_USB_DEVICE_Get_Status	610
NU_USB_DEVICE_Get_String	611
NU_USB_DEVICE_Get_String_Desc	612
NU_USB_DEVICE_Get_SuprSpd_Desc	613
NU_USB_DEVICE_Get_USB2Ext_Desc	614
NU_USB_DEVICE_Release	615
NU_USB_DEVICE_Set_Active_Cfg	616
NU_USB_DEVICE_Set_bcdUSB	618
NU_USB_DEVICE_Set_bDeviceClass	619
NU_USB_DEVICE_Set_bMaxPacketSize0	620
NU_USB_DEVICE_Set_Desc	621
NU_USB_DEVICE_Set_Device_Qualifier	622
NU_USB_DEVICE_Set_Hw	623
NU_USB_DEVICE_Set_Link_State	624
NU_USB_DEVICE_Set_Manf_String	625
NU_USB_DEVICE_Set_Product_String	626
NU_USB_DEVICE_Set_Stack	627
NU_USB_DEVICE_Set_Status	628
NU_USB_DEVICE_Set_String	629
NU_USB_DEVICE_Set_Serial_Num_String	630
Nucleus USB Driver Services	631
NU_USB_DRV_R_Deregister_User	632
NU_USB_DRV_R_Get_Users	633
NU_USB_DRV_R_Get_Users_Count	634
NU_USB_DRV_R_Register_User	635
Nucleus USB ENDPoint Services	636
NU_USB_ENDP_Get_Alt_Settg	637
NU_USB_ENDP_Get_Bulk_MaxStreams	638
NU_USB_ENDP_Get_BytesPerInterval	639
NU_USB_ENDP_Get_Class_Desc	640
NU_USB_ENDP_Get_Companion_Desc	641
NU_USB_ENDP_Get_Desc	642
NU_USB_ENDP_Get_Device	643
NU_USB_ENDP_Get_Direction	644
NU_USB_ENDP_Get_Interval	645
NU_USB_ENDP_Get_Iso_MaxPktPerIntrvl	646
NU_USB_ENDP_Get_MaxBurst	647
NU_USB_ENDP_Get_Max_Packet_Size	648
NU_USB_ENDP_Get_Number	649
NU_USB_ENDP_Get_Num_Transactions	650

Table of Contents

NU_USB_ENDP_Get_Pipe	651
NU_USB_ENDP_Get_SSEPC_bmAttributes	652
NU_USB_ENDP_Get_Status	653
NU_USB_ENDP_Get_Sync_Type	654
NU_USB_ENDP_Get_Transfer_Type	655
NU_USB_ENDP_Get_Usage_Type	656
Nucleus USB Interface Association Descriptor (IAD) Services	657
NU_USB_IAD_Check_Interface	658
NU_USB_IAD_Get_Desc	659
NU_USB_IAD_Get_First_Interface	660
NU_USB_IAD_Get_Last_Interface	661
Nucleus USB Interface Services	662
NU_USB_INTF_Claim	663
NU_USB_INTF_Find_Alt_Setting	664
NU_USB_INTF_Get_Active_Alt_Setting	666
NU_USB_INTF_Get_Active_Alt_Setting_Num	667
NU_USB_INTF_Get_Alt_Setting	668
NU_USB_INTF_Get_Cfg	669
NU_USB_INTF_Get_Class	670
NU_USB_INTF_Get_Desc	671
NU_USB_INTF_Get_Device	672
NU_USB_INTF_Get_IAD	673
NU_USB_INTF_Get_Intf_Num	674
NU_USB_INTF_Get_Is_Claimed	675
NU_USB_INTF_Get_Num_Alt_Settings	676
NU_USB_INTF_Get_Protocol	677
NU_USB_INTF_Get_String	678
NU_USB_INTF_Get_String_Desc	679
NU_USB_INTF_Get_String_Num	680
NU_USB_INTF_Get_SubClass	681
NU_USB_INTF_Release	682
NU_USB_INTF_Set_Interface	683
Nucleus USB IRP Services	685
NU_USB_IRP_Create	686
NU_USB_IRP_Delete	688
NU_USB_IRP_Get_Accept_Short_Packets	689
NU_USB_IRP_Get_Actual_Length	690
NU_USB_IRP_Get_Buffer_Type_Cachable	691
NU_USB_IRP_Get_Callback	692
NU_USB_IRP_Get_Context	693
NU_USB_IRP_Get_Data	694
NU_USB_IRP_Get_Interval	695
NU_USB_IRP_Get_Length	696
NU_USB_IRP_Get_Pipe	697
NU_USB_IRP_Get_Status	698
NU_USB_IRP_Get_Use_Empty_Pkt	699
NU_USB_IRP_Set_Accept_Short_Pkt	700
NU_USB_IRP_Set_Actual_Length	701
NU_USB_IRP_Set_Buffer_Type_Cachable	702

NU_USB_IRP_Set_Callback	703
NU_USB_IRP_Set_Context	704
NU_USB_IRP_Set_Data	705
NU_USB_IRP_Set_Interval	706
NU_USB_IRP_Set_Length	707
NU_USB_IRP_Set_Pipe	708
NU_USB_IRP_Set_Status	709
NU_USB_IRP_Set_Use_Empty_Pkt	711
USB Memory Component	712
USB_Allocate_Aligned_Memory	713
USB_Allocate_Memory	714
USB_Allocate_Object	715
USB_Deallocate_Memory	716
Nucleus USB Pipe Services	717
NU_USB_PIPE_Flush	718
NU_USB_PIPE_Get_Device	719
NU_USB_PIPE_Get_Endp	720
NU_USB_PIPE_Get_Is_Active	721
NU_USB_PIPE_Get_Is_Stalled	722
NU_USB_PIPE_Set_Device	723
NU_USB_PIPE_Set_Endp	724
NU_USB_PIPE_Set_Is_Active	725
NU_USB_PIPE_Stall	726
NU_USB_PIPE_Submit_IRP	727
NU_USB_PIPE_Uninstall	728
Nucleus USB Power Management Services	729
NU_USB_PMG_Get_Dev_Pwr_Attrib	730
NU_USB_PMG_Update_Dev_LTM_Enable	731
NU_USB_PMG_Update_Dev_Pwr_Src	732
NU_USB_PMG_Update_Dev_U1_Enable	733
NU_USB_PMG_Update_Dev_U2_Enable	734
NU_USB_PMG_Update_Intf_Pwr_Attrib	735
NU_USB_PMG_Set_Dev_Pwr_Attrib	736
NU_USB_PMG_Set_Link_State	737
Nucleus USB Stack Services	738
NU_USB_STACK_Add_Hw	739
NU_USB_STACK_Deregister_Drvr	740
NU_USB_STACK_Function_Suspend	741
NU_USB_STACK_Register_Drvr	742
NU_USB_STACK_Remove_Hw	743
Nucleus USB Bulk Streaming Services	744
NU_USB_STRM_Create	745
NU_USB_STRM_Get_State	746
NU_USB_STRM_Get_Stream_ID	747
NU_USB_STRM_GRP_Acquire_Arbitrate	748
NU_USB_STRM_GRP_Acquire_Forceful	749
NU_USB_STRM_GRP_Create	750
NU_USB_STRM_GRP_Delete	751
NU_USB_STRM_GRP_Release_Strm	752

Table of Contents

NU_USB_STRM_Set_State	753
NU_USB_STRM_Set_Stream_ID	754
Nucleus USB User Services	755
NU_USB_USER_Connect	756
NU_USB_USER_Disconnect	757
NU_USB_USER_Get_Protocol	758
NU_USB_USER_Get_Subclass	759
Nucleus USB Host Services	760
NU_USBH_Create	761
NU_USBH_Delete	762
NU_USBH_Validate_Drvr_Object	763
NU_USBH_Validate_HW_Object	764
NU_USBH_Validate_Stack_Object	765
NU_USBH_Validate_User_Object	766
Nucleus USB Host Control IRP Services	767
NU_USBH_CTRL_IRP_Create	768
NU_USBH_CTRL_IRP_Get_bRequest	770
NU_USBH_CTRL_IRP_Get_bmRequestType	771
NU_USBH_CTRL_IRP_Get_Direction	772
NU_USBH_CTRL_IRP_Get_Setup_Pkt	773
NU_USBH_CTRL_IRP_Get_wIndex	774
NU_USBH_CTRL_IRP_Get_wLength	775
NU_USBH_CTRL_IRP_Get_wValue	776
NU_USBH_CTRL_IRP_Set_bRequest	777
NU_USBH_CTRL_IRP_Set_bmRequestType	778
NU_USBH_CTRL_IRP_Set_wIndex	779
NU_USBH_CTRL_IRP_Set_wLength	780
NU_USBH_CTRL_IRP_Set_wValue	781
Nucleus USB Host Class Driver Services	782
_NU_USBH_DRVVR_Create	783
_NU_USBH_DRVVR_Delete	786
Nucleus USB Host Hardware Driver Services	787
_NU_USBH_HW_Create	788
_NU_USBH_HW_Delete	790
Nucleus USB Host STACK Services	791
NU_USBH_STACK_Create	792
NU_USBH_STACK_Get_Config_Info	794
NU_USBH_STACK_Get_Devices	795
NU_USBH_STACK_LTM_Disable	796
NU_USBH_STACK_LTM_Enable	797
NU_USBH_STACK_Resume_Device	798
NU_USBH_STACK_Suspend_Device	799
NU_USBH_STACK_Switch_Config	800
NU_USBH_STACK_U1_Disable	801
NU_USBH_STACK_U1_Enable	802
NU_USBH_STACK_U2_Disable	803
NU_USBH_STACK_U2_Enable	804
Nucleus USB Host User Driver Services	805
NU_USBH_USER_Close_Device	806

NU_USBH_USER_Get_Drvr.	807
NU_USBH_USER_Open_Device	808
NU_USBH_USER_Remove_Device	809
NU_USBH_USER_Wait	810
_NU_USBH_USER_Create	811
_NU_USBH_USER_Delete	812
Nucleus USB Function Class Driver Services.	813
_NU_USBH_DRVR_Create	814
_NU_USBH_DRVR_Delete	817
USB Function Device Configuration Services	818
USBF_DEVCFG_Activate_Device	819
USBF_DEVCFG_Add_Config_String.	820
USBF_DEVCFG_Add_Function	821
USBF_DEVCFG_Add_Intf_String	823
USBF_DEVCFG_Bind_Function	824
USBF_DEVCFG_Create_Config.	825
USBF_DEVCFG_Deactivate_Device	826
USBF_DEVCFG_Delete_Config.	827
USBF_DEVCFG_Delete_Function	828
USBF_DEVCFG_Disable_Function	829
USBF_DEVCFG_Enable_Function.	830
USBF_DEVCFG_Unbind_Function	831
Nucleus USB Function Hardware Driver Services	832
_NU_USBH_HW_Create	833
_NU_USBH_HW_Delete	835
Nucleus USB Function STACK Services	836
NU_USBH_STACK_Attach_Device	837
NU_USBH_STACK_Create	838
NU_USBH_STACK_Detach_Device.	839
NU_USBH_STACK_New_Transfer.	840
NU_USBH_STACK_Notify	841
NU_USBH_STACK_Speed_Change	842
Nucleus USB Function Services	844
NU_USBH_Create	845
NU_USBH_Delete	846
NU_USBH_Validate_Drvr_Object.	847
NU_USBH_Validate_Hw_Object.	848
NU_USBH_Validate_Stack_Object	849
NU_USBH_Validate_User_Object.	850
Nucleus USB Function User Driver Services	851
NU_USBH_USER_New_Command	852
NU_USBH_USER_New_Transfer	854
NU_USBH_USER_Notify	855
NU_USBH_USER_Transfer_Complete	856
_NU_USBH_USER_Create	858
_NU_USBH_USER_Delete	859
Dispatch Table Reference	860
NU_USB.	861
Delete	862

Table of Contents

Get_Name	863
Get_Object_Id.	864
NU_USB_DRVR	865
Initialize_Device	866
Initialize_Interface	867
Examine_Device	868
Examine_Intf.	869
Get_Score	870
NU_USB_USER.	872
Connect	873
Disconnect.	874
NU_USBF_USER	875
New_Command	876
New_Transfer	878
Notify	879
Transfer_Complete	880
Nucleus USB Function Stack Initialization	882
USB Function Stack Interface with Hardware Driver	882
Nucleus USB Host Stack Initialization	883
Nucleus USB Host Stack Interface with Hardware driver	883
Nucleus USB Function and Host IOCTLs.	884
NU_USB_IOCTL_INITIALIZE	886
NU_USB_IOCTL_IO_REQUEST	887
NU_USB_IOCTL_OPEN_PIPE.	888
NU_USB_IOCTL_CLOSE_PIPE	889
NU_USB_IOCTL_MODIFY_PIPE	890
NU_USB_IOCTL_FLUSH_PIPE	891
NU_USB_IOCTL_EXECUTE_ISR.	892
NU_USB_IOCTL_ENABLE_INT.	893
NU_USB_IOCTL_DISABLE_INT	894
NU_USB_IOCTL_GET_ROLE.	895
NU_USB_IOCTL_START_SESSION	896
NU_USB_IOCTL_END_SESSION.	897
NU_USB_IOCTL_NOTIF_ROLE_SWITCH	898
NU_USB_IOCTL_GET_SPEED	899
NU_USB_IOCTL_GET_HW_CB	900
NU_USB_IOCTL_IS_CURR_AVAILABLE	901
NU_USB_IOCTL_RELEASE_POWER	902
NU_USB_IOCTL_REQ_POWER_DOWN.	903
NU_USB_IOCTL_OPEN_SS_PIPE	904
NU_USB_IOCTL_MODIFY_SS_PIPE.	905
NU_USB_IOCTL_UPDATE_PWR_MODE.	906
NU_USB_IOCTL_UPDATE_BELT_VAL	907
NU_USBF_IOCTL_GET_CAPABILITY	908
NU_USBF_IOCTL_SET_ADDRESS	909
NU_USBF_IOCTL_GET_DEV_STATUS	910
NU_USBF_IOCTL_GET_ENDP_STATUS	911
NU_USBF_IOCTL_STALL_ENDP	912
NU_USBF_IOCTL_UNSTALL_ENDP	913

NU_USBH_IOCTL_START_HNP	914
NU_USBH_IOCTL_ACQUIRE_ENDP.....	915
NU_USBH_IOCTL_RELEASE_ENDP.....	916
NU_USBH_IOCTL_ENABLE_PULLUP	917
NU_USBH_IOCTL_DISABLE_PULLUP.....	918
NU_USBH_IOCTL_GET_EP0_MAXP.....	919
NU_USBH_IOCTL_SEND_FUNCWAKENOTIF	920
NU_USBH_IOCTL_SET_FEATURE_U0/U1_ENABLE.....	921
NU_USBH_IOCTL_SET_LTM_ENABLE	922
NU_USBH_IOCTL_CHECK_LTM_CAPABLE.....	923
NU_USBH_IOCTL_GET_SUPPORTED_SPEEDS	924
NU_USBH_IOCTL_GET_U1/U2DEVEXITLAT.....	925
NU_USBH_IOCTL_UPDATE_DEVICE	926
NU_USBH_IOCTL_INIT_DEVICE	927
NU_USBH_IOCTL_DEINIT_DEVICE	928
NU_USBH_IOCTL_UNSTALL_PIPE	929
NU_USBH_IOCTL_DISABLE_PIPE.....	930
NU_USBH_IOCTL_RESET_BANDWIDTH.....	931

Embedded Software and Hardware License Agreement

List of Figures

Figure 2-1. Nucleus CAN Driver.	30
Figure 3-1. Nucleus I2C Based System.	67
Figure 3-2. Interaction of Nucleus I2C with Multiple I2C Interfaces.	68
Figure 3-3. A Typical I2C Transfer.	69
Figure 4-1. Nucleus SPI Based System.	127
Figure 4-2. Interaction of Nucleus SPI with Multiple SPI Interfaces.	128
Figure 4-3. SPI Devices Interconnection.	129
Figure 4-4. Nucleus Lightweight SPI.	209
Figure 5-1. Serial Demo Output.	250
Figure 6-1. DMA Initialization Process Flow.	252
Figure 6-2. DMA Channel Acquisition Process Flow.	253
Figure 6-3. DMA Data Transfer Process Flow.	254
Figure 7-1. Host Component Hierarchy.	268
Figure 7-2. Host Communications User Driver Component Hierarchy.	300
Figure 7-3. Comm Class Driver Function Component Hierarchy.	320
Figure 7-4. Function Communications Data Interface Driver Component Hierarchy.	322
Figure 8-1. Host Mass Storage Class Driver Component Hierarchy.	360
Figure 8-2. USB Host Driver Diagram.	362
Figure 8-3. Communication Between Host Mass Storage Class Driver and FILE.	371
Figure 8-4. USB Function Mass Storage Class Driver Component Hierarchy.	373
Figure 8-5. USB Function SCSI User Driver.	389
Figure 9-1. USB Host Mouse User Driver Component Hierarchy.	443
Figure 9-2. USB Host Keyboard User Driver Component Hierarchy.	449
Figure 10-1. USB Host Audio Driver Block Diagram.	468
Figure 10-2. Nucleus USB Host Audio Driver Initialization Sequence.	469
Figure 10-3. Nucleus USB Audio Device Connect/Disconnect Process Flow.	471
Figure 11-1. USB Function DFU Driver Block Diagram.	500
Figure 11-2. USB Function DFU Driver Descriptor Switching.	501
Figure 11-3. USB Function DFU Driver Initialization Sequence.	504
Figure 12-1. NU_USB_DEVICE_Get_Status Bit Description.	610
Figure 12-2. NU_USB_ENDP_Get_Status Bit Description.	653

List of Tables

Table 1-1. Nucleus Connectivity Components	26
Table 2-1. Nucleus CAN Configuration	58
Table 2-2. Nucleus CAN Error Codes	63
Table 3-1. Nucleus I2C Configuration Options	121
Table 3-2. Nucleus I2C Error Codes	123
Table 4-1. Nucleus SPI Metadata Configuration	130
Table 4-2. Nucleus SPI Configuration	130
Table 4-3. Nucleus SPI Error Codes	207
Table 5-1. Serial Driver Functions	222
Table 5-2. Serial Driver SIO Functions	228
Table 5-3. Serial Driver Middleware Functions	233
Table 5-4. Serial Demo Services	249
Table 6-1. DMA_ADDRESS_TYPE	255
Table 6-2. DMA_REQUEST_TYPE	256
Table 6-3. DMA_REQ	256
Table 6-4. DMA Device Driver APIs	258
Table 7-1. NU_USBH_COM_XBLOCK	275
Table 7-2. NU_USBH_COM_USER_HDL	306
Table 7-3. NU_USBH_COM_XBLOCK	314
Table 7-4. Class Driver Configuration Parameters	321
Table 8-1. USB Host Driver Command Block Specification	361
Table 8-2. USB Storage Device Default Options	372
Table 8-3. USB_STORE_IOCTL_BASE IOCTLS	385
Table 8-4. NU_USBFS_MS_CALLBACKS Descriptions	386
Table 8-5. USBF_SCSI_MEDIA_SET_LUN_SIZE	386
Table 8-6. USBF_SCSI_MEDIA_SET_BLOCK_SIZE	386
Table 8-7. USBF_SCSI_SET_INQUIRY_DATA Descriptions	386
Table 8-8. USBF_SCSI_SET_CAPACITY_DATA Descriptions	387
Table 8-9. USBF_SCSI_INSERT_MEDIA Descriptions	387
Table 8-10. USBF_SCSI_REMOVE_MEDIA Descriptions	388
Table 8-11. Device Configuration Options	388
Table 8-12. SCSI Container Configuration Parameters	395
Table 8-13. SCSI Command Mapping	399
Table 8-14. SCSI Command Implementation	400
Table 9-1. Details of the HID_MSE_IOCTL_DATA Structure	444
Table 9-2. Details of the USBH_MSE_Get_USAGES IOCTL	444
Table 9-3. Details of USBH_MSE_REG_EVENT_HANDLER	445
Table 9-4. Mouse Event Handler Properties	445
Table 9-5. NU_USBH_MOUSE_EVENT Elements	447
Table 9-6. Details of USBH_MSE_REG_STATE_HANDLER	447

List of Tables

Table 9-7. Callback Functions Properties	448
Table 9-8. Details of USBH_MSE_DELETE	448
Table 9-9. Details of USBH_MSE_GET_HANDLE	448
Table 9-10. Details of USBH_MSE_GET_INFO	448
Table 9-11. Details of the HID_KBD_IOCTL_DATA Structure	450
Table 9-12. Details of USBH_USBH_KBD_Get_USAGES	450
Table 9-13. NU_USBH_HID_USAGE Elements	451
Table 9-14. Details of USBH_KBD_REG_EVENT_HANDLER	451
Table 9-15. Keyboard Event Handler Properties	452
Table 9-16. NU_USBH_KBD_EVENT Elements	452
Table 9-17. Details of Modifier Keys	453
Table 9-18. Details of USBH_KBD_REG_STATE_HANDLER	453
Table 9-19. Callback Function Properties	454
Table 9-20. Details of USBH_KBD_DELETE	454
Table 9-21. Details of USBH_KBD_GET_HANDLE	454
Table 9-22. Configuration Macros	455
Table 9-23. Details of NU_USBH_MSE_IOCTL_WAIT	457
Table 9-24. Details of USBH_MSE_WAIT_DATA Elements	458
Table 9-25. Details of NU_USBH_MSE_IOCTL_SEND_LFT_BTN_CLICK	458
Table 9-26. Details of NU_USBH_MSE_IOCTL_SEND_RHT_BTN_CLICK	459
Table 9-27. Details of NU_USBH_MSE_IOCTL_SEND_MDL_BTN_CLICK	459
Table 9-28. Details of NU_USBH_MSE_IOCTL_MOVE_POINTER	459
Table 9-29. NU_USBH_MSE_REPORT Element Details	460
Table 9-30. Details of NU_USBH_MSE_IOCTL_GET_HID_CB	460
Table 9-31. Details of NU_USBH_MSE_IOCTL_IS_HID_DEV_CONNECTED	460
Table 9-32. Nucleus USB Function User Driver Configuration Macros	461
Table 9-33. NU_USBH_KB_IN_REPORT	461
Table 9-34. NU_USBH_KB_OUT_REPORT	462
Table 9-35. Details of NU_USBH_KBD_IOCTL_WAIT	462
Table 9-36. USBH_KBD_WAIT_DATA Element Details	463
Table 9-37. Details of NU_USBH_KBD_IOCTL_REG_CALLBACK	463
Table 9-38. KEYBOARD_RX_CALLBACK Properties	463
Table 9-40. USBH_KBD_SEND_KEY_DATA Elements	464
Table 9-39. Details of NU_USBH_KBD_IOCTL_SEND_KEY_EVENT	464
Table 9-41. Details of NU_USBH_KBD_IOCTL_GET_HID_CB	465
Table 9-42. Details of NU_USBH_KBD_IOCTL_IS_HID_DEV_CONNECTED	465
Table 10-1. NU_AUDIO_DEVICE_SETTINGS	473
Table 10-2. NU_USBH_AUD_Data_Callback	474
Table 10-3. NU_AUDIO_OP_PARAM	475
Table 10-4. NU_USBH_AUDIO_NOTIFY_CALLBACKS	476
Table 10-5. NU_AUDIO_USER_DEVICE_SETTINGS	476
Table 10-6. NU_USBH_AUD_USER_DEV	477
Table 10-7. NU_USBH_AUD_USR_FUNCTIONS	478
Table 11-1. USBH_DFU_APP_CALLBACK	505
Table 11-2. USBH_DFU_IOCTL_SET_STATUS Lookup Table	507

Table 11-3. USBF_DFU_IOCTL_GET_STATE Lookup Table	509
Table 11-4. Callbacks To Be Registered To The DFU Driver	512
Table 12-1. Configurable Parameters Read from Registry	882

Chapter 1

Nucleus Connectivity Components

The Nucleus connectivity components manage the connectivity for a Nucleus application. These components make up the connectivity package located at `\os\connectivity`. Depending on your embedded application, any of these components can be configured for use.

Note



Your source code is initially located in the `<install_root>\nucleus` directory. The source from this directory is copied into the project folder you specify.

[Table 1-1](#) on page 26 summarizes each connectivity component and links to detailed information in this manual. For more information about packages and components, see “Nucleus ReadyStart Configuration” in the *Nucleus ReadyStart Guide* or “Nucleus Source Code Configuration” in the *Nucleus Source Code Guide*.

Build Configurations

All components are by default enabled through the `.metadata` file located at the top level of each component’s installation, for example for the SPI: `\os\connectivity\spi`. When a component is enabled, it is included in the build. You can create a user configuration file to override the default configuration and exclude a component from the build.


For example, if you have two different applications of Nucleus, one requiring CAN, I2C, and HID, and a second one requiring CAN, Serial Driver, and USB Mass Storage, you can create two configuration files and use one for building each application.

For more information, see “Creating a Custom Configuration” in the *Nucleus ReadyStart Guide* or in the *Nucleus Source Code Guide*.

Required Include File

To make all the Connectivity Package Components APIs visible to an application, include the file `connectivity\nu_connectivity.h` in your application using the following statement:

```
#include "connectivity/nu_connectivity.h"
```

 **Warning** In your applications, use only interfaces, structures, macros, and so on, that are documented within this and other Nucleus guides. There is no guarantee of future support or compatibility for any interface that is not documented.

Connectivity Components

Table 1-1 summarizes each Nucleus connectivity component and links to additional detailed usage information in this document.

Table 1-1. Nucleus Connectivity Components

Connectivity Component	Description	Previously Documented in
Controller Area Network (CAN)	Nucleus CAN is a C library implementation of Bosch CAN specification 2.0B and ISO11898-I.	Nucleus CAN Reference Manual
Inter Integrated Circuit (I2C)	Nucleus I2C is an embedded implementation of the I2C-BUS Specification V2.1 that provides an automated and fine control APIs.	Nucleus I2C Reference Manual
Serial Peripheral Interface (SPI)	Nucleus SPI is an implementation of the SPI protocol designed for an embedded application.	Nucleus SPI Reference Manual
Serial Driver	Nucleus serial driver is a serial bus implementation.	Nucleus Serial Driver Reference Manual
USB Communications	Nucleus USB communications class software module is an implementation of the middle ware necessary to develop a USB communication driver or a communications application.	Nucleus USB Communications Class Driver Reference Manual
USB Mass Storage	The Nucleus USB Mass Storage software module enables the Nucleus USB Host to access mass storage devices, which comply with the <i>USB Mass Storage Class Specification Overview</i> .	Nucleus USB Mass Storage Class Driver User's Guide and Reference Manual
USB Human Interface Design (HID)	Nucleus USB HID software module consists of the Nucleus USB Host, Nucleus USB Function, and keyboard and mouse user drivers.	Nucleus USB HID Class Driver Reference Manual
USB Host and Functions	Nucleus USB host and functions software module provides middle ware to interface with the Nucleus host.	Nucleus USB Reference Manual

Note



The documents have been reorganized in the 2012.3 release. The manuals from prior releases have been grouped by category and relocated to a new document for that category to reduce the total number of documents and to increase navigability between them. The *Nucleus Connectivity Guide* is a new document that consists of reference manuals from prior releases that contain information pertaining to the connectivity components. The third column of [Table 1-1](#) maps earlier reference manuals to the chapters in the *Nucleus Connectivity Guide*.

Nucleus Connectivity Examples

This guide offers information about a working [Serial Demo](#) project that is included in your installation. This example can be used as a guide when creating your own project.

- [Nucleus Serial Driver Examples](#)

Chapter 2

Controller Area Network (CAN)

This chapter describes the Nucleus CAN software module and the requirements for using it.

Caution



To guarantee future support and compatibility for your application, use only documented Nucleus interfaces, structures, macros, and so on.

CAN Module Overview

Nucleus CAN is a C library implementation of Bosch CAN specification 2.0B and ISO11898-I. Higher layer protocols or applications implement the callback routines provided by Nucleus CAN to interface with Nucleus CAN.

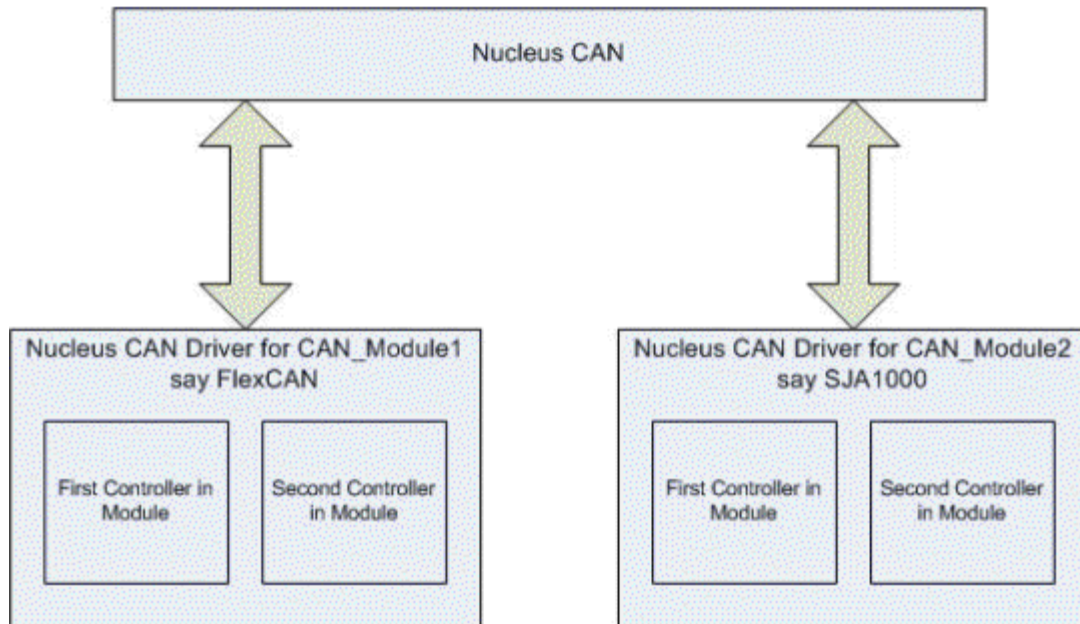
The services provided by Nucleus CAN make it easy to develop and maintain an application across various operating systems provided the support for a particular operating system is added in Nucleus CAN. Nucleus CAN is supported by the Nucleus PLUS RTOS kernel.

Nucleus CAN is designed for use in loopback or with a hardware driver. For more information, see [“Driver Selection \(Loopback/Hardware\)”](#) on page 58.

CAN Driver

Nucleus CAN driver shown in [Figure 2-1](#) is an add-on that drives a specific controller for interfacing with Nucleus CAN. By default, Nucleus CAN runs in loopback mode, but it may be made to use a hardware driver if a certain target supports a CAN controller. Also Nucleus CAN supports multiple ports simultaneously such as the integration of up to four CAN hardware driver ports. Each CAN hardware driver port may support more than one controller. A CAN driver port for a particular module is responsible for handling all the controllers in that module.

Figure 2-1. Nucleus CAN Driver



Enabling CAN

To enable CAN in an application, include the following header in your application code:

```
<install_root>/nucleus/os/include/connectivity/nu_connectivity.h.
```

CAN Function Reference

This section describes the functions supported by Nucleus CAN.

- [NU_CAN_Start](#)
- [NU_CAN_Assign_RTR_Response](#)
- [NU_CAN_Clear_RTR_Response](#)
- [NU_CAN_Close_Driver](#)
- [NU_CAN_Get_Baud_Rate](#)
- [NU_CAN_Get_Node_State](#)
- [NU_CAN_Receive_Data](#)
- [NU_CAN_Request_Data_Transfer](#)
- [NU_CAN_Request_Remote_Transfer](#)
- [NU_CAN_Set_Acceptance_Mask](#)
- [NU_CAN_Set_Baud_Rate](#)
- [NU_CAN_Sleep](#)
- [NU_CAN_Wakeup](#)

NU_CAN_Start

This API initializes Nucleus CAN and must be called before using any service of Nucleus CAN, otherwise undesirable results may occur.

Usage

```
STATUS NU_CAN_Start (CAN_HANDLE *can_dev,  
                    CAN_INIT   *can_init)
```

Arguments

- **can_dev**
Pointer to the location where a handle to the initialized device will be returned. This handle must be used for further interaction with the device.
- **can_init**
Pointer to the initialization structure for Nucleus CAN.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **CAN_OS_ERROR**
Error in the underlying operating system of Nucleus CAN.
- **CAN_UNSUPPORTED_PORT**
The specified port is not supported. Either it does not exist or it is not integrated properly with Nucleus CAN.
- **CAN_UNSUPPORTED_CONTROLLER**
The specified CAN device is not supported.
- **CAN_DEV_ALREADY_INIT**
The specified CAN device/controller is already initialized.
- **CAN_INVALID_PARAMETER**
Invalid parameter value, possibly null passed to the API.
- **CAN_INVALID_BAUDRATE**
The specified baud rate is out of the specified range of 0-1000kbps or is not supported by the driver.
- **CAN_NULL_GIVEN_FOR_MEM_POOL**
Memory pool pointer if pointing to null.

Description

This service sets up the data structures for the protocol stack and initializes either the underlying hardware driver or the CAN loopback device depending upon the configuration chosen by the

user. Message I/O queues, user callback function pointers, driver services pointers, baud rate, IRQs and message buffers setup is performed at this stage. For more information regarding the customization/configuration of Nucleus CAN initialization, refer to the [“CAN Configuration Options”](#) on page 58.

Example 1

```
/* This example demonstrates the initialization of Nucleus CAN without
   support for multiple ports. */

/* Declare the required variables. */
CAN_INIT can_init;
STATUS   status;

/* Declare the variable to get the handle of the initialized CAN device.*/
CAN_HANDLE can_dev;

/* Configure the CAN driver initialization structure. */
can_init.can_memory_pool = uncached_memory;
can_init.can_baud = 1000; /* Set baud rate to 1000kbps. */
can_init.can_controller_label = {CAN_DEVICE_LABEL};

can_init.can_callbacks.can_data_indication = CAN_Data_Indication;
can_init.can_callbacks.can_rtr_indication  = CAN_Remote_Indication;
can_init.can_callbacks.can_data_confirm    = CAN_Data_Confirm;
can_init.can_callbacks.can_rtr_confirm     = CAN_Remote_Confirm;
can_init.can_callbacks.can_error           = CAN_Error_Handler;

/* Call the CAN driver initialization function. */
status = NU_CAN_Start(&can_dev, &can_init);

/* At this point status indicates if CAN initialization was successful. */
```

Example 2

```
/* This example demonstrates the initialization of multiple CAN driver
   ports integrated with Nucleus CAN. */

/* Declare the required variables. */
CAN_INIT can_init;
STATUS   status;

/* Declare the variable to get the handle of the initialized CAN device.*/
CAN_HANDLE can_dev_one;
CAN_HANDLE can_dev_two;

/* Configure the CAN driver initialization structure for initializing
   first CAN device of first CAN Driver Port. */
can_init.can_memory_pool    = uncached_memory;
can_init.can_baud           = 1000;
can_init.can_port_id        = CAN_PORT1;
can_init.can_controller_label = {CAN_DEVICE_ONE_LABEL};

can_init.can_callbacks.can_data_indication = CAN_Data_Indication;
can_init.can_callbacks.can_rtr_indication  = CAN_Remote_Indication;
can_init.can_callbacks.can_data_confirm    = CAN_Data_Confirm;
```

```
can_init.can_callbacks.can_rtr_confirm    = CAN_Remote_Confirm;
can_init.can_callbacks.can_error         = CAN_Error_Handler;

/* Call the CAN driver initialization function. */
status = NU_CAN_Start(&can_dev_one, &can_init);

/* Initialize the FIRST device of the second port. */
can_init.can_port_id                    = CAN_PORT2;
can_init.can_controller_label = {CAN_DEVICE_TWO_LABEL};
status = NU_CAN_Start(&can_dev_two, &can_init);

/* At this point status indicates if CAN initialization was successful. */
```

Related Topics

[CAN Function Reference](#)

NU_CAN_Assign_RTR_Response

This API assigns a message buffer for automatically responding to a specified message ID request.

Usage

```
STATUS NU_CAN_Assign_RTR_Response (CAN_PACKET *can_msg)
```

Arguments

- **can_msg**
Pointer to Nucleus CAN message structure.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **CAN_INVALID_MSG_POINTER**
The given CAN message pointer is invalid.
- **CAN_ID_RANGE_ERROR**
Specified CAN message ID is out of range.
- **CAN_INVALID_DATA_LENGTH**
Specified data length is invalid. Data length set to maximum data length allowed.
- **CAN_NO_FREE_MB**
No message buffer is free to reserve for RTR response.
- **CAN_RTR_ALREADY_ASSIGNED**
The specified message ID has already been setup for RTR response. You may need to clear the previous one and then re-assign in order to modify the RTR response.
- **CAN_INVALID_MSG_ID_TYPE**
Message ID type is not set properly. It should either be **CAN_STANDARD_ID** or **CAN_EXTENDED_ID**.
- **CAN_UNSUPPORTED_PORT**
The specified CAN driver port is not supported.
- **CAN_UNSUPPORTED_CONTROLLER**
The specified CAN device is not supported.
- **CAN_DEV_NOT_INIT**
The specified CAN device/controller is not initialized.

Description

The requesting node sends an RTR packet with the ID of the message for which data is required. On receiving this request each node check if it has a matching response, previously set by a call to this API. Detection of a successful match usually triggers the transmission of the required data message. This feature is usually hardware limited and only very few automatic responses may be set. For more information, see “[Remote Transmission Request \(RTR\)](#)” on page 60.

Example

```
/* This example sets the response for a standard ID (0x541) remote
transmission request. */

/* Declare the required variables. */
CAN_PACKET CAN_Rx_Msg;
STATUS      status;
CAN_HANDLE can_dev;

/* Assume that "can_dev" contains device handle of a CAN device which has
previously been initialized with Nucleus CAN NU_CAN_Start service call. */

/* Set the CAN driver to use for communication. */
CAN_Rx_Msg.can_dev = can_dev;

/* Set the type of the Message ID to 11-bit message ID. */
CAN_Rx_Msg.can_msg_id_type = CAN_STANDARD_ID;

/* Set the message ID to 0x541. */
CAN_Rx_Msg.can_msg_id = 0x541;

/* Set the response data length to 1 byte. */
CAN_Rx_Msg.can_msg_length = 0x01;

/* Set the response data value. */
CAN_Rx_Msg.can_msg_data[0] = 0x03;

/* Set a buffer for RTR response */
status = NU_CAN_Assign_RTR_Response(&CAN_Rx_Msg);

/* At this point status indicates if the service was successful. */
```

Related Topics

[CAN Function Reference](#)

[NU_CAN_Clear_RTR_Response](#)

NU_CAN_Clear_RTR_Response

This API clears the response of the specified message ID, which was previously set by a call to [NU_CAN_Assign_RTR_Response](#). The service performs the operation correctly even if a wrong length is specified, however it is indicated to the user that the length specified was invalid.

Usage

```
STATUS NU_CAN_Clear_RTR_Response (CAN_PACKET *can_msg)
```

Arguments

- **can_msg**
Pointer to the CAN message structure.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **CAN_INVALID_MSG_POINTER**
The given CAN message pointer is invalid.
- **CAN_ID_RANGE_ERROR**
Specified CAN message ID is out of range.
- **CAN_NO_ASSIGNED_MB**
No message buffer has been assigned a response with the specified message ID.
- **CAN_INVALID_MSG_ID_TYPE**
Message ID type is not set properly. It should either be **CAN_STANDARD_ID** or **CAN_EXTENDED_ID**.
- **CAN_UNSUPPORTED_PORT**
The specified CAN driver port is not supported.
- **CAN_UNSUPPORTED_CONTROLLER**
The specified CAN device is not supported.
- **CAN_DEV_NOT_INIT**
The specified CAN device/controller is not initialized.

Example

```
/* This example clears the response for a standard ID (0x541) remote
   transmission request. */

/* Declare the required variables. */
CAN_PACKET CAN_Rx_Msg;
STATUS      status;
```

```
/* Set the CAN driver previously responsible for the transmission of the
   specified data message. */
CAN_Rx_Msg.can_dev = CAN1_DRIVER;

/* Set the message ID to 0x541. */
CAN_Rx_Msg.can_msg_id = 0x541;

/* Clear the RTR response */
status = NU_CAN_Clear_RTR_Response (&CAN_Rx_Msg);

/* At this point status indicates if the service was successful. */
```

Related Topics

[CAN Function Reference](#)

[NU_CAN_Assign_RTR_Response](#)

NU_CAN_Close_Driver

This API stops all functionality of the specified CAN device and frees the resources being utilized by the specified device.

Usage

```
STATUS NU_CAN_Close_Driver (UINT8      can_port_id,  
                           CAN_HANDLE can_dev_id)
```

Arguments

- **can_port_id**
Nucleus CAN port ID. If multiple ports support is not enabled in Nucleus CAN then this parameter is ignored and may be set to 0.
- **can_dev_id**
Handle to the device to close.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **CAN_SHUTDOWN_ERROR**
Error while closing the device
- **CAN_UNSUPPORTED_PORT**
The specified CAN driver port is not supported.
- **CAN_UNSUPPORTED_CONTROLLER**
The specified CAN device is not supported.
- **CAN_DEV_NOT_INIT**
The specified CAN device/controller is not initialized.

Description

After a call to this API, no CAN communication should be made via the specified device. A device closed using this API may be re-initialized using [NU_CAN_Start](#) API.

Example

```
/* Declare the required variables. */  
CAN_HANDLE can_dev;  
  
/* Assume that "can_dev" contains device handle of a CAN device which has  
previously been initialized with Nucleus CAN NU_CAN_Start service call. */  
  
STATUS status;
```

```
/* Shutdown first CAN device. */  
status = NU_CAN_Close_Driver (0, can_dev);  
  
/* At this point status indicates whether or not the driver was shutdown  
successfully. */
```

Related Topics

[CAN Function Reference](#)

[NU_CAN_Start](#)

NU_CAN_Get_Baud_Rate

This API function can be used to get the current baud rate value of the node.

Usage

```
UINT16 NU_CAN_Get_Baud_Rate (UINT8    can_port_id,  
                             CAN_HANDLE can_dev_id)
```

Arguments

- **can_port_id**
Nucleus CAN port ID. If multiple ports support is not enabled in Nucleus CAN then this parameter is ignored and may be set to 0.
- **can_dev_id**
Nucleus CAN device handle.

Return Values

- Baud rate
Current baud rate of the node.
- 0
The specified CAN device not supported.

Example

```
/* Declare the required variables. */  
CAN_HANDLE can_dev;  
  
/* Assume that "can_dev" contains device handle of a CAN device which has  
previously been initialized with Nucleus CAN NU_CAN_Start service call. */  
  
UINT16 baud;  
  
/* Get the baud rate of first CAN device. */  
baud = NU_CAN_Get_Baud_Rate (0, can_dev);
```

Related Topics

[CAN Function Reference](#)

[NU_CAN_Set_Baud_Rate](#)

NU_CAN_Get_Node_State

This API gets the last reported state of the local node.

Usage

```
STATUS NU_CAN_Get_Node_State (UINT8      can_port_id,  
                             CAN_HANDLE can_dev_id)
```

Arguments

- **can_port_id**
Nucleus CAN port ID. If multiple ports support is not enabled in Nucleus CAN then this parameter is ignored and may be set to 0.
- **can_dev_id**
Nucleus CAN device handle.

Return Values

- **CAN_UNSUPPORTED_PORT**
Invalid CAN driver port specified.
- **CAN_UNSUPPORTED_CONTROLLER**
Invalid CAN controller specified.
- **CAN_ERROR_ACTIVE_STATE**
CAN node is in error active state. This is the normal state for Nucleus CAN.
- **CAN_ERROR_PASSIVE_STATE**
CAN node is in error passive state.
- **CAN_BUS_OFF_STATE**
CAN node is in bus-off state.
- **CAN_BIT_ERROR**
Bit error detected by CAN node.
- **CAN_FRAME_ERROR**
Framing error reported to CAN node.
- **CAN_ACK_ERROR**
CAN node could not receive an acknowledgement of a transmitted message.
- **CAN_CRC_ERROR**
CRC error reported to CAN node.
- **CAN_ERROR_LIGHT**
The CAN node is experiencing some errors on the bus.

- **CAN_ERROR_HEAVY**
The node is experiencing too many errors on the bus.
- **CAN_ERROR_FATAL**
The node has stopped taking part in communication due to excessive errors.
- **CAN_HW_OVERRUN**
Hardware overflow in message box/mail box occurred.
- **CAN_GENERAL_HARDWARE_ERROR**
An undocumented CAN controller specific error occurred.

Example

```
/* Declare the required variables. */
CAN_HANDLE can_dev;

/* Assume that "can_dev" contains device handle of a CAN device which has
previously been initialized with Nucleus CAN NU_CAN_Start service call. */

STATUS status;

/* Get the last reported state of the first CAN device on this target. */
status = NU_CAN_Get_Node_State(0, can_dev);

/* At this point status indicates the last reported state of the node. */
```

Related Topics

[CAN Function Reference](#)

NU_CAN_Receive_Data

This API gets the data from the input queue of Nucleus CAN when the application gets an indication from Nucleus CAN that the node has received a data message. The pointers passed through arguments is used to receive the CAN message from the input queue of the signaling device.

Usage

```
STATUS NU_CAN_Receive_Data (CAN_PACKET *can_msg)
```

Arguments

- **can_msg**
Pointer to Nucleus CAN message structure.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **CAN_QUEUE_EMPTY**
Input queue has no received messages in store.
- **CAN_UNSUPPORTED_PORT**
The specified CAN driver port is not supported.
- **CAN_UNSUPPORTED_CONTROLLER**
The specified CAN device is not supported.
- **CAN_DEV_NOT_INIT**
The specified CAN device/controller is not initialized.
- **CAN_INVALID_PARAMETER**
Invalid parameter value/s passed to the API.
- **CAN_DEV_SLEEPING**
The specified CAN device/controller is in sleep mode.

Example

```
/* Assume the following variable has been declared globally. */
CAN_PACKET can_msg;

/* Declare the required variables. */
STATUS      status;
CAN_HANDLE  can_dev;

/* Assume that "can_dev" contains device handle of a CAN device which has
previously been initialized with Nucleus CAN NU_CAN_Start service call. */
```

```
/* Specify the CAN device from which to receive the message. */  
can_msg.can_dev = can_dev;  
  
/* Receive the message from the receive queue */  
status = CAN_Receive_Data (&can_msg);  
  
/* At this point status indicates if the service was successful. */
```

Related Topics

[CAN Function Reference](#)

NU_CAN_Request_Data_Transfer

This API is used to transmit a CAN data message on the CAN network.

Usage

```
STATUS NU_CAN_Request_Data_Transfer (CAN_PACKET *can_msg)
```

Arguments

- `can_msg`
Pointer to the CAN data message to transmit.

Return Values

- `NU_SUCCESS`
Service completed successfully.
- `CAN_INVALID_MSG_POINTER`
The given CAN message pointer is invalid.
- `CAN_ID_RANGE_ERROR`
Specified CAN message ID is out of range.
- `CAN_INVALID_DATA_LENGTH`
Specified data length is invalid. Data length. Set to maximum data length allowed.
- `CAN_NO_FREE_MB`
No transmission message buffer is free.
- `CAN_INVALID_MSG_ID_TYPE`
Message identifier is not correctly specified. It should be either `CAN_STANDARD_ID` or `CAN_EXTENDED_ID`
- `CAN_QUEUE_FULL`
Output queue is full. No more packets can be put into the output queue.
- `CAN_UNSUPPORTED_PORT`
The specified CAN driver port is not supported.
- `CAN_UNSUPPORTED_CONTROLLER`
The specified CAN device is not supported.
- `CAN_DEV_NOT_INIT`
The specified CAN device/controller is not initialized.
- `CAN_DEV_SLEEPING`
The specified CAN device/controller is in sleep mode.

Description

A CAN_PACKET structure must be filled properly with the required fields to transmit the data packet. However if the length is not specified correctly the service sets the length to maximum allowed value and transmits the message.

Example

```
/* This example prepares and request to transmit a standard data message
   on CAN network. */

/* Declare the required variables. */
CAN_PACKET can_msg;
STATUS      status;

/* Set the CAN driver to use for communication. */
can_msg.can_dev_id = CAN1_DRIVER;

/* Set the type of the Message ID to 11-bit message ID. */
can_msg.can_msg_id_type = CAN_STANDARD_ID;

/* Set the message ID to 0x234. */
can_msg.can_msg_id = 0x234;

/* Set the data length to 4 byte. */
can_msg.can_msg_length = 4;

/* Set the transmit data value. */
can_msg.can_msg_data[0] = 0x03;
can_msg.can_msg_data[1] = 0x00;
can_msg.can_msg_data[2] = 0x09;
can_msg.can_msg_data[3] = 0x01;

/* Request to transmit the data message. */
status = NU_CAN_Request_Data_Transfer (&can_msg);

/* At this point status indicates if the service was successful. */
```

Related Topics

[CAN Function Reference](#)

[NU_CAN_Assign_RTR_Response](#)

[NU_CAN_Get_Node_State](#)

NU_CAN_Request_Remote_Transfer

This API is used to transmit a CAN RTR message on the CAN network.

Usage

```
STATUS NU_CAN_Request_Remote_Transfer (CAN_PACKET *can_msg)
```

Arguments

- **can_msg**
Pointer to the CAN message to transmit.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **CAN_INVALID_MSG_POINTER**
The given CAN message pointer is invalid.
- **CAN_ID_RANGE_ERROR**
Specified CAN message ID is out of range.
- **CAN_INVALID_DATA_LENGTH**
Specified data length is invalid. Data length. Set to maximum data length allowed.
- **CAN_NO_FREE_MB**
No transmission message buffer is free.
- **CAN_INVALID_MSG_ID_TYPE**
Message identifier is not correctly specified. It should be either **CAN_STANDARD_ID** or **CAN_EXTENDED_ID**
- **CAN_QUEUE_EMPTY**
Output queue is empty. The driver couldn't get the packet from the application.
- **CAN_QUEUE_FULL**
Output queue is full. No more packets can be put into the output queue.
- **CAN_UNSUPPORTED_PORT**
The specified CAN driver port is not supported.
- **CAN_UNSUPPORTED_CONTROLLER**
The specified CAN device is not supported.
- **CAN_DEV_NOT_INIT**
The specified CAN device/controller is not initialized.

- CAN_DEV_SLEEPING

The specified CAN device/controller is in sleep mode.

Description

A CAN_PACKET structure must be filled properly with the required fields to transmit the data packet. However if the length is not specified correctly the service sets the length to maximum allowed value and transmits the message. Also it is not necessary to fill the data for the RTR message, as RTR packets carry no data.

Example

```
/* This example prepares and request to transmit a standard RTR message on
CAN network. */

/* Declare the required variables. */
CAN_PACKET can_msg;
STATUS      status;
CAN_HANDLE can_dev;

/* Assume that "can_dev" contains device handle of a CAN device which has
previously been initialized with Nucleus CAN NU_CAN_Start service call. */

/* Set the CAN driver to use for communication. */
can_msg.can_dev = can_dev;

/* Set the type of the Message ID to 11-bit message ID. */
can_msg.can_msg_id_type = CAN_STANDARD_ID;

/* Set the message ID to 0x234. */
can_msg.can_msg_id = 0x234;

/* Set the data length to 4 byte. This is the required number of data bytes
from the RTR response. */
can_msg.can_msg_length = 4;

/* Request to transmit RTR message. */
status = NU_CAN_Request_Remote_Transfer (&can_msg);

/* At this point status indicates if the service was successful. */
```

Related Topics

[CAN Function Reference](#)

[NU_CAN_Assign_RTR_Response](#)

[NU_CAN_Get_Node_State](#)

NU_CAN_Set_Acceptance_Mask

This API provides the facility to set acceptance mask at hardware level for the incoming message ID's.

Usage

```
STATUS NU_CAN_Set_Acceptance_Mask (UINT8      can_port_id,  
                                  CAN_HANDLE  can_dev_id,  
                                  UINT8      buffer_no,  
                                  UINT32     mask_value)
```

Arguments

- **can_port_id**
Nucleus CAN port ID. If multiple ports support is not enabled in Nucleus CAN then this parameter is ignored and may be set to 0.

- **can_dev_id**
Nucleus CAN device handle.

- **buffer_no**
The message buffer that will be configured for the mask. Possible values for this include:

CAN_STANDARD_RECEIVE_MB

CAN_EXTENDED_RECEIVE_MB

CAN_GLOBAL_RECEIVE_MB

In addition to this, it is possible for some CAN controllers to configure individual mask buffers per each message buffer. In that case the number of the mask buffer may be passed for this parameter counting the first mask buffer numbered as '0'. Refer to the controller's documentation for more information on using that feature.

- **mask_value**
Mask value for the buffer.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **CAN_NO_MASK_BUFF**
Specified mask buffer is not supported by the device.
- **CAN_SERVICE_NOT_SUPPORTED**
The service is not supported by CAN driver.
- **CAN_UNSUPPORTED_PORT**
The specified CAN driver port is not supported.

- **CAN_UNSUPPORTED_CONTROLLER**
 The specified CAN device is not supported.
- **CAN_DEV_NOT_INIT**
 The specified CAN device/controller is not initialized.

Description

A bit set as '0' in the mask means that do not mask the incoming message ID bit at that position rather consider it as passed, and vice versa. This API may be used to restrict the reception of standard/extended IDs to a subset of the full range.

This is an optional feature and some controllers may not provide this service. In the case Nucleus CAN will return the status as **CAN_SERVICE_NOT_SUPPORTED**.

Example 1

```
/* Declare the required variables. */
UINT8      buffer_no;
UINT32     mask_value;
STATUS     status;
CAN_HANDLE can_dev;

/* Assume that "can_dev" contains device handle of a CAN device which has
previously been initialized with Nucleus CAN NU_CAN_Start service call. */

/* Configure mask to receive standard ID messages into the buffer. */
buffer_no  = CAN_STANDARD_RECEIVE_MB;
mask_value = CAN_STANDARD_RECEIVE_MB_MASK;

/* Set the mask for standard message reception buffer on first CAN device
of the target. */
status = NU_CAN_Set_Acceptance_Mask (0, can_dev, buffer_no, mask_value);
```

Example 2

```
/* Declare the required variables. */
UINT8      buffer_no;
UINT32     mask_value;
STATUS     status;
CAN_HANDLE can_dev;

/* Assume that "can_dev" contains device handle of a CAN device which has
previously been initialized with Nucleus CAN NU_CAN_Start service call. */

/* Configure mask for message buffer 7 to receive messages having matching
values in their lowest nibble. */
buffer_no  = 7;
mask_value = 0x0000000F;

/* Set the mask for on first CAN device of the target. */
status = NU_CAN_Set_Acceptance_Mask (0, can_dev, buffer_no, mask_value);
```

Related Topics

[CAN Function Reference](#)

NU_CAN_Set_Baud_Rate

This API provides the facility to switch baud rate of a CAN nodes at run-time.

Usage

```
STATUS NU_CAN_Set_Baud_Rate (UINT8    can_port_id,  
                             CAN_HANDLE can_dev_id,  
                             UINT16    baud_rate)
```

Arguments

- **can_port_id**
Nucleus CAN port ID. If multiple ports support is not enabled in Nucleus CAN then this parameter is ignored and may be set to 0.
- **can_dev_id**
Nucleus CAN device handle.
- **baud_rate**
New baud rate to set for the specified node.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **CAN_INVALID_BAUDRATE**
Given baud rate is not supported. CAN controller set to its default baud rate.
- **CAN_UNSUPPORTED_PORT**
The specified CAN driver port is not supported.
- **CAN_UNSUPPORTED_CONTROLLER**
The specified CAN device is not supported.
- **CAN_DEV_NOT_INIT**
The specified CAN device/controller is not initialized.
- **CAN_DEV_SLEEPING**
The specified CAN device/controller is in sleep mode.

Description

If a particular node does not support a baud rate, which is given as an input parameter, the baud rate of the device is set to the default baud rate. The value of default baud rate as specified by Nucleus CAN is usually 500 kbps. For more information, see [“CAN Baud Rate”](#) on page 61. This API should only be executed synchronously on a CAN network.

Example

```
/* Declare the required variables. */
UINT16      baud_rate;
STATUS      status;
CAN_HANDLE  can_dev;

/* Assume that "can_dev" contains device handle of a CAN device which has
previously been initialized with Nucleus CAN NU_CAN_Start service call. */

/* Configure the baud rate to 1000 Kbps. */
baud_rate = 1000;

/* Set the baud rate first CAN device. */
status = NU_CAN_Set_Baud_Rate (0, can_dev, baud_rate);

/* At this point status indicates if the specified baud rate was set
successfully. */
```

Related Topics

[CAN Function Reference](#)

NU_CAN_Sleep

This API provides the facility to switch a running CAN node to sleep mode.

Usage

```
STATUS NU_CAN_Sleep (UINT8      can_port_id,  
                    CAN_HANDLE can_dev_id)
```

Arguments

- **can_port_id**
Nucleus CAN port ID. If multiple ports support is not enabled in Nucleus CAN then this parameter is ignored and may be set to 0.
- **can_dev_id**
Nucleus CAN device handle.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **CAN_SERVICE_NOT_SUPPORTED**
The service is not supported by Nucleus CAN driver.
- **CAN_UNSUPPORTED_PORT**
The specified CAN driver port is not supported.
- **CAN_UNSUPPORTED_CONTROLLER**
The specified CAN device is not supported.
- **CAN_DEV_NOT_INIT**
The specified CAN device/controller is not initialized.

Description

In sleep mode the node will not transmit anything on the network, however it might be able to wake up and resume communication on the reception of a frame. If this functionality is not supported by hardware, then the controller must be awakened by calling [NU_CAN_Wakeup](#). This API is not supported in loop back mode.

Example

```
/* Declare the required variables. */  
STATUS      status;  
CAN_HANDLE  can_dev;  
  
/* Assume that "can_dev" contains device handle of a CAN device which has  
previously been initialized with Nucleus CAN NU_CAN_Start service call. */  
  
/* Sleep the first CAN node on the target. */  
status = NU_CAN_Sleep (0, can_dev);
```

```
/* At this point status indicates if the service was successful. */
```

Related Topics

[CAN Function Reference](#)

NU_CAN_Wakeup

This API provides the facility to switch a sleeping CAN node to wake up and resume its normal communication on the network. This API is not supported in loop back mode.

Usage

```
STATUS NU_CAN_Wakeup (UINT8      can_port_id,  
                     CAN_HANDLE can_dev_id)
```

Arguments

- **can_port_id**
Nucleus CAN port ID. If multiple ports support is not enabled in Nucleus CAN then this parameter is ignored and may be set to 0.
- **can_dev_id**
Nucleus CAN device handle.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **CAN_SERVICE_NOT_SUPPORTED**
The service is not supported by Nucleus CAN driver.
- **CAN_UNSUPPORTED_PORT**
The specified CAN driver port is not supported.
- **CAN_UNSUPPORTED_CONTROLLER**
The specified CAN device is not supported.
- **CAN_DEV_NOT_INIT**
The specified CAN device/controller is not initialized.

Example

```
/* Declare the required variables. */  
STATUS      status;  
CAN_HANDLE  can_dev;  
  
/* Assume that "can_dev" contains device handle of a CAN device which has  
previously been initialized with Nucleus CAN NU_CAN_Start service call. */  
  
/* Wake up the first CAN node on the target. */  
status = NU_CAN_Wakeup (0, can_dev);  
  
/* At this point status indicates if the service was successful. */
```

Related Topics

[CAN Function Reference](#)

CAN Configuration Options

This section describes the configuration options available for Nucleus CAN. Configurations are set in either the *os/connectivity/can/.metadata* or *os/include/connectivity/can_cfg.h*.

Table 2-1 lists the default setting for each Nucleus options and a link to more information.

Table 2-1. Nucleus CAN Configuration

Option	Default Value/Description
NU_CAN_OPERATING_MODE	NU_CAN_HW_DRIVER_NORMAL
NU_CAN_SUPPORTS_RTR	1
NU_CAN_AUTOMATIC_RTR_RESPONSE	1
NU_CAN_REJECT_SELF_TX_PACKET	1
NU_CAN_DEBUG	1
NU_CAN_OPTIMIZE_FOR_SIZE	1
NU_CAN_OPTIMIZE_FOR_SPEED	0
NU_CAN_ENABLE_TIME_STAMP	0
NU_CAN_MULTIPLE_PORTS_SUPPORT	0
NU_CAN_MMU_SUPPORT	0

Note



Unless otherwise specified, the following configurations are set in the *os/include/connectivity/can_cfg.h*.

Driver Selection (Loopback/Hardware)

By default, Nucleus CAN runs in loopback mode, so you can build an application before a CAN controller and CAN network are established. The CAN loopback device provides the necessary functionality to test a CAN application. However, the CAN loopback device does not provide the bus error simulation. The device always assumes to be operating correctly.

This option is controlled through the metadata of the generic CAN driver and is set in the *os/connectivity/can/.metadata* file. You should not change any configuration outside this metadata file.

The following configurations are available:

- **Loopback Device Selection** — Nucleus CAN is set to run in loopback mode by setting `NU_CAN_OPERATING_MODE` to `NU_CAN_LOOPBACK`. In this case, Nucleus

CAN does not require the Nucleus CAN Driver library, and it should not be in the link line for the Nucleus CAN application.

- **Hardware Driver Selection** — A CAN network would be running using CAN controllers, therefore, for normal operation Nucleus CAN requires CAN hardware driver for the CAN controller. The normal support of CAN hardware driver(s) may be enabled in Nucleus CAN by setting `NU_CAN_OPERATING_MODE` to `NU_CAN_HW_DRIVER_NORMAL`.
- **Normal Hardware Driver** — This is the mode when `NU_CAN_OPERATING_MODE` is set as follows:

```
#define NU_CAN_OPERATING_MODE NU_CAN_HW_DRIVER_NORMAL
```

In this mode Nucleus CAN is able to transmit and receive data/RTR messages and send responses to RTR messages. This should be the normal working mode for a Nucleus CAN application requiring maximum functionality support.

- **Hardware Loopback** — Some CAN controllers support a loopback at hardware level. In this case the output stream is fed back as the input to the controller. The controller behaves as if it has received the data from the network. To enable the support for hardware loopback, make sure `NU_CAN_OPERATING_MODE` is set as follows:

```
#define NU_CAN_OPERATING_MODE NU_CAN_HW_DRIVER_LOOPBACK
```

This is a controller and hardware driver dependant feature.

- **Listen Only Mode** — Some CAN controller support a listen only mode. In this case the CAN controller is able to receive the data, without sending acknowledgement but not to transmit the data. This mode can be used to build CAN sniffer type testing tools To enable the support for hardware loopback, make sure `NU_CAN_OPERATING_MODE` is set as follows:

```
#define NU_CAN_OPERATING_MODE NU_CAN_HW_DRIVER_LISTENONLY
```

The following APIs are not available in listen only mode:

- [NU_CAN_Request_Data_Transfer](#)
- [NU_CAN_Request_Remote_Transfer](#)
- [NU_CAN_Assign_RTR_Response](#)
- [NU_CAN_Clear_RTR_Response](#)

Listen only mode is a controller and hardware driver dependant feature.

Remote Transmission Request (RTR)

Nucleus CAN provides the RTR option to enable you to make optimizations for the CAN application. In an application where RTR frames type is not used, you can reduce the size of the code by disabling the RTR support.

- **RTR Support Selection** — In order to enable RTR services support in Nucleus CAN and Nucleus CAN Driver, make sure to set the following define to '1'.

```
#define NU_CAN_SUPPORTS_RTR 1
```

By default, RTR support is enabled in Nucleus CAN and CAN hardware driver should support it.

Setting CAN_SUPPORT_RTR to '0' disables the support for RTR transmission and response assignment services in Nucleus CAN. The following APIs are not available if RTR is not supported.

- [NU_CAN_Request_Remote_Transfer](#)
 - [NU_CAN_Assign_RTR_Response](#)
 - [NU_CAN_Clear_RTR_Response](#)
- **RTR Response Configuration** — Nucleus CAN provides the facility to automatically respond to incoming Remote Transmission Requests from other nodes in CAN network. This option is enabled by default as the following define is set to '1'.

```
#define NU_CAN_AUTOMATIC_RTR_RESPONSE 1
```

When the automatic RTR response option is enabled, only those requests will be answered automatically that have been assigned using [NU_CAN_Assign_RTR_Response](#) API. Other requests will be ignored by the hardware driver. It also important to note that automatic response is available for only a limited number of messages due to a limited number of message buffers available on a CAN controller or CAN hardware driver in case they are not available on the controller. Refer to your controller documentation for further information on the number of message buffers on the underlying CAN controller.

Setting CAN_AUTOMATIC_RTR_RESPONSE to '0' does the following:

- Disables the Automatic RTR response. You must send the RTR response manually. The application will be notified of an arriving RTR through a callback, whereby it may send the response or ignore it.
- Makes the following APIs unavailable:
 - [NU_CAN_Assign_RTR_Response](#)
 - [NU_CAN_Clear_RTR_Response](#)

Queue Size

Nucleus CAN provides queues for message input and output. Each element of the queue stores a complete CAN message. The length of the input and output queues CAN be configured as per application need and available resources. Change the following defines to your desired value but never set them to '0'. The minimum value for these is '1'. The default values for the input and output queues are listed as follows.

```
#define CAN_INQUEUE_SIZE 128
#define CAN_OUTQUEUE_SIZE 64
```

Maximum CAN Driver

The maximum number of CAN controllers integrated with Nucleus CAN is represented by CAN_MAX_DRIVERS in the `<can_controller>_driver.h` file of Nucleus CAN Driver. Most of the processors supporting CAN provide more than one CAN controller, and the value of this macro is usually greater than '1'. If your application needs only one CAN controller, set this to 1. This helps reduce the code size for your final product.

CAN Baud Rate

Nucleus CAN sets the default baud rate to 1000 kbps/1Mbps. It is indicated as CAN_DEFAULT_BAUDRATE in the configuration file. You may change the default baud rate to any value supported by your CAN hardware driver.

This option is controlled through the baud_rate option dev_settings group in board's platform file. You can set the baud rate as follows:

```
<BOARD>.<CAN_CONTROLLER>.dev_settings.baud_rate = 1000
```

Self Transmitted Frame Handling

Nucleus CAN may receive the frames transmitted by it if they are not rejected at hardware level by the CAN controller. By default, self-transmitted frames are rejected by Nucleus CAN Driver.

```
#define NU_CAN_REJECT_SELF_TX_PACKET 1
```

To receive the self transmitted frames, NU_CAN_REJECT_SELF_TX_PACKET must be set to '0'.

CAN messages with a '0' ID are always received back regardless of the configuration of this option.

Debug Mode

Nucleus CAN provides the application developer with the ease of a debug mode where all the API parameters are checked for validity. This option is enabled by default. It is useful in reducing the size of the code and speeding up the communication. It may be turned off when an application is fully developed. To disable this option change `NU_CAN_DEBUG` to '0'.

```
#define NU_CAN_DEBUG 0
```

Optimization

Nucleus CAN provides this option for the case where a target has limited resources or speed-up of the application is required. By default, optimizations are disabled in Nucleus CAN to provide maximum ease in application development.

```
#define NU_CAN_OPTIMIZE_FOR_SIZE 0  
#define NU_CAN_OPTIMIZE_FOR_SPEED 0
```

These options are controlled through the metadata of the generic CAN driver and is set in the *os/connectivity/can/.metadata* file. You should not change any configuration outside this metadata file.

To enable optimization for speed, set `NU_CAN_OPTIMIZE_FOR_SPEED` to '1' and to enable optimization for size set `NU_CAN_OPTIMIZE_FOR_SIZE` to '1'.

Turning any of these optimizations ON in Nucleus CAN disables the `NU_CAN_DEBUG` mode as well.

Time Stamp

Some CAN controllers provide time stamp for messages transmitted/received on CAN network. Nucleus CAN may be configured to forward this timestamp to the application by setting `NU_CAN_ENABLE_TIME_STAMP` to '1'. By default, time stamping support is disabled in Nucleus CAN.

```
#define NU_CAN_ENABLE_TIME_STAMP 0
```

The value of the time stamp is forwarded 'as is' from the CAN controller. Refer to the controller's documentation for description of the time stamp value.

When optimization for Nucleus CAN is enabled, time stamp support is unavailable.

Multiple Port

Nucleus CAN may be configured to handle multiple CAN hardware driver ports simultaneously. By default, this option is not enabled in Nucleus CAN.

```
#define NU_CAN_MULTIPLE_PORTS_SUPPORT 0
```

It may be enabled by setting `NU_CAN_MULTIPLE_PORTS_SUPPORT` to '1'.

If this option is enabled, you must configure

bsp/<board>/include/bsp/drivers/can/<processor>/<can_controller>_driver.h to include initialization functions for all of the ports.

MMU Support

Nucleus CAN provides MMU support, where an OS provides this facility. By default, MMU support is disabled. To enable it, include the following:

```
#define NU_CAN_MMU_SUPPORT 0
```

MMU support in CAN may be enabled by setting `NU_CAN_MMU_SUPPORT` to '1'. Refer to the appropriate OS manual to determine how to enable the MMU support in the OS before enabling this option in Nucleus CAN.

CAN Error Codes

Table 2-2 describes the error codes returned to the application by Nucleus CAN.

Table 2-2. Nucleus CAN Error Codes

Symbol	Value	Description
CAN_GENERAL_ERROR	1	An unknown error in Nucleus CAN.
CAN_NULL_GIVEN_FOR_MEM_POOL	3	Memory pointer was not set properly and was pointing to null.
CAN_ID_RANGE_ERROR	11	ID of the specified CAN message is out of range.
CAN_INVALID_MSG_ID_TYPE	12	CAN message ID type is not standard or extended.
CAN_NOT_RTR_MSG	13	The specified message is not an RTR.
CAN_NOT_DATA_MSG	14	The specified message is not a data message.
CAN_INVALID_MSG_TYPE	15	The specified message type is not valid. (data/RTR)
CAN_INVALID_DATA_LENGTH	16	Message length of the CAN message is not valid.
CAN_RTR_ALREADY_ASSIGNED	17	The specified message ID is already assigned as RTR response message.
CAN_UNSUPPORTED_PORT	31	The specified CAN port has not been integrated with Nucleus CAN.

Table 2-2. Nucleus CAN Error Codes (cont.)

Symbol	Value	Description
CAN_UNSUPPORTED_CONTROLLER	32	The specified CAN device (controller) is not supported. In other words, either the controller is non-existent or the driver does not support the controller.
CAN_SERVICE_NOT_SUPPORTED	33	The requested Nucleus CAN service is not supported. It is either not implemented or is disabled as per current Nucleus CAN configuration.
CAN_INVALID_POINTER	34	The specified pointer is null.
CAN_INVALID_MSG_POINTER	35	The specified CAN message pointer is null.
CAN_SHUTDOWN_ERROR	36	The specified CAN controller can not be shutdown.
CAN_INVALID_PARAMETER	37	Value of a parameter given to the API is not valid.
CAN_DEV_ALREADY_INIT	38	Nucleus CAN device is already initialized.
CAN_DEV_NOT_INIT	39	Nucleus CAN device is not initialized.
CAN_DEV_SLEEPING	40	Nucleus CAN device is currently in sleep mode.
CAN_QUEUE_EMPTY	51	Nucleus CAN message queue is empty.
CAN_QUEUE_FULL	52	Nucleus CAN message queue is full.
CAN_INPUT_QUEUE_ERROR	53	Error in Nucleus CAN message input queue.
CAN_OUTPUT_QUEUE_ERROR	54	Error in Nucleus CAN message output queue.
CAN_SW_OVERRUN	55	Error in Nucleus CAN message output queue.
CAN_IN_LOOPBACK	61	Nucleus CAN is running in loopback.
CAN_LOOPBACK_TX_ERROR	62	Error in message transmission for Nucleus CAN loopback.
CAN_LOOPBACK_RX_ERROR	63	Error in message reception for Nucleus CAN loopback.
CAN_LOOPBACK_NOT_INIT	64	Nucleus CAN loopback is not initialized.
CAN_GENERAL_HARDWARE_ERROR	100	An unspecified error occurred in CAN hardware.
CAN_HW_OVERRUN	101	Hardware overrun while receiving CAN messages.
CAN_ERROR_ACTIVE_STATE	102	CAN is currently in error active state which is normal behavior.

Table 2-2. Nucleus CAN Error Codes (cont.)

Symbol	Value	Description
CAN_ERROR_PASSIVE_STATE	103	CAN is currently in error passive state.
CAN_BUS_OFF_STATE	104	CAN node has entered in bus off state due to heavy bus disturbances or errors.
CAN_BIT_ERROR	105	Bit error occurred for CAN controller.
CAN_FRAME_ERROR	106	CAN framing error occurred.
CAN_ACK_ERROR	107	CAN message acknowledge error.
CAN_CRC_ERROR	108	CRC error occurred.
CAN_ERROR_LIGHT	109	CAN node is experiencing errors on the bus. The node is still able to transmit and receive but with errors.
CAN_ERROR_HEAVY	110	CAN node is experiencing heavy errors on the bus but communication is still possible.
CAN_ERROR_FATAL	111	Heavy errors continued for quite some time so CAN node has stopped taking part in communication.
CAN_NO_FREE_MB	112	No message buffer/mailbox is free.
CAN_NO_ASSIGNED_MB	113	No message buffer/mailbox has been assigned for RTR.
CAN_NO_MASK_BUFF	114	The specified mask buffer does not exist.
CAN_INVALID_BAUD_RATE	115	The specified baud rate is not valid. The node baud rate set to default baud rate for Nucleus CAN.
CAN_TRANSMISSION_ABORTED	116	Transmission of the message has been aborted.

Chapter 3

Inter Integrated Circuit (I2C)

This chapter describes the Nucleus I2C (Inter Integrated Circuit) bus software and the requirements for using it.

Caution



To guarantee future support and compatibility for your application, only use the Nucleus interfaces, structures, macros, and so on as documented.

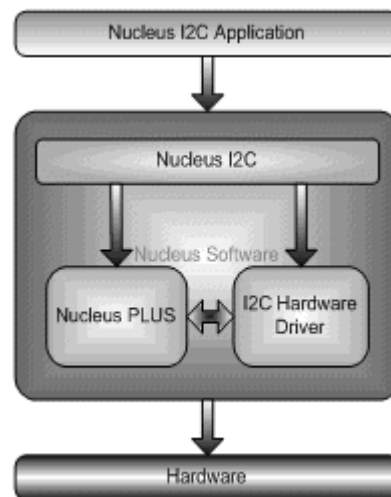
I2C Module Overview

Nucleus I2C is an embedded implementation of the I2C-BUS Specification V2.1 that provides two types of APIs:

- **Automated** — provides all communication operations including start, transmission/reception, stop, and the final output. You only need to provide the initial parameters. This API can be used in applications that do not require custom specifications.
- **Fine control** — allows you to specify the communication behavior for the I2C node. These APIs can be configured for a custom application.

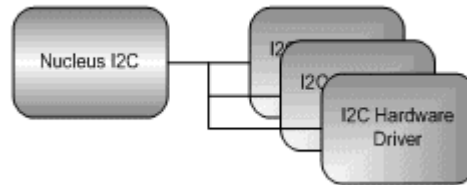
Figure 3-1 shows the basic components in a Nucleus-based I2C system.

Figure 3-1. Nucleus I2C Based System



Nucleus I2C supports both 7-bit and 10-bit addressing schemes as well as special addressing schemes. Nucleus I2C software stack also provides support for handling an indefinite number of communication channels (data transfers between masters and slaves) on multiple I2C networks as shown in [Figure 3-2](#).

Figure 3-2. Interaction of Nucleus I2C with Multiple I2C Interfaces



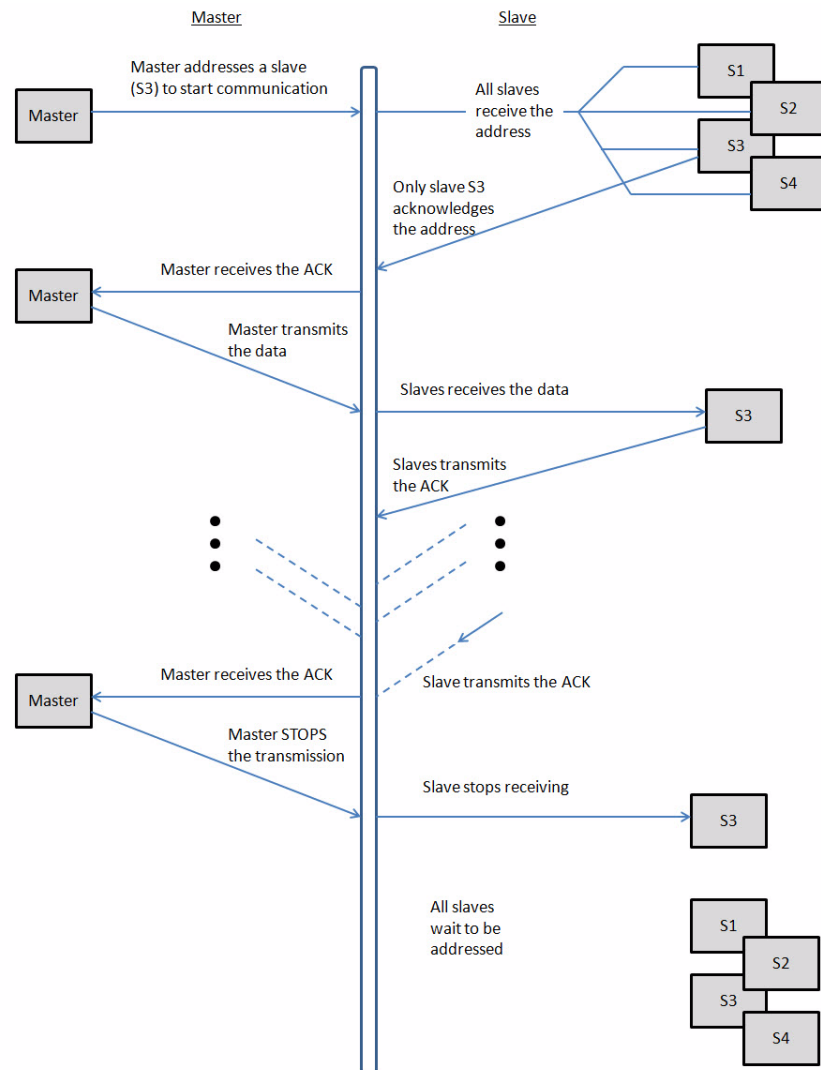
Standard I2C Model

I2C (also known IICbus) is a serial communication bus that transfers information among ICs and PCBs at shorter distances and moderate data rates. It uses a two-wired bus with byte-oriented bi-directional transfer. One of the wires carries a data signal (SDA) and the other carries a clock signal (SCL). The data is only significant when SDA SCL is high.

I2C employs the Master/Slave communication model. The data communication always takes place between two nodes; one of them being a master and the other being a slave. An I2C master is a device on the network that can initiate a transfer (transmission/reception) on the network and stop it, while a slave cannot initiate a transfer or terminate it. Also, each slave has a unique 7-bit/10-bit address, but the maximum device count within the address space provided by 7 or 10 bits is limited by a maximum bus capacity of 400pF. It is legal for a device to provide the capability of a master and a slave, but only one of the two modes can be active at any time.

The following diagram depicts a typical I2C transfer operation.

Figure 3-3. A Typical I2C Transfer



I2C Driver

Nucleus I2C driver is the target-specific add-on to Nucleus I2C that controls the specific bus for interfacing with Nucleus I2C. Each I2C hardware driver port supports multiple I2C controllers of the same type. An I2C driver port for a particular module handles all of the controllers in that module.

Enabling I2C

To enable this in an application, include the following header file in your application code:

```
<install_root>/nucleus/os/include/connectivity/nu_connectivity.h.
```

I2C Function Reference

This section describes the functions provides with Nucleus I2C.

Each function is prefixed with `NU_I2C_Master`, which means this should be used only for I2C master node/functionality. `NU_I2C_Slave` means it is intended only for I2C slave node/functionality. `NU_I2C` means the API may be used for master and/or slave node/functionality.

The I2C functions are organized into three sections as follows:

- [I2C Common API Functions](#) — describes the functions in the API that provide common functionality like Start/Close a device.
- [I2C Automatic API Functions](#) — describes functions for the Automatic API.
- [I2C Fine Control API Functions](#) — describes functions for the Fine Control API.

All of the functions are useful only after [NU_I2C_Open](#) has been called and a valid `I2C_HANDLE` has been obtained. None of the functions, except [NU_I2C_Open](#), can be called for a device after [NU_I2C_Close](#) has been called.

I2C Common API Functions

The functions in this section may be used in interrupt-driven mode as well as polling mode, whether acting as transmitter or receiver, except that polling mode for the slave receiver is not supported.

- [NU_I2C_Close](#)
- [NU_I2C_Get_Node_State](#)
- [NU_I2C_Master_Get_Baudrate](#)
- [NU_I2C_Master_Get_Callbacks](#)
- [NU_I2C_Master_Get_Mode](#)
- [NU_I2C_Master_Set_Baudrate](#)
- [NU_I2C_Master_Set_Callbacks](#)
- [NU_I2C_Master_Remove_Callbacks](#)
- [NU_I2C_Open](#)
- [NU_I2C_Receive_Data](#)
- [NU_I2C_Slave_Get_Address](#)
- [NU_I2C_Slave_Set_Address](#)

Related Topics

[I2C Function Reference](#)

NU_I2C_Close

This function closes a specified I2C device.

Usage

```
STATUS NU_I2C_Close (I2C_HANDLE i2c_dev)
```

Arguments

- **i2c_dev**
Handle to the initialized device.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_INVALID_HANDLE**
Nucleus I2C device handle is not valid.
- **I2C_OS_ERROR**
Error in the underlying operating system of Nucleus I2C.
- **I2C_SHUTDOWN_ERROR**
Error in closing Nucleus I2C device.
- **I2C_TRANSFER_STOP_FAILED**
Transfer on I2C bus could not be stopped.
- **I2C_DEVICE_IN_USE**
The device is not closed because some other user is using the device.

Description

If the device for which this function is called is currently a master node, it will first send a STOP signal and only after the STOP signal has been sent successfully it will attempt to close the device and frees the resources being utilized by the specified device. A device closed using this function can be re-initialized using [NU_I2C_Open](#) API.

Example

```
/* This example demonstrates the shut down of Nucleus I2C. */

/* Declare the required variables. */
STATUS status;

/* Close the specific device. */
status = NU_I2C_Close(I2C_Dev_Handle);

/* At this point status indicates whether or not the driver was shut
down successfully. */
```


Related Topics

[I2C Common API Functions](#)

[NU_I2C_Open](#)

NU_I2C_Get_Node_State

This API may be used to get the last reported state of the local node. This API is useful for interrupt-driven mode only.

Usage

```
STATUS NU_I2C_Get_Node_State (I2C_HANDLE i2c_dev,  
                             UINT8      *node_state)
```

Arguments

- **i2c_dev**
Handle to the initialized device. This must be used for further interaction with the device.
- **node_state**
Returned value of the current node state.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_INVALID_HANDLE**
Nucleus I2C device handle is not valid.

Description

This API may be used to get the last reported state of the local node. Possible values of the node states include:

I2C_TRANSMITTING_ADDRESS	1
I2C_WAITING_ADDRESS_ACK	2
I2C_SLAVE_ACKED	3
I2C_STOP_SENT	4
I2C_NODE_IDLE	5

This API is useful for interrupt-driven mode only.

Example

```
/* This example demonstrates get node state API of Nucleus I2C. */  
  
/* Declare the required variables. */  
STATUS      status;  
I2C_HANDLE i2c_dev;  
UINT8      node_state;  
  
/* Get the current node state. */  
status = NU_I2C_Get_Node_State (I2C_Dev_Handle, &node_state);  
  
/* At this point status indicates if the node state was retrieved  
successfully and in case of success node_state indicates the current  
state of the specified node. */
```

Related Topics

[I2C Common API Functions](#)

NU_I2C_Master_Get_Baudrate

This function returns the baud rate for the specified node.

Usage

```
STATUS NU_I2C_Master_Get_Baudrate (I2C_HANDLE i2c_dev,  
                                  UINT16      *baudrate)
```

Arguments

- **i2c_dev**
Handle to the initialized device.
- **baudrate**
Baud rate for the I2C network (in kbps).

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_INVALID_HANDLE**
Nucleus I2C device handle is not valid.
- **I2C_INVALID_PARAM_POINTER**
Pointer given to return baudrate is null.
- **I2C_NODE_NOT_MASTER**
The specified device does not support master functionality.

Description

The service is supported only if the node supports the functionality of a master. It may not represent the actual effective baud rate on the network as the slave is able to affect the baud rate by holding I2C Clock Line (SCL) for sending the ACK.

Example

```
/* This example gets the baud rate which the master had set for the  
transfer. Note that this may be different from actual baud rate as the  
slave is able to affect the baud rate by holding  
I2C Clock Line (SCL) for sending the ACK. */  
  
/* Declare the required variables. */  
STATUS status;  
UINT16 baudrate;  
  
/* Get the baud rate of the node. */  
status = NU_I2C_Get_Baudrate(I2C_Dev_Handle, &baudrate);  
  
/* At this point status indicates if Nucleus I2C baudrate was obtained was  
successfully and baudrate indicates the value in kHz. */
```

Related Topics

[I2C Common API Functions](#)

[NU_I2C_Master_Set_Baudrate](#)

NU_I2C_Master_Get_Callbacks

This API provides the facility to get the callbacks for the specified i2c slave device. The service is supported only if the specified node has the functionality of a master.

Usage

```
STATUS NU_I2C_Master_Get_Callbacks (I2C_HANDLE      i2c_dev,  
                                   UINT16          i2c_slave_address,  
                                   I2C_APP_CALLBACKS **callbacks)
```

Arguments

- **i2c_dev**
Handle to the initialized device.
- **i2c_slave_address**
The i2c slave device address.
- **callbacks**
Pointer to return the callbacks.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_INVALID_PARAM_POINTER**
Given data pointer is null.
- **I2C_INVALID_HANDLE**
Nucleus I2C device handle is not valid.

Example

```
/* This example gets the callbacks for the slave 0x28. */  
/* Declare the required variables. */  
STATUS          status;  
I2C_APP_CALLBACKS *callbacks;  
  
/* Get the callbacks for the slave address 0x28. */  
status = NU_I2C_Master_Get_Callbacks(I2C_Device, 0x28, &callbacks);  
  
/* At this point status indicates if the specified callbacks were get  
successfully */
```

Related Topics

[NU_I2C_Master_Remove_Callbacks](#)

[NU_I2C_Master_Set_Callbacks](#)

[I2C Common API Functions](#)

NU_I2C_Master_Get_Mode

This API provides the facility to get the mode (interrupt or polling) of the specified i2c device. The service is supported only if the specified node has the functionality of a master.

Usage

```
STATUS NU_I2C_Master_Get_Mode (I2C_HANDLE      i2c_dev,  
                               I2C_DRIVER_MODE *i2c_driver_mode)
```

Arguments

- **i2c_dev**
Handle to the initialized device.
- **i2c_driver_mode**
Pointer to return the driver mode. The mode is either I2C_INTERRUPT_MODE or I2C_POLLING_MODE.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_INVALID_HANDLE**
Nucleus I2C device handle is not valid.
- **I2C_INVALID_PARAM_POINTER**
Given data pointer is null.

Example

```
/* This example gets the mode for the i2c device. */  
  
/* Declare the required variables. */  
STATUS      status;  
I2C_DRIVER_MODE *i2c_driver_mode;  
  
/* Get the mode of the i2c device. */  
status = NU_I2C_Master_Get_Mode (I2C_Device, i2c_driver_mode);  
  
/* At this point status indicates if the specified callbacks were get  
   successfully */
```

Related Topics

[I2C Common API Functions](#)

NU_I2C_Master_Set_Baudrate

This API provides the facility to switch the baud rate of an I2C node at run-time. The service is supported only if the specified node has the functionality of a master.

Usage

```
STATUS NU_I2C_Master_Set_Baudrate (I2C_HANDLE i2c_dev,  
                                  UINT16      baudrate)
```

Arguments

- **i2c_dev**
Handle to the initialized device.
- **baudrate**
Baud rate value to set (in kbps).

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_INVALID_HANDLE**
Nucleus I2C device handle is not valid.
- **I2C_INVALID_BAUDRATE**
Baud rate for the Nucleus I2C network transmission is not valid.
- **I2C_NODE_NOT_MASTER**
The specified device does not support master functionality.

Example

```
/* This example sets the baud rate for the Nucleus I2C transmission. */  
  
/* Declare the required variables. */  
STATUS status;  
UINT16 baudrate;  
  
/* Configure the baudrate to default baudrate. */  
baudrate = I2C_DEFAULT_BAUDRATE;  
  
/* Set the baud rate of the master node. */  
status = NU_I2C_Set_Baudrate(I2C_Dev_Handle, baudrate);  
  
/* At this point status indicates if the specified baud rate was set  
successfully */
```

Related Topics

[I2C Common API Functions](#)

[NU_I2C_Master_Get_Baudrate](#)

NU_I2C_Master_Set_Callbacks

This API provides the facility to set the callbacks for the specified i2c slave device. The service is supported only if the specified node has the functionality of a master.

Usage

```
STATUS NU_I2C_Master_Set_Callbacks (I2C_HANDLE      i2c_dev,  
                                   UINT16          i2c_slave_address,  
                                   I2C_APP_CALLBACKS *callbacks)
```

Arguments

- **i2c_dev**
Handle to the initialized device.
- **i2c_slave_address**
The I2C slave device for which the callbacks are being added.
- **callbacks**
Pointer to the callbacks that need to be set for the specified i2c slave device.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_INVALID_PARAM_POINTER**
Given data pointer is null.
- **I2C_INVALID_HANDLE**
Nucleus I2C device handle is not valid.
- **I2C_OS_ERROR**
Error in the underlying operating system of Nucleus I2C.

Example

```
/* This example sets the callbacks for the slave 0x28. */  
  
/* Declare the required variables. */  
STATUS      status;  
I2C_APP_CALLBACKS callbacks;  
  
/* Initialize the callbacks structure. */  
callbacks.i2c_data_indication      = I2C_Demo_Data_Indication;  
callbacks.i2c_transmission_complete = I2C_Demo_Tx_Completion_Indication;  
callbacks.i2c_address_indication   = I2C_Demo_Address_Indication;  
callbacks.i2c_ack_indication        = I2C_Demo_Ack_Indication;  
callbacks.i2c_error                 = I2C_Demo_Error;  
  
/* Set the callbacks for the slave address 0x28. */  
status = NU_I2C_Master_Set_Callbacks(I2C_Device, 0x28, &callbacks);
```

```
/* At this point status indicates if the specified callbacks were set  
successfully */
```

Related Topics

[NU_I2C_Master_Get_Callbacks](#)

[NU_I2C_Master_Remove_Callbacks](#)

[I2C Common API Functions](#)

NU_I2C_Master_Remove_Callbacks

This API provides the facility to remove the callbacks for the specified i2c slave device. The service is supported only if the specified node has the functionality of a master.

Usage

```
STATUS NU_I2C_Master_Remove_Callbacks (I2C_HANDLE i2c_dev,  
                                       UINT16     i2c_slave_address)
```

Arguments

- **i2c_dev**
Handle to the initialized device.
- **i2c_slave_address**
The i2c slave device for which the callbacks need to be removed.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_INVALID_HANDLE**
Nucleus i2c device handle is not valid.
- **I2C_OS_ERROR**
Error in the underlying operating system of Nucleus I2C.

Example

```
/* This example removes the callbacks for the slave address 0x28. */  
  
/* Declare the required variables. */  
STATUS status;  
  
/* Remove the callbacks for the slave address 0x28. */  
status = NU_I2C_Master_Set_Callbacks(I2C_Device, 0x28, &callbacks);  
  
/* At this point status indicates if the specified callbacks were removed  
   successfully */
```

Related Topics

[NU_I2C_Master_Get_Callbacks](#)

[NU_I2C_Master_Set_Callbacks](#)

[I2C Common API Functions](#)

NU_I2C_Open

This service sets up the data structures for the protocol stack and initializes the specified underlying hardware driver depending upon the configuration you choose.

Usage

```
NU_I2C_Open (DV_DEV_LABEL *i2c_controller_name,  
            I2C_HANDLE    *i2c_dev,  
            I2C_INIT      *i2c_init)
```

Arguments

- **i2c_controller_name**
One or more labels that specify which I2C controller to be open. If none is passed, the default I2C controller will be used.
- **i2c_dev**
Location where a handle to the initialized device will be returned. This handle must be used for further interaction with the device.
- **i2c_init**
Pointer to the initialization structure for Nucleus I2C.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_OS_ERROR**
Error in the underlying operating system of Nucleus I2C.
- **I2C_DEV_ALREADY_INIT**
The specified device has already been initialized. You can use the returned device handle in this case.
- **I2C_DRIVER_REGISTER_FAILED**
Nucleus I2C driver could not be registered with Nucleus I2C.
- **I2C_INVALID_ADDRESS_TYPE**
Address type is neither 7-bit nor 10-bit.
- **I2C_INVALID_BAUDRATE**
The specified baud rate is not supported by the driver.
- **I2C_INVALID_DEVICE_ID**
Specified device is not valid.
- **I2C_INVALID_DRIVER_MODE**
Specified driver mode is not valid.

- **I2C_INVALID_HANDLE_POINTER**
Null given for returning the I2C_HANDLE pointer.
- **I2C_INVALID_NODE_TYPE**
Node type is not master/slave.
- **I2C_INVALID_PORT_ID**
Specified port is not valid.
- **I2C_INVALID_SLAVE_ADDRESS**
Slave address is not valid as it falls into the reserved range or is more than the maximum possible value.
- **I2C_NULL_GIVEN_FOR_INIT**
Initialization structure pointer is null.
- **I2C_NULL_GIVEN_FOR_MEM_POOL**
Memory pool pointer is null.

Description

This API initializes Nucleus I2C and must be called before using any service of Nucleus I2C, otherwise undesirable results may occur. The callbacks given in `i2c_init` are used as default callbacks. If the callbacks are not registered for a slave device, the default callbacks are used. For more information regarding the customization/configuration of Nucleus I2C initialization, see [“I2C Configuration Options”](#) on page 121.

Example

```
/* This example demonstrates the initialization of Nucleus I2C. */

/* Set the memory pool pointer for Nucleus I2C. */
i2c_init.i2c_memory_pool = system_memory;

/* Configure the driver parameters. */
i2c_init.i2c_driver_mode = I2C_INTERRUPT_MODE;
i2c_init.i2c_baudrate    = I2C_DEFAULT_BAUDRATE;

/* Specify the sizes of the buffers. */
i2c_init.i2c_tx_buffer_size = I2C_DEFAULT_TX_BUFFER_SIZE;
i2c_init.i2c_rx_buffer_size = I2C_DEFAULT_RX_BUFFER_SIZE;

/* Configure the indication callback functions for the application.*/
i2c_init.i2c_app_cb.i2c_data_indication    = I2C_Demo_Data_Indication;
i2c_init.i2c_app_cb.i2c_address_indication =
    I2C_Demo_Address_Indication;
i2c_init.i2c_app_cb.i2c_ack_indication     = I2C_Demo_Ack_Indication;
i2c_init.i2c_app_cb.i2c_error              = I2C_Demo_Error;
```

```
/* Set the slave node type and the slave address. By default the
   slave will have the functionality of the mater as well as slave.
   See i2c_cfg.h for changing the defaults. */

i2c_init.i2c_node_address.i2c_slave_address =
                                   I2C_DEFAULT_SLAVE_ADDRESS;
i2c_init.i2c_node_address.i2c_address_type = I2C_DEFAULT_ADDRESS_TYPE;
i2c_init.i2c_node_address.i2c_node_type     = I2C_MASTER_SLAVE;

/* Start the specified I2C device and get the handle to the
   initialized device. */
status = NU_I2C_Open (NU_NULL, &I2C_Device, &i2c_init);

/* At this point status indicates if Nucleus I2C initialization was
   successful. */
```

Related Topics

[I2C Common API Functions](#)

[NU_I2C_Close](#)

NU_I2C_Receive_Data

This API gets the data received from the transmitter.

Usage

```
STATUS NU_I2C_Receive_Data (I2C_HANDLE    i2c_dev,
                           UINT8          *data,
                           UNSIGNED_INT   length)
```

Arguments

- **i2c_dev**
Handle to the initialized device.
- **data**
Data pointer.
- **length**
Data length.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_INVALID_HANDLE**
Nucleus I2C device handle is not valid.
- **I2C_BUFFER_EMPTY**
Data buffer is empty.
- **I2C_BUFFER_HAS_LESS_DATA**
Requested data is more than the buffer has.
- **I2C_BUFFER_NOT_EMPTY**
Buffer has some more bytes even after the reading.
- **I2C_INVALID_PARAM_POINTER**
Given data pointer is null.
- **I2C_INVALID_RX_DATA_LENGTH**
Invalid data length to put in the receive buffer.
- **I2C_RX_BUFFER_NOT_ENOUGH**
Insufficient receive buffer memory space.

Description

In interrupt-driven mode, this API should only be called after the application has been given an indication through the I2C data indication callback as specified at the initialization time. The

device handle passed to the API should be the one from which the data has been received and the length should be equal to the length as told by the data indication callback function. When the master receiver is running in polling mode, this API may be called immediately after the [NU_I2C_Master_Read](#) API has completed successfully. In that case, I2C_Indicating_Device and Rx_Length should be same as the parameters passed to NU_I2C_Master_Read.

Example

```
/* This example demonstrates the data receiving of Nucleus I2C. */

/* Declare the required variables. */
STATUS status;

/* Get the data from the input buffer. */
status = NU_I2C_Receive_Data(I2C_Dev_Handle, &Rx_Data[0], Rx_Length);

/* At this point status indicates if the specified number of data bytes
   were retrieved from input buffer successfully. */
```

Related Topics

[I2C Common API Functions](#)

[NU_I2C_Master_Read](#)

NU_I2C_Slave_Get_Address

This API function can be used to learn the details of the local slave address of a node that provides the functionality of an I2C slave.

Usage

```
STATUS NU_ I2C_Slave_Get_Address (I2C_HANDLE i2c_dev,  
                                UINT16      *address,  
                                UINT8       *address_type)
```

Arguments

- **i2c_dev**
Handle to the initialized device.
- **address**
Location where slave address will be returned.
- **address_type**
Location where slave address type (7-bit/10-bit) will be returned. This can be one of the following settings:

```
I2C_7BIT_ADDRESS  
I2C_10BIT_ADDRESS
```

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_INVALID_HANDLE**
Nucleus I2C device handle is not valid.
- **I2C_NODE_NOT_SLAVE**
The node does not provide the functionality of a slave.
- **I2C_INVALID_PARAM_POINTER**
Given data pointer is null.

Example

```
/* This example demonstrates how to determine the local slave address of  
the specified device of Nucleus I2C. */  
  
/* Declare the required variables. */  
STATUS status;  
UINT16 slave_address;  
UINT8 slave_address_type;  
  
/* Get the slave address of the specified local device. */  
status = NU_ I2C_Slave_Get_Address(I2C_Dev_Handle, &slave_address,  
                                   &slave_address_type);
```

```
/* At this point status indicates if the address information was returned  
successfully in address and address_type. */
```

Related Topics

[I2C Common API Functions](#)

[NU_I2C_Master_Set_Slave_Address](#)

NU_I2C_Slave_Set_Address

This API function allows you to set the local slave address of a node that provides the functionality of an I2C slave.

Usage

```
STATUS NU_I2C_Slave_Set_Address (I2C_HANDLE i2c_dev,  
                                UINT16      slave_address,  
                                UINT8       address_type)
```

Arguments

- **i2c_dev**
Handle to the initialized device.
- **slave_address**
Address of the node as a slave.
- **address_type**
Type of the slave address. Possible values for 7bit/10bit are:
 I2C_7BIT_ADDRESS
 I2C_10BIT_ADDRESS

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_INVALID_HANDLE**
Nucleus I2C device handle is not valid.
- **I2C_NODE_NOT_SLAVE**
The node does not provide the functionality of a slave.
- **I2C_INVALID_ADDRESS_TYPE**
Type of the slave address is neither 7-bit nor 10-bit.
- **I2C_INVALID_SLAVE_ADDRESS**
Address of the slave node is not valid.

Example

```
/* This example demonstrates how to set the local slave address of the  
specified device of Nucleus I2C. */  
  
/* Declare the required variables. */  
STATUS status;  
UINT16 slave_address      = 0x07;  
UINT8  slave_address_type = I2C_7BIT_ADDRESS;  
  
/* Get the slave address of the specified local device. */
```

```
status = NU_I2C_Slave_Set_Address(I2C_Dev_Handle, slave_address,  
                                  slave_address_type);  
  
/* At this point status indicates if the address was set successfully. */
```

Related Topics

[I2C Common API Functions](#)

[NU_I2C_Slave_Get_Address](#)

I2C Automatic API Functions

This set of API provides you with a single interface to execute a complete distinct transfer on I2C bus. You only need to give the necessary data to perform an I2C operation (for example, read/write) and you will be informed of the end result of the full transfer operation. You will not need to acquire the details of the micro events at byte level as defined by I2C protocol.

All the master transmission/reception APIs in this section support interrupt-driven as well as polling mode.

- [NU_I2C_Master_Config_HW_Master](#)
- [NU_I2C_Master_Multi_Transfer](#)
- [NU_I2C_Master_Read](#)
- [NU_I2C_Master_Set_Slave_Address](#)
- [NU_I2C_Master_Write](#)
- [NU_I2C_Slave_Response_To_Read](#)

Related Topics

[I2C Function Reference](#)

NU_I2C_Master_Config_HW_Master

This API function configures the hardware master for the slave address to which it will be writing the data.

Usage

```
STATUS NU_I2C_Master_Config_HW_Master (I2C_HANDLE i2c_dev,  
                                       UINT8      hw_master_address,  
                                       UINT8      hw_master_slave)
```

Arguments

- **i2c_dev**
Handle to the initialized device. This must be used for further interaction with the device.
- **hw_master_address**
Address of the hardware master.
- **hw_master_slave**
Address of the slave to which hardware master will be writing the data.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_INVALID_HANDLE**
Nucleus I2C device handle is not valid.
- **I2C_ADDRESS_TX_FAILED**
Master could not address the slave node.
- **I2C_BUS_BUSY**
I2C bus is busy.
- **I2C_DATA_TRANSMISSION_FAILED**
Data reception failed.
- **I2C_INVALID_DRIVER_MODE**
Driver mode is not set to either interrupt driven or polling mode.
- **I2C_SLAVE_NOT_ACKED**
Slave did not acknowledge.
- **I2C_SLAVE_TIME_OUT**
Slave timed out.
- **I2C_TRANSFER_IN_PROGRESS**
An I2C transfer is already in progress.

- I2C_TRANSFER_STOP_FAILED

Transfer on I2C bus could not be stopped.

Example

```
/* This example demonstrates the configuration of hardware master for the
Slave node. */

/* Declare the required variables. */
STATUS status;
UINT8 hw_master_address;
UINT8 hw_master_slave;

/* Set the hardware master address. */
hw_master_address = 0x32;          /* Hardware master address. */
hw_master_slave   = 0x07;          /* Slave address. */

/* Request the hardware configuration for the master node. */
status = NU_I2C_Master_Config_HW_Master(I2C_Dev_Handle,
                                          hw_master_address,
                                          hw_master_slave);

/* At this point status indicates if Nucleus I2C master configured slave
successful. */
```

Related Topics

[I2C Automatic API Functions](#)

NU_I2C_Master_Multi_Transfer

This function allows you to perform reads and writes to multiple nodes using restart instead of stop and start, or it allows you to switch the direction of the data without sending stop and start rather than using restart.

Usage

```
STATUS NU_I2C_Master_Multi_Transfer (I2C_HANDLE    i2c_dev,  
                                     I2C_NODE      *slaves,  
                                     UINT8         *tx_data,  
                                     UINT8         *rx_data,  
                                     UNSIGNED_INT  *lengths,  
                                     UINT8         *rw,  
                                     UINT8         slave_count)
```

Arguments

- **i2c_dev**
A single element specifying the I2C device with which target slaves are connected.
- **slaves**
Array of node structure address details.
- **tx_data**
Pointer to data output buffer containing data for all slaves for which write operation is to be performed.
- **rx_data**
Pointer to data input buffer where data received for the slaves for whom read operation is to be performed, will be returned.
- **lengths**
Array of number of data bytes for each corresponding nodes.
- **rw**
Array of Read/Write for each corresponding node.
- **slave_count**
Number of slaves on the bus involved in multiple transfers.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_INVALID_HANDLE**
Nucleus I2C device handle is not valid.
- **I2C_ACK_TX_FAILED**
Acknowledgment of data failed.

- **I2C_ADDRESS_TX_FAILED**
Master could not address the slave node.
- **I2C_BUFFER_FULL**
Data buffer is full.
- **I2C_BUS_BUSY**
I2C bus is busy.
- **I2C_DATA_RECEPTION_FAILED**
Data reception failed.
- **I2C_DATA_TRANSMISSION_FAILED**
Data transmission failed.
- **I2C_INVALID_ADDRESS_TYPE**
Type of the slave address is neither 7-bit nor 10-bit.
- **I2C_INVALID_DRIVER_MODE**
Driver mode is not set to either interrupt driven or polling mode.
- **I2C_INVALID_OPERATION**
Invalid operation type. It should be I2C_READ or I2C_WRITE.
- **I2C_INVALID_PARAM_POINTER**
Null given instead of a variable pointer.
- **I2C_INVALID_SLAVE_COUNT**
Slave count for multi-transfer is wrong.
- **I2C_NODE_NOT_MASTER**
The node is not master.
- **I2C_SLAVE_NOT_ACKED**
Slave did not acknowledge.
- **I2C_SLAVE_TIME_OUT**
Slave timed out.
- **I2C_TRANSFER_IN_PROGRESS**
An I2C transfer is already in progress.
- **I2C_TRANSFER_STOP_FAILED**
Transfer on I2C bus could not be stopped.
- **I2C_TX_BUFFER_NOT_ENOUGH**
Insufficient transmission buffer memory space.

Example

```
/* This example demonstrates the transfer of data to multiple nodes of
   Nucleus I2C. */

/* Declare the required variables. */
STATUS status;

/* Call the I2C master multi transfer function.*/
status = NU_I2C_Master_Multi_Transfer(I2C_Dev_Handle, slaves, &data,
                                       lengths, rw, slave_count);

/* At this point status indicates if Nucleus I2C master transfers data to
   multiple slaves were successful. */
```

Related Topics

[I2C Automatic API Functions](#)

NU_I2C_Master_Read

This function provides you with a single interface from which to read data from a slave. Concurrent calls of this API may be made which will be executed in sequence in the physical network.

Usage

```
STATUS NU_I2C_Master_Read (I2C_HANDLE    i2c_dev,  
                           I2C_NODE      slave,  
                           UINT8         *data,  
                           UNSIGNED_INT  length)
```

Arguments

- **i2c_dev**
Handle to the initialized device. This must be used for further interaction with the device.
- **slave**
Address of the slave.
- **data**
Data pointer where data will be returned.
- **length**
Length of data in number of data bytes.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_INVALID_HANDLE**
Nucleus I2C device handle is not valid.
- **I2C_ADDRESS_TX_FAILED**
Master could not address the slave node.
- **I2C_BUS_BUSY**
I2C bus is busy.
- **I2C_DATA_RECEPTION_FAILED**
Data reception failed.
- **I2C_INVALID_ADDRESS_TYPE**
Type of the slave address is neither 7-bit nor 10-bit.
- **I2C_INVALID_DRIVER_MODE**
Driver mode is not set to either interrupt driven or polling mode.

- **I2C_INVALID_PARAM_POINTER**
Null given instead of a variable pointer.
- **I2C_INVALID_RX_DATA_LENGTH**
Invalid data length to put in the transmission buffer.
- **I2C_INVALID_SLAVE_ADDRESS**
Address of the slave node is not valid.
- **I2C_NODE_NOT_MASTER**
The node is not master.
- **I2C_RX_BUFFER_NOT_ENOUGH**
Insufficient receive buffer memory space.
- **I2C_SLAVE_NOT_ACKED**
Slave did not acknowledge.
- **I2C_SLAVE_TIME_OUT**
Slave timed out.
- **I2C_TRANSFER_IN_PROGRESS**
An I2C transfer is already in progress.
- **I2C_TRANSFER_STOP_FAILED**
Transfer on I2C bus could not be stopped.

Example

```
/* This example demonstrates the data reading by master node of Nucleus
I2C. */

/* Declare the required variables. */
STATUS status;

/* Call the I2C master read function. */
status = NU_I2C_Master_Read (i2c_dev, slave_address, length);

/* At this point status indicates if Nucleus I2C master read data was
successful. */
```

Related Topics

[NU_I2C_Slave_Response_To_Read](#)

[I2C Automatic API Functions](#)

NU_I2C_Master_Set_Slave_Address

This API directs the remote slaves to write their programmable part of the slave address by hardware.

Usage

```
STATUS NU_I2C_Master_Set_Slave_Address (I2C_HANDLE i2c_dev,  
                                         BOOLEAN    reset)
```

Arguments

- **i2c_dev**
Handle to the initialized device. This must be used for further interaction with the device.
- **reset**
Whether the slave should also reset along with address change.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_INVALID_HANDLE**
Nucleus I2C device handle is not valid.
- **I2C_ADDRESS_TX_FAILED**
Master could not address the slave node.
- **I2C_BUS_BUSY**
I2C bus is busy.
- **I2C_DATA_TRANSMISSION_FAILED**
Data reception failed.
- **I2C_INVALID_DRIVER_MODE**
Driver mode is not set to either interrupt driven or polling mode.
- **I2C_SLAVE_NOT_ACKED**
Slave did not acknowledge.
- **I2C_SLAVE_TIME_OUT**
Slave timed out.
- **I2C_TRANSFER_IN_PROGRESS**
An I2C transfer is already in progress.
- **I2C_TRANSFER_STOP_FAILED**
Transfer on I2C bus could not be stopped.

Example

```
/* This example demonstrates the address setting for the slave node. */  
  
/* Declare the required variables. */  
STATUS status;  
  
/* Call the Nu_I2C_Master_Set_Slave_Address API. */  
status = NU_I2C_Master_Set_Slave_Address(I2C_Dev_Handle, I2C_FALSE);  
  
/* At this point status indicates if Nucleus I2C sets slave address for  
the appropriate node. */
```

Related Topics

[I2C Automatic API Functions](#)

NU_I2C_Master_Write

This function writes any amount of data up to a certain limit to a specified slave. The API may be called from various threads and the calls may be executed sequentially on the physical network.

Usage

```
STATUS NU_I2C_Master_Write (I2C_HANDLE    i2c_dev,  
                             I2C_NODE      slave,  
                             UINT8         *data,  
                             UNSIGNED_INT  length)
```

Arguments

- **i2c_dev**
Handle to the initialized device. This must be used for further interaction with the device.
- **slave**
Address of the slave.
- **data**
Pointer to data.
- **length**
Length of data in number of data bytes.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_INVALID_HANDLE**
Nucleus I2C device handle is not valid.
- **I2C_ADDRESS_TX_FAILED**
Master could not address the slave node.
- **I2C_BUS_BUSY**
I2C bus is busy.
- **I2C_DATA_TRANSMISSION_FAILED**
Data reception failed.
- **I2C_INVALID_ADDRESS_TYPE**
Type of the slave address is neither 7-bit nor 10-bit.
- **I2C_INVALID_DRIVER_MODE**
Driver mode is not set to either interrupt driven or polling mode.

- **I2C_INVALID_PARAM_POINTER**
Null given instead of a variable pointer.
- **I2C_INVALID_SLAVE_ADDRESS**
Address of the slave node is not valid.
- **I2C_INVALID_TX_DATA_LENGTH**
Given length value is zero.
- **I2C_SLAVE_NOT_ACKED**
Slave did not acknowledge.
- **I2C_SLAVE_TIME_OUT**
Slave timed out.
- **I2C_TRANSFER_IN_PROGRESS**
An I2C transfer is already in progress.
- **I2C_TRANSFER_STOP_FAILED**
Transfer on I2C bus could not be stopped.
- **I2C_TX_BUFFER_NOT_ENOUGH**
Insufficient transmission buffer memory space.

Example

```
/* This example demonstrates the data write by master node of Nucleus
I2C.*/

/* Declare the required variables. */
STATUS status;

/* Call the I2C master write function. */
status = NU_I2C_Master_Write (i2c_dev, slave_address, &data, length);

/* At this point status indicates if Nucleus I2C master write data was
successful. */
```

Related Topics

[I2C Automatic API Functions](#)

NU_I2C_Slave_Response_To_Read

This API function allows you to respond to the read command from the master.

Usage

```
STATUS NU_I2C_Slave_Response_To_Read (I2C_HANDLE    i2c_dev,
                                      UINT8          *data,
                                      UNSIGNED_INT    length)
```

Arguments

- **i2c_dev**
Nucleus I2C device handle.
- **data**
Data pointer.
- **length**
Number of data bytes in the data.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_INVALID_HANDLE**
Nucleus I2C device handle is not valid.
- **I2C_INVALID_PARAM_POINTER**
Given data pointer is null.
- **I2C_INVALID_DRIVER_MODE**
Driver mode is not set to interrupt driver.
- **I2C_INVALID_RX_DATA_LENGTH**
Invalid data length to put in the receive buffer.
- **I2C_TX_BUFFER_NOT_ENOUGH**
Insufficient transmission buffer memory space.

Example

```
/* This example demonstrates the slave response to read command issued by
the master. */

/* Declare the required variables. */
STATUS      status;
UINT8       data;
UNSIGNED_INT length;

/* Specify the length of data. */
length = 4;          /* Length is specified in data bytes. */
```

```
/* Request the slave response against read command.*/  
status = NU_I2C_Slave_Response_To_Read(I2C_Dev_Handle, &data, length);  
  
/* At this point status indicates if Nucleus I2C master command to read  
the slave data was successful. */
```

Related Topics

[NU_I2C_Master_Read](#)

[I2C Automatic API Functions](#)

I2C Fine Control API Functions

This set of API provides you with full control on the I2C protocol behavior. You are informed of all the events as defined by the protocol, and you are responsible for making sure that the protocol is followed correctly. These are never necessary to be used by an application. They actually provide a uniform interface to the underlying hardware thus enabling you to develop an application without knowing the details of the hardware but with full control on the protocol behavior. It is recommended that these API are used in conjunction with polling mode.

- [NU_I2C_Check_Ack](#)
- [NU_I2C_Ioctl_Driver](#)
- [NU_I2C_Master_Free_Bus](#)
- [NU_I2C_Master_Read_Byte](#)
- [NU_I2C_Master_Restart](#)
- [NU_I2C_Master_Start_Transfer](#)
- [NU_I2C_Master_Stop_Transfer](#)
- [NU_I2C_Master_Write_Byte](#)
- [NU_I2C_Send_Ack](#)

Related Topics

[I2C Function Reference](#)

NU_I2C_Check_Ack

This API function can be used to poll for the acknowledgement of the reception of address or data.

Usage

```
STATUS NU_I2C_Check_Ack (I2C_HANDLE i2c_dev)
```

Arguments

- **i2c_dev**
Handle to the initialized device. This must be used for further interaction with the device.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_INVALID_HANDLE**
Nucleus I2C device handle is not valid.
- **I2C_NO_RESTART_SIGNAL**
RESTART signal not received.
- **I2C_NO_STOP_SIGNAL**
STOP signal not received.
- **I2C_POLLING_ABORTED**
Polling for acknowledgment aborted.
- **I2C_SLAVE_NOT_ACKED**
Slave did not acknowledge.

Example

```
/* This example demonstrates the acknowledgement checking for slave  
address matching or the reception/transmission of data byte. */  
  
/* Declare the required variables. */  
STATUS status;  
  
/* Call the service to check the acknowledgement. */  
status = NU_I2C_Check_Ack(I2C_Dev_Handle);  
  
/* At this point status indicates if Nucleus I2C acknowledgement check was  
successful. */
```

Related Topics

[NU_I2C_Master_Read_Byte](#)

[NU_I2C_Master_Restart](#)

[NU_I2C_Master_Start_Transfer](#)
[I2C Fine Control API Functions](#)

[NU_I2C_Master_Stop_Transfer](#)

NU_I2C_ioctl_Driver

This API function provides an interface to the `i2c_driver_ioctl` function which performs I/O operations on the controller.

Usage

```
STATUS NU_I2C_Ioctl_Driver (I2C_HANDLE i2c_dev,  
                           UINT8      operation_code,  
                           VOID        *operation_data)
```

Arguments

- `i2c_dev`
Handle to the initialized device. This must be used for further interaction with the device.
- `operation_code`
Code to perform I/O operation. Possible operation codes are listed as follows:

<code>I2C_CHECK_ADDRESS_ACK</code>	<code>I2C_CHECK_DATA_ACK</code>
<code>I2C_CHECK_NACK</code>	<code>I2C_CHECK_BUS_FREE</code>
<code>I2C_CHECK_DATA</code>	<code>I2C_CHECK_RESTART</code>
<code>I2C_CHECK_STOP</code>	<code>I2C_DISABLE_DEVICE</code>
<code>I2C_DISABLE_INTERRUPT</code>	<code>I2C_ENABLE_DEVICE</code>
<code>I2C_ENABLE_INTERRUPT</code>	<code>I2C_FREE_BUS</code>
<code>I2C_GET_MW_CONFIG_PATH</code>	<code>I2C_SEND_NACK</code>
<code>I2C_SEND_ACK</code>	<code>I2C_SEND_START_ADDRESS</code>
<code>I2C_SEND_RESTART_ADDRESS</code>	<code>I2C_SEND_ADDRESS2</code>
<code>I2C_SEND_DATA</code>	<code>I2C_SEND_STOP</code>
<code>I2C_SET_BAUDRATE</code>	<code>I2C_SET_CONTROL_BLOCK</code>
<code>I2C_SET_SLAVE_ADDRESS</code>	<code>I2C_SET_MASTER_MODE</code>
<code>I2C_SET_NODE_MODE_RX</code>	<code>I2C_SET_NODE_MODE_TX</code>

- `operation_data`
Pointer to the data required to perform the operation or pointer where the data of an operation will be returned.

Return Values

- `NU_SUCCESS`
Service completed successfully.
- `I2C_INVALID_HANDLE`
Nucleus I2C device handle is not valid.

- **I2C_BUS_BUSY**
I2C bus is occupied by some I2C master.
- **I2C_INVALID_BAUDRATE**
Invalid baudrate value for the device.
- **I2C_INVALID_IOCTL_OPERATION**
Unsupported operation requested for I/O control.
- **I2C_INVALID_PARAM_POINTER**
Null given instead of a variable pointer.
- **I2C_SLAVE_NOT_ACKED**
Slave did not acknowledge.

Example

```
/* This example demonstrates the I/O operation (e.g. send data to I2C
   hardware module) for I2C controller. */

/* Declare the required variables. */
STATUS status;
UINT8 operation_data;

/* Send the dummy data. */
operation_data = I2C_DUMMY_DATA;

/* Request the desired I/O operation form the I2C controller. */
status = NU_I2C_Ioctl_Driver(i2c_dev, I2C_SEND_DATA, &operation_data);

/* At this point status indicates if I/O operation on I2C controller
   was successful. */
```

Related Topics

[I2C Fine Control API Functions](#)

NU_I2C_Master_Free_Bus

This API makes the I2C bus free by sending a stop signal.

Usage

```
STATUS NU_I2C_Master_Free_Bus (I2C_HANDLE i2c_dev)
```

Arguments

- **i2c_dev**
Handle to the initialized device. This must be used for further interaction with the device.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_INVALID_HANDLE**
Nucleus I2C device handle is not valid.
- **I2C_NODE_NOT_MASTER**
The specified device doesn't support master functionality.
- **I2C_TRANSFER_STOP_FAILED**
Transfer on I2C bus could not be stopped.

Example

```
/* This example demonstrates the bus check for free by master node of  
Nucleus I2C. */  
  
/* Declare the required variables. */  
STATUS status;  
  
/* Call the I2C master check for free bus function.*/  
status = NU_I2C_Master_Free_Bus(i2c_dev);  
  
/* At this point status indicates if Nucleus I2C master bus check for free  
was successful. */
```

Related Topics

[NU_I2C_Master_Stop_Transfer](#)

[I2C Fine Control API Functions](#)

NU_I2C_Master_Read_Byte

This API reads the data from the local buffer.

Usage

```
STATUS NU_I2C_Master_Read_Byte (I2C_HANDLE i2c_dev,  
                                UINT8      *data)
```

Arguments

- **i2c_dev**
Handle to the initialized device. This must be used for further interaction with the device.
- **data**
Pointer to data.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_INVALID_HANDLE**
Nucleus I2C device handle is invalid.
- **I2C_DATA_RX_FAILED**
Data reception failed.

Example

```
/* This example demonstrates the reading data from local buffer by master  
node of Nucleus I2C. */  
  
/* Declare the required variables. */  
STATUS status;  
UINT8  data;  
  
/* Call the I2C master read byte function.*/  
status = NU_I2C_Master_Read_Byte(I2C_Dev_Handle, &data);  
  
/* At this point status indicates if Nucleus I2C master read data byte was  
successful. */
```

Related Topics

[NU_I2C_Check_Ack](#)

[NU_I2C_Master_Stop_Transfer](#)

[NU_I2C_Send_Ack](#)

[NU_I2C_Master_Restart](#)

[NU_I2C_Master_Write_Byte](#)

[I2C Fine Control API Functions](#)

NU_I2C_Master_Restart

This API transmits the address of the slave on the I2C bus for initializing a read/write re-transfer operation without terminating the established connection.

Usage

```
STATUS NU_I2C_Master_Restart (I2C_HANDLE i2c_dev,  
                             I2C_NODE   slave,  
                             UINT8      rw)
```

Arguments

- **i2c_dev**
Handle to the initialized device. This must be used for further interaction with the device.
- **slave**
Slave node information.
- **rw**
Read/Write.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_INVALID_HANDLE**
Nucleus I2C device handle is not valid.
- **I2C_ADDRESS_TX_FAILED**
Master could not address the slave node perhaps due to the reason that it lost arbitration.
- **I2C_BUS_BUSY**
I2C bus is busy.
- **I2C_INVALID_ADDRESS_TYPE**
Type of the slave address is neither 7-bit nor 10-bit.
- **I2C_NODE_NOT_MASTER**
The node is not master.
- **I2C_SLAVE_NOT_ACKED**
Slave didn't acknowledge.
- **I2C_SLAVE_TIME_OUT**
Slave timed out.
- **I2C_TRANSFER_IN_PROGRESS**
An I2C transfer is already in progress.

Example

```
/* This example demonstrates the restart by master node of Nucleus I2C. */

/* Declare the required variables. */
STATUS    status;
I2C_NODE  slave_address;
UINT8     rw;

/* Configure the slave address details. */
slave_address.i2c_slave_address = 0x07;
slave_address.i2c_address_type  = I2C_7BIT_ADDRESS;

/* Configure the read operation. */
rw = I2C_READ;

/* Call the I2C master restart function.*/
status = NU_I2C_Master_Restart(I2C_Dev_Handle, slave_address, rw);

/* At this point status indicates if Nucleus I2C master restart was
successful. */
```

Related Topics

[NU_I2C_Check_Ack](#)

[NU_I2C_Master_Read_Byte](#)

[NU_I2C_Master_Stop_Transfer](#)

[NU_I2C_Master_Write_Byte](#)

[NU_I2C_Send_Ack](#)

[I2C Fine Control API Functions](#)

NU_I2C_Master_Start_Transfer

This API function can be used to initiate a transfer on the network.

Usage

```
STATUS NU_I2C_Master_Start_Transfer (I2C_HANDLE i2c_dev,  
                                     I2C_NODE   slave,  
                                     UINT8      rw)
```

Arguments

- **i2c_dev**
Handle to the initialized device. This must be used for further interaction with the device.
- **slave**
Node structure containing node address details.
- **rw**
Read/Write bit in the address byte.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_INVALID_HANDLE**
Nucleus I2C device handle is not valid.
- **I2C_ADDRESS_TX_FAILED**
Master could not address the slave node perhaps due to the reason that it lost arbitration.
- **I2C_BUS_BUSY**
I2C bus is busy.
- **I2C_INVALID_ADDRESS_TYPE**
Type of the slave address is neither 7-bit nor 10-bit.
- **I2C_NODE_NOT_MASTER**
The node is not master.
- **I2C_SLAVE_NOT_ACKED**
Slave did not acknowledge.
- **I2C_SLAVE_TIME_OUT**
Slave timed out.
- **I2C_TRANSFER_IN_PROGRESS**
An I2C transfer is already in progress.

Example

```
/* This example demonstrates the transfer initialization Nucleus I2C. */

/* Declare the required variables. */
STATUS    status;
I2C_NODE  slave;
UINT8     rw;

/* Configure the slave address details. */
slave.i2c_slave_address = 0x07;
slave.i2c_address_type  = I2C_7BIT_ADDRESS;

/* Configure the transfer type to transmission. */
rw = I2C_WRITE;

/* Request the transfer initialization. */
status = NU_I2C_Master_Start_Transfer(I2C_Dev_Handle, slave, rw);

/* At this point status indicates if Nucleus I2C send acknowledgement
was successful. */
```

Related Topics

[NU_I2C_Check_Ack](#)

[NU_I2C_Master_Read_Byte](#)

[NU_I2C_Master_Restart](#)

[NU_I2C_Master_Stop_Transfer](#)

[NU_I2C_Master_Write_Byte](#)

[NU_I2C_Send_Ack](#)

[I2C Fine Control API Functions](#)

NU_I2C_Master_Stop_Transfer

This API is responsible for ending a transfer on the network.

Usage

```
STATUS NU_I2C_Master_Stop_Transfer (I2C_HANDLE i2c_dev)
```

Arguments

- **i2c_dev**
Handle to the initialized device. This must be used for further interaction with the device.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_INVALID_HANDLE**
Nucleus I2C device handle is not valid.
- **I2C_NODE_IS_NOT_MASTER**
The node is not master.
- **I2C_TRANSFER_STOP_FAILED**
Transfer on I2C bus could not be stopped.

Example

```
/* This example demonstrates the stop transfer request by master node of  
Nucleus I2C. */  
  
/* Declare the required variables. */  
STATUS status;  
  
/* Call the I2C master stop function. */  
status = NU_I2C_Master_Stop_Transfer (I2C_Dev_Handle);  
  
/* At this point status indicates if stop transfer request by master node  
was successful. */
```

Related Topics

[NU_I2C_Check_Ack](#)

[NU_I2C_Master_Read_Byte](#)

[NU_I2C_Master_Start_Transfer](#)

[NU_I2C_Master_Write_Byte](#)

[NU_I2C_Master_Write_Byte](#)

[I2C Fine Control API Functions](#)

NU_I2C_Master_Write_Byte

This API transmits the data from the slave to the master.

Usage

```
STATUS NU_I2C_Master_Write_Byte (I2C_HANDLE i2c_dev,  
                                UINT8      data)
```

Arguments

- **i2c_dev**
Handle to the initialized device. This must be used for further interaction with the device.
- **data**
Data byte to write.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_INVALID_HANDLE**
Nucleus I2C device handle is invalid.
- **I2C_DATA_TX_FAILED**
Data transmission failed.

Example

```
/* This example demonstrates the write byte for master of Nucleus I2C. */  
  
/* Declare the required variables. */  
STATUS status;  
  
/* Call the I2C master write byte function.*/  
status = NU_I2C_Master_Write_Byte(i2c_devs, data);  
  
/* At this point status indicates if Nucleus I2C master write byte was  
successful. */
```

Related Topics

[NU_I2C_Check_Ack](#)

[NU_I2C_Master_Restart](#)

[NU_I2C_Master_Start_Transfer](#)

[NU_I2C_Master_Stop_Transfer](#)

[NU_I2C_Master_Read_Byte](#)

[NU_I2C_Send_Ack](#)

[I2C Fine Control API Functions](#)

NU_I2C_Send_Ack

This API function can be used to transmit acknowledgement of data or address on the bus.

Usage

```
STATUS NU_I2C_Send_Ack (I2C_HANDLE i2c_dev)
```

Arguments

- **i2c_dev**
Handle to the initialized device. This must be used for further interaction with the device.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **I2C_INVALID_HANDLE**
Nucleus I2C device handle is not valid.
- **I2C_ACK_TX_FAILED**
Acknowledgment transmission failure.

Example

```
/* This example demonstrates the send acknowledgement request. */  
  
/* Declare the required variables. */  
STATUS status;  
  
/* Call the I2C send acknowledgement function. */  
status = NU_I2C_Send_Ack(I2C_Dev_Handle);  
  
/* At this point status indicates if acknowledgement was successfully  
sent. */
```

Related Topics

[NU_I2C_Check_Ack](#)

[NU_I2C_Master_Start_Transfer](#)

[NU_I2C_Master_Read_Byte](#)

[I2C Fine Control API Functions](#)

[NU_I2C_Master_Restart](#)

[NU_I2C_Master_Stop_Transfer](#)

[NU_I2C_Master_Write_Byte](#)

I2C Configuration Options

This section describes the configuration options available for Nucleus I2C. These options are available in `<install_root>/nucleus/os/include/connectivity/i2c_cfg.h`. [Table 3-1](#) lists the default configuration for Nucleus I2C.

Table 3-1. Nucleus I2C Configuration Options

Default	Default Value	Meaning
NU_I2C_ERROR_CHECKING	1	To enable/disable the error checking for the parameters.
NU_I2C_SUPPORT_POLLING_MODE	1	Nucleus I2C provides the code base to support polling mode for any device initialized in polling mode.
NU_I2C_ACK_WAIT	100	100 timer ticks.
NU_I2C_NODE_TYPE	I2C_MASTER_SLAVE	Node has the functionality of I2C master as well as slave.
I2C_DEFAULT_BAUDRATE	100	Default values for the corresponding defines.
I2C_DEFAULT_SLAVE_ADDRESS	0x22	
I2C_DEFAULT_ADDRESS_TYPE	I2C_7BIT_ADDRESS	
I2C_DEFAULT_NODE_TYPE	I2C_MASTER_SLAVE	

Note



If you change any configuration options, you must rebuild Nucleus I2C, Nucleus I2C Driver and Nucleus I2C demonstration.

Acknowledgment Wait

This option defines the number of system clock units to wait before the master finishes waiting for an acknowledgment from the slave. It allows you to configure the time period the master should wait before aborting the wait for an acknowledgment from the slave.

This option is valid for I2C master devices only that are operating in polling mode.

The default value of this option is one.

```
#define NU_I2C_ACK_WAIT 1
```

Slave Address of the Node

Nucleus I2C provides the facility of an I2C slave that can operate in interrupt-driven mode. The following defines configure the default address settings for the slave. These may be changed to assign a new address to the slave. Currently, only the slave address can be changed, but the slave address type should not be changed.

```
#define I2C_DEFAULT_SLAVE_ADDRESS 0x22
#define I2C_DEFAULT_ADDRESS_TYPE I2C_7BIT_ADDRESS
```

I2C Baud Rate

This option allows you to configure the baud rate for Nucleus I2C that the Nucleus I2C master will use for communication on the bus. The value is configurable in units of kbps. Exact values supported by a particular Nucleus I2C port depend upon the Nucleus I2C hardware driver. The default value for the baud rate is 100 kbps and can be configured using the following option:

```
#define I2C_DEFAULT_BAUDRATE 100
```

I2C Error Checking

Error checking can be enabled/disabled by changing the value of this define. Error checking is performed on the parameters of the API functions of Nucleus I2C. By default, error checking is enabled, but it can be disabled by setting the value for this define to zero.

```
#define NU_I2C_ERROR_CHECKING 1
```

I2C Device Selection

Each target might have several I2C controllers. Each controller is assigned a unique label. This label is passed in the NU_I2C_Open function to select which controller. In case the target has only one I2C controller, a NULL label will indicate that the only, default controller will be selected.

Node Type

This option allows you to configure the default behavior of Nucleus I2C. A Nucleus I2C node may act as an I2C slave only, an I2C master only, or both as an I2C master and I2C slave as defined respectively by the following defines:

- I2C_SLAVE_NODE
- I2C_MASTER_NODE

- I2C_MASTER_SLAVE

By default, Nucleus I2C is configured to support the functionality of both the master and the slave as per the following setting:

```
#define NU_I2C_NODE_TYPE I2C_MASTER_SLAVE
```

Support for Polling Mode Selection

This option provides polling support in Nucleus I2C. If none of the device uses polling, setting this option to zero helps to reduce the code size.

Setting this define to one does not mean that the devices will start operating in polling mode. It only enables Nucleus I2C to support the drives running in polling mode. The actual operating mode of the driver should be specified at the time of initialization.

The default value of this option is zero, which means polling mode will not be available for Nucleus I2C.

```
#define NU_I2C_SUPPORT_POLLING_MODE 0
```

I2C Error Codes

Table 3-2 describes the error codes returned to the application by Nucleus I2C.

Table 3-2. Nucleus I2C Error Codes

Symbol	Value	Description
NU_SUCCESS	0	Successful completion.
I2C_OS_ERROR	1	Error occurred in underlying operating system kernel.
I2C_NULL_GIVEN_FOR_MEM_POOL	2	Memory pointer was not set properly and was pointing to null.
I2C_NO_ACTIVE_NOTIFICATION	3	No active notification for I2C notification handler.
I2C_NULL_GIVEN_FOR_INIT	4	Initialization pointer is null.
I2C_INVALID_SLAVE_ADDRESS	11	Slave address is not valid.
I2C_SLAVE_NOT_ACKED	12	Slave did not acknowledge.
I2C_BUS_BUSY	13	I2C bus is busy.
I2C_TRANSFER_INIT_FAILED	14	Transfer on I2C bus could not be initialized.
I2C_ACK_TX_FAILED	15	Acknowledgment transmission failure.

Table 3-2. Nucleus I2C Error Codes (cont.)

Symbol	Value	Description
I2C_INVALID_ADDRESS_TYPE	16	Type of the slave address is neither 7-bit nor 10-bit.
I2C_7BIT_ADDRESS_TYPE	17	7-bit address type.
I2C_10BIT_ADDRESS_TYPE	18	10-bit address type.
I2C_TRANSFER_STOP_FAILED	19	Transfer on I2C bus could not be stopped.
I2C_NO_STOP_SIGNAL	20	STOP signal not received.
I2C_NO_RESTART_SIGNAL	21	RESTART signal not received.
I2C_ADDRESS_TX_FAILED	22	Master could not address the slave node perhaps due to the reason that it lost arbitration.
I2C_TRANSFER_IN_PROGRESS	23	An I2C transfer is already in progress.
I2C_INVALID_NODE_TYPE	24	Node is not master/slave.
I2C_NODE_NOT_SLAVE	25	The node is not slave.
I2C_NODE_NOT_MASTER	26	The node is not master.
I2C_INVALID_OPERATION	27	Invalid operation type. It should be I2C_READ or I2C_WRITE.
I2C_INVALID_PORT_ID	31	Invalid I2C driver port.
I2C_INVALID_DEVICE_ID	32	Invalid controller ID.
I2C_INVALID_HANDLE	33	Nucleus I2C device handle is invalid.
I2C_INVALID_HANDLE_POINTER	34	Specified pointer is not valid.
I2C_SHUTDOWN_ERROR	35	I2C controller could not be closed.
I2C_DEV_ALREADY_INIT	36	Nucleus I2C device is already initialized.
I2C_DEV_NOT_INIT	37	Nucleus I2C device is not initialized.
I2C_DEV_SLEEPING	38	Nucleus I2C device is currently in sleep mode.
I2C_DRIVER_REGISTER_FAILED	39	Hardware driver registration with Nucleus I2C failed.
I2C_DRIVER_INIT_FAILED	40	The device initialization failed.
I2C_INVALID_DRIVER_MODE	41	Driver mode is not set to either interrupt driven or polling mode.
I2C_SLAVE_TIME_OUT	42	Slave timed out.
I2C_INVALID_VECTOR		Invalid vector ID.

Table 3-2. Nucleus I2C Error Codes (cont.)

Symbol	Value	Description
I2C_INVALID_PARAM_POINTER	43	Null given instead of a variable pointer.
I2C_INVALID_POINTER	44	Invalid parameter pointer.
I2C_INVALID_SLAVE_COUNT	45	Slave count for multi-transfer is wrong.
I2C_DATA_RECEPTION_FAILED	46	All data could not be received properly.
I2C_DATA_TRANSMISSION_FAILED	47	All data could not be transmitted properly.
I2C_INVALID_IOCTL_OPERATION	48	Unsupported operation requested from ioctl (driver I/O control) routine.
I2C_DEVICE_IN_USE	49	Device is being used by other user.
I2C_BUFFER_EMPTY	51	Data buffer is empty.
I2C_BUFFER_FULL	52	Data buffer is full.
I2C_BUFFER_NOT_EMPTY	53	Buffer still has data.
I2C_BUFFER_HAS_LESS_DATA	54	More data was requested from buffer than available.
I2C_INVALID_TX_BUFFER_SIZE	55	Transmission buffer size if zero.
I2C_INVALID_RX_BUFFER_SIZE	56	Transmission buffer size if zero.
I2C_INVALID_TX_DATA_LENGTH	57	Invalid data length to put in the transmission buffer.
I2C_INVALID_RX_DATA_LENGTH	58	Invalid data length to put in the transmission buffer.
I2C_TX_BUFFER_NOT_ENOUGH	59	Insufficient transmission buffer memory space.
I2C_RX_BUFFER_NOT_ENOUGH	60	Insufficient receive buffer space.
I2C_GENERAL_HARDWARE_ERROR	100	An undocumented error occurred in I2C driver.
I2C_INVALID_BAUDRATE	101	The specified baud rate is not supported.
I2C_TRANSMISSION_ABORTED	102	Message transmission aborted.
I2C_POLLING_ABORTED	103	Polling aborted.

Chapter 4

Serial Peripheral Interface (SPI)

This chapter describes the Nucleus SPI software module and the requirements for using it.

Caution



To guarantee future support and compatibility for your application, use only documented Nucleus interfaces, structures, macros, and so on.

SPI Module Overview

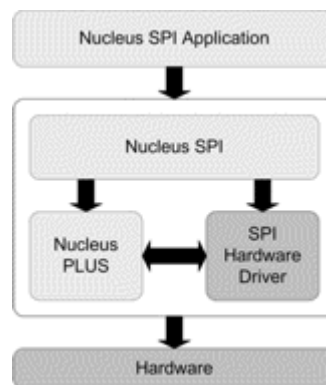
Nucleus SPI is an implementation of the SPI protocol designed for an embedded application. The implementation provides transmission, reception, and duplex transfer functions and a set of functions that control different SPI transfer attributes such as baud rate, transfer size, and so on. You can choose to be notified whenever a data transfer completes.

To accommodate custom applications, Nucleus SPI supports three modes as follows:

- Polling mode
- Interrupt-driven mode
- Interrupt-driven with user buffering mode

Figure 4-1 shows the basic components in a Nucleus SPI based system

Figure 4-1. Nucleus SPI Based System

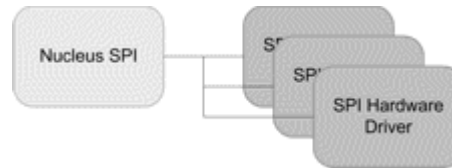


Nucleus SPI supports multiple SPI interfaces on a microprocessor. For example, if two SPI interfaces are available in a processor, both can be used in slave/master configuration. Different

types of SPI modules can also be used at the same time. This is especially useful for FPGA-based systems.

Figure 4-2 illustrates the interaction between Nucleus SPI and multiple SPI interfaces.

Figure 4-2. Interaction of Nucleus SPI with Multiple SPI Interfaces



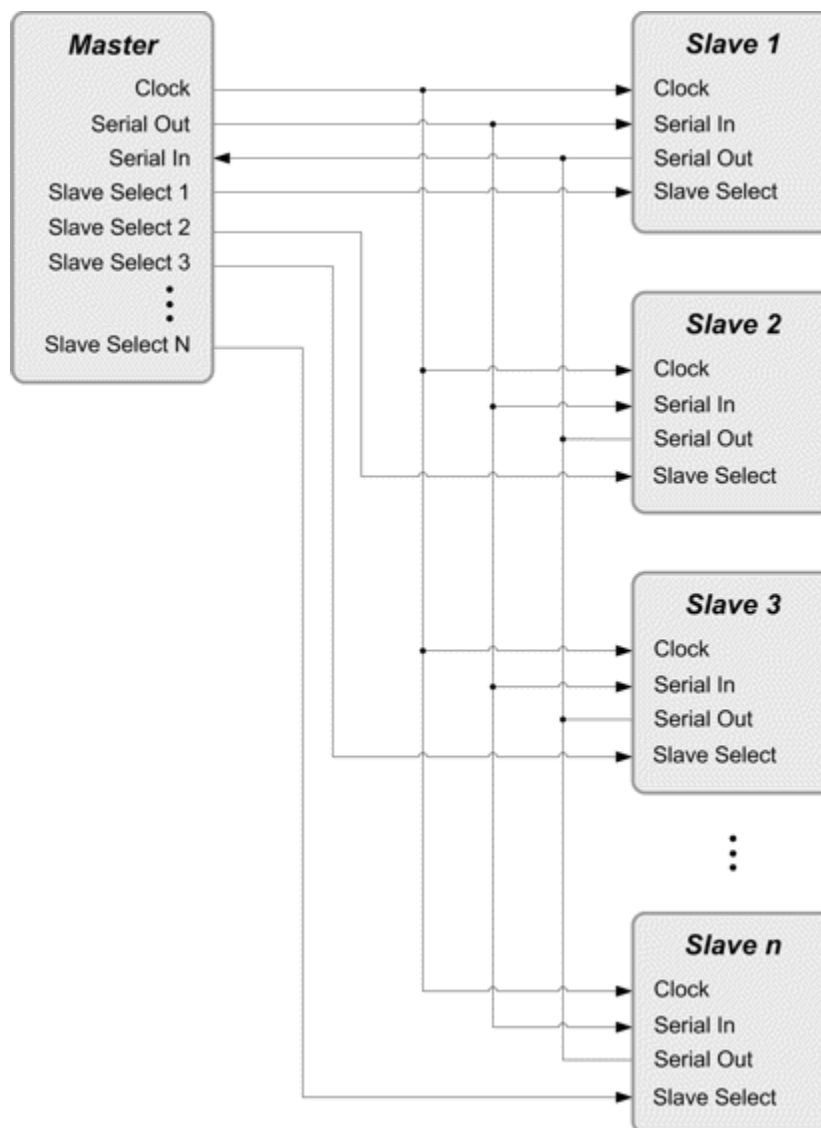
Standard SPI Model

SPI is a synchronous full duplex serial communication interface that transfers information between ICs and PCBs at shorter distances. SPI employs the Master/Slave communication model as shown in Figure 4-3. One wire carries data from master to slaves, a second wire carries data from the slaves to the master, and a third wire carries the clock signal from the master to the slave. SPI uses hardware addressing of slave devices. The master provides each slave with a Slave Select signal. To address a slave, the master asserts its Slave Select signal. Hence, from the slaves' point of view this is a four-wire bus. From the master's point of view, it is a 3+N wire bus, N being the number of slaves.

All transfers are initiated and controlled by the SPI master. Moreover, all transfers are fully duplex. SPI acts like a programmable length shift register. Data is shifted out by one device and the data from the other device is shifted in at the same time.

Initially, the SPI bus only supported 8-bit byte transfers, however, now, devices support variable transfer sizes (4-bit, 7-bit, 16-bit, and so on). The transfers can be LSB (least significant bit) first or MSB (most significant bit) first. There is no limit for the maximum or minimum data transfer rates except the device capability and the distance. Devices supporting several tens of MHz are available.

SPI provides a raw serial communication medium. There is no acknowledgement, so neither the master nor the slave knows whether the other end received the data transmitted.

Figure 4-3. SPI Devices Interconnection

Nucleus SPI offers a SPI Driver and Lightweight SPI.

The SPI driver is an add-on that controls the specific bus to interface for Nucleus SPI. For more information on the SPI Driver, refer to the [SPI Driver](#).

The Lightweight SPI provides an API that translates SPI driver calls. For more information Lightweight SPI Lightweight, refer to [Lightweight SPI](#).

SPI Driver

Nucleus SPI driver is an add-on that controls the specific bus to interface for Nucleus SPI.

SPI Configuration Options

This section describes the configuration options available for Nucleus SPI. Depending on the option, they are configured in one of the following files:

- [Configuration in SPI Metadata File](#)
- [Configuration in nu_connectivity.h File](#)

Configuration in SPI Metadata File

The metadata file for SPI software is located in `/nucleus/os/connectivity/spi`. It contains two configurable entities: `queue_size` and `buffer_size`. [Table 4-1](#) shows their default values.

Table 4-1. Nucleus SPI Metadata Configuration

Entity	Default Value	Meaning of Default Value
<code>queue_size</code>	16	Number of elements in Request Queue is 16
<code>buffer_size</code>	16	Size of buffer in bytes is 16

The entities shown in [Table 4-1](#) can be optionally configured on a per-device basis in the platform file of the target.

Configuration in nu_connectivity.h File

The Nucleus SPI options are available in `/os/include/connectivity/nu_connectivity.h`. [Table 4-2](#) lists the default configuration for Nucleus SPI.

Table 4-2. Nucleus SPI Configuration

Define	Default Value	Meaning of Default Value
<code>NU_SPI_ERROR_CHECKING</code>	1	Nucleus SPI performs error checking for the API parameters.
<code>NU_SPI_SUPPORT_POLLING_MODE</code>	0	The support for the polling mode is excluded from compilation. No SPI device can be initialized in polling mode.
<code>NU_SPI_USER_BUFFERING_ONLY</code>	0	The support for internal buffering performed by Nucleus SPI in the interrupt driven mode is excluded from compilation. Any SPI device to be initialized in interrupt-driven mode must use user buffering.

Table 4-2. Nucleus SPI Configuration (cont.)

Define	Default Value	Meaning of Default Value
NU_SPI_SPEED_COPY	0	Fast buffer copy from user buffers to the internal buffers of Nucleus SPI is disabled.
SPI_MAX_DEV_COUNT	2	Maximum number of devices (controllers) being handled by Nucleus SPI.

A brief description of each of these options and its possible values is given in the following section(s).

Note

Nucleus SPI, Nucleus SPI Driver, and the Nucleus SPI demonstration must be built after changing the value of any option in the configuration file.

SPI Error Checking

Error checking can be enabled/disabled by changing the value of this define. Error checking is performed on the parameters of the API functions of Nucleus SPI. By default, error checking is enabled. Error checking can be disabled by setting the value of this define to zero.

```
#define NU_SPI_ERROR_CHECKING 1
```

Support for Polling Mode

This option allows you to enable/disable support for polling mode in Nucleus SPI. If none of the devices use polling mode, setting this option to zero helps to reduce the code size.

Setting this define to one does not mean that the devices will start operating in polling mode. Setting this define to one enables Nucleus SPI to support the drives running in polling mode. The actual operating mode of the driver should be specified at the time of initialization.

The default value of this option is zero, which means that polling mode will not be available for Nucleus SPI.

```
#define NU_SPI_SUPPORT_POLLING_MODE 0
```

Support for Internal Buffering

This option allows you to enable/disable support for internal buffering performed in interrupt driven mode. If all devices use user buffering, setting this option to one helps to reduce the code size.

Setting this define to one means that internal buffering will not be available and user buffering must be used for any SPI device to be initialized in interrupt driven mode.

When this option is set to one, the `NU_SPI_SPEED_COPY` option (see the following example) has no effect.

```
#define NU_SPI_USER_BUFFERING_ONLY 0
```

Optimize Buffer Copy for Speed

This option allows you to enable/disable fast copying from the user buffer to the internal buffer with increased code size. This option is used only in interrupt driven mode when user buffering is not being used. Set this option to one to enable speed optimization.

This option has no effect when user buffering is utilized.

```
#define NU_SPI_SPEED_COPY 0
```

Number of SPI Devices

All memory for device control blocks are statically allocated rather than dynamic to avoid heap memory fragmentation. The number of SPI Devices can be configured here that will allocate the required space for the device control blocks.

```
#define SPI_MAX_DEV_COUNT 2
```

SPI Pin Naming

SPI modules found on different processors follow different naming for the signals/pins they utilize for SPI functionality. This generally does not matter much for clock and slave select lines, however the serial data signals are affected by the way they are named. Though exact naming can differ, these can be divided into two general categories:

- Some SPI modules use a classic scheme. In this scheme, each SPI interface, whether master or slave, has two data pins: MOSI (Master Out Slave In) and MISO (Master In Slave Out). The MOSI pin is an output pin on the SPI master interface and an input on the slave interface. The MISO pin is an input on the master side and output on the slave side. This facilitates the connections; just connect the master MOSI to the slave's MOSI marked pins and the master MISO pin to the slave's MISO pins. The hardware configuration as master or slave automatically makes the appropriate pins input or output.
- Some SPI modules, however, use a different approach. Instead of naming the signals from a master and slave point-of-view, they provide two pins for serial data: serial-out and serial-in. In this scheme, the serial-out of the master must be connected with the serial-in of the slaves and the serial-in of the master with the serial-out of the slaves.

These schemes are similar, however, the difference is that in MOSI/MISO schemes, the pins change their type from input to output or vice versa depending upon the master/slave configuration of the SPI module. On the other hand, in a serial-out/serial-in scheme, the pin type (input or output) does not change whether it is on the slave side or the master side.

If you are testing the same device, once as a master, and then as a slave, in the MOSI/MISO case, you do not need to change your data pin connections. In the case of a serial-out/serial-in scheme, the data pin connections must be swapped when changing from master to slave or vice versa. This assumes that the testing tool uses the MOSI/MISO scheme (for example, AARDVARK I2C/SPI Testing Tool from Total Phase). However, if the testing tool uses a serial-out/serial-in scheme, then the situation is reversed; in this case, the data pin connections must be swapped when the device being tested uses the MOSI/MISO scheme.

SPI Function Reference

This section describes the functions provided by Nucleus SPI.

Nucleus SPI functions can be divided into four categories:

- [Initialization and Shutdown](#)
- [Transfer Attribute Control](#)
- [Data Transfer](#)
- [Miscellaneous](#)

Functions are useful only after [NU_SPI_Start](#) is called and a valid SPI_HANDLE is obtained. After [NU_SPI_Close](#) is called for a device, only [NU_SPI_Start](#) may be called.

Initialization and Shutdown

This section describes the functions intended for initialization and shutdown of SPI devices.

- [NU_SPI_Close](#)
- [NU_SPI_Start](#)

Related Topics

[SPI Function Reference](#)

NU_SPI_Close

This function shuts down the specified SPI device and releases the associated resources.

Usage

```
STATUS NU_SPI_Close (SPI_HANDLE spi_dev)
```

Arguments

- `spi_dev`
Handle to the device to close.

Return Values

- `NU_SUCCESS`
Service completed successfully.
- `SPI_INVALID_HANDLE`
Nucleus SPI device handle is not valid.
- `SPI_OS_ERROR`
Error in the underlying operating system of Nucleus SPI.
- `SPI_SHUTDOWN_ERROR`
SPI controller could not be closed.
- `SPI_DEVICE_IN_USE`
The device is being used by some other user.

Example

```
/* This example demonstrates the shutdown/closing of a Nucleus SPI
   device. */

SPI_HANDLE SPI_Dev_Handle;

/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device
   which has previously been initialized with Nucleus SPI NU_SPI_Start
   service call. */

/* Declare the required variables. */
STATUS status;

/* Close the SPI device. */
status = NU_SPI_Close(SPI_Dev_Handle);

/* At this point status indicates if the Nucleus SPI device was
   successfully closed. */
```

Related Topics

[NU_SPI_Start](#)

[Initialization and Shutdown](#)

NU_SPI_Start

This function initializes the specified SPI device and must be called before using any other Nucleus SPI functions.

Usage

```
STATUS NU_SPI_Start (SPI_HANDLE *spi_dev,  
                    SPI_INIT   *spi_init)
```

Arguments

- **spi_dev**
Pointer to the location where a handle to the initialized device will be returned. This handle must be used for further interaction with the device.
- **spi_init**
Pointer to Nucleus SPI device initialization block structure.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_DEVICE_ERROR**
The device could not be opened.
- **SPI_INVALID_HANDLE_POINTER**
Specified Nucleus SPI device handle pointer is null.
- **SPI_NULL_GIVEN_FOR_INIT**
Initialization structure pointer is null.
- **SPI_NULL_GIVEN_FOR_MEM_POOL**
The specified memory pool pointer is null.
- **SPI_DRIVER_REGISTER_FAILED**
Nucleus SPI driver could not be registered with Nucleus SPI.
- **SPI_INVALID_PORT_ID**
Specified port is not valid.
- **SPI_INVALID_DEVICE_ID**
Specified device is not valid.
- **SPI_INVALID_DRIVER_MODE**
Specified driver mode is not valid.
- **SPI_INVALID_QUEUE_SIZE**
Specified queue size is not valid.

- **SPI_INVALID_BUFFER_SIZE**
Specified buffer size is not valid.
- **SPI_DRIVER_MODE_UNAVAILABLE**
Specified driver mode is not available in current configuration.
- **SPI_OS_ERROR**
Error occurred in the underlying OS.
- **SPI_DEV_ALREADY_INIT**
The specified device has already been initialized.
- **SPI_INVALID_MODE**
Specified SPI mode is not valid.
- **SPI_INVALID_BIT_ORDER**
Specified bit order is invalid.
- **SPI_REGISTRY_ERROR**
Some kind of system registry error occurred.
- **SPI_UNSUPPORTED_BAUD_RATE**
The specified baud rate is not supported.
- **SPI_UNSUPPORTED_MODE**
Specified SPI mode is not supported.
- **SPI_UNSUPPORTED_BIT_ORDER**
Specified bit order is not supported.
- **SPI_UNSUPPORTED_TRANSFER_SIZE**
Specified transfer size is not supported.
- **SPI_MASTER_MODE_NOT_SUPPORTED**
Specified SPI device cannot function as an SPI master.
- **SPI_SLAVE_MODE_NOT_SUPPORTED**
Specified SPI device cannot function as an SPI slave.

Description

This service sets up the data structures for the protocol stack and initializes the specified underlying hardware driver depending upon the configuration you choose. For more information on Nucleus SPI initialization, see [“SPI Error Codes”](#) on page 207.

Note



Polling driver mode is not supported for an SPI slave device.

Example

```
/* This example demonstrates the initialization of a Nucleus SPI
device. */

/* Declare the variable to get the handle of the initialized SPI
device. */
SPI_HANDLE SPI_Dev_Handle;

/* Declare the required variables. */
SPI_INIT spi_init;
STATUS    status;

/* Configure the Nucleus SPI device initialization structure. */

/* This is the SPI controller label */
DV_DEV_LABEL spi_controller = {SPI_DEVICE_LABEL};

spi_init.spi_controller_label = spi_controller;

/* This is the SPI bus chip select # on which the device is connected to
the SPI controller */
spi_init.spi_address = CHIP_SELECT_NUMBER;

/* Initial SPI data transfer attributes. */
spi_init.spi_baud_rate      = 1000;
spi_init.spi_clock_phase    = SPI_CPHA_0;
spi_init.spi_clock_polarity = SPI_CPOL_0;
spi_init.spi_bit_order      = SPI_MSB_FIRST;
spi_init.spi_transfer_size  = 8;

/* Specify application callback routines. */
spi_init.spi_callbacks.spi_transfer_complete = SPI_Transfer_Complete;
spi_init.spi_callbacks.spi_error            = SPI_Error_Handler;

/* Call the Nucleus SPI initialization function. */
status = NU_SPI_Start(&SPI_Dev_Handle, &spi_init);

/* At this point status indicates if Nucleus SPI initialization was
successful. */
```

Related Topics

[NU_SPI_Close](#)

[Initialization and Shutdown](#)

Transfer Attribute Control

This section describes the functions intended for SPI transfer attribute control.

Nucleus SPI supports a separate set of transfer attributes for each slave address of an SPI master device. This allows you to specify different transfer attributes initially. Transfers can then be completed, with any slave, without stopping to switch attributes for a specified slave. Nucleus SPI automatically switches to the transfer attributes associated with the specified slave.

At the time of device initialization, you must specify a default set of transfer attributes. For SPI master devices, the default transfer attributes are associated to all of its slaves. Slave devices must have transfer attributes specified directly. If different slaves need different values for their transfer attributes, those attributes can be set after device initialization using the functions described in the following sections.

This section describes the following SPI transfer attribute control functions.

- [NU_SPI_Master_Get_Configuration](#)
- [NU_SPI_Master_Set_Baud_Rate](#)
- [NU_SPI_Master_Set_Bit_Order](#)
- [NU_SPI_Master_Set_Configuration](#)
- [NU_SPI_Master_Set_SPI_Mode](#)
- [NU_SPI_Master_Set_Transfer_Size](#)
- [NU_SPI_Slave_Get_Configuration](#)
- [NU_SPI_Slave_Set_Bit_Order](#)
- [NU_SPI_Slave_Set_Configuration](#)
- [NU_SPI_Slave_Set_SPI_Mode](#)
- [NU_SPI_Slave_Set_Transfer_Size](#)

Related Topics

[SPI Function Reference](#)

NU_SPI_Master_Get_Configuration

This function gets the transfer attributes for the given slave of the specified SPI master device.

Usage

```
STATUS NU_SPI_Master_Get_Configuration (SPI_HANDLE      spi_dev,  
                                       UINT16          address,  
                                       SPI_TRANSFER_CONFIG *config)
```

Arguments

- **spi_dev**
The specified SPI device.
- **address**
Address of the slave.
- **config**
Pointer to a Nucleus SPI transfer attributes configuration structure which will get the queried transfer attributes information.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
Nucleus SPI device handle is not valid.
- **SPI_INVALID_ADDRESS**
Specified slave address is not valid for the specified SPI device.
- **SPI_INVALID_PARAM_POINTER**
Null given instead of a variable pointer.

Example

```
/* This example demonstrates querying the transfer attributes associated  
   with a particular slave. */  
  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */  
  
/* Declare the required variables. */  
STATUS      status;  
SPI_TRANSFER_CONFIG config;  
  
/* Query the transfer attributes associated with slave 2. */  
status = NU_SPI_Master_Get_Configuration(SPI_Dev_Handle, 2, &config);
```

```
/* At this point status indicates if the transfer attributes query was  
successful. */
```

Related Topics

[NU_SPI_Master_Set_Configuration](#)

[Transfer Attribute Control](#)

NU_SPI_Master_Set_Baud_Rate

This function sets the baud rate for the specified slave of the specified SPI master device.

Usage

```
STATUS NU_SPI_Master_Set_Baud_Rate (SPI_HANDLE spi_dev,  
                                   UINT16      address,  
                                   UINT32      baud_rate)
```

Arguments

- **spi_dev**
The specified SPI device.
- **address**
Address of the slave.
- **baud_rate**
The desired baud rate.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
Nucleus SPI device handle is not valid.
- **SPI_INVALID_ADDRESS**
Specified slave address is not valid for the specified SPI device.
- **SPI_UNSUPPORTED_BAUD_RATE**
The specified baud rate is not supported.

Example

```
/* This example demonstrates setting the baud rate associated  
   with a particular slave. */  
  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */  
  
/* Declare the required variables. */  
STATUS status;  
  
/* Set the baud rate associated with slave 2 to 1Mbps (1000Kbps). */  
status = NU_SPI_Master_Set_Baud_Rate(SPI_Dev_Handle, 2, 1000);  
  
/* At this point status indicates if the specified baud rate was  
   successfully set. */
```

Related Topics

[NU_SPI_Master_Get_Configuration](#)

[NU_SPI_Master_Set_Configuration](#)

[Transfer Attribute Control](#)

NU_SPI_Master_Set_Bit_Order

This function sets the bit order for the specified slave of the specified SPI master device.

Usage

```
STATUS NU_SPI_Master_Set_Bit_Order (SPI_HANDLE spi_dev,  
                                   UINT16      address,  
                                   UINT8       bit_order)
```

Arguments

- **spi_dev**
The specified SPI device.
- **address**
Address of the slave.
- **bit_order**
The desired bit order.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
Nucleus SPI device handle is not valid.
- **SPI_INVALID_ADDRESS**
Specified slave address is not valid for the specified SPI device.
- **SPI_INVALID_BIT_ORDER**
Specified bit order is invalid.
- **SPI_UNSUPPORTED_BIT_ORDER**
Specified bit order is not supported.

Example

```
/* This example demonstrates setting the bit order associated  
   with a particular slave. */  
  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */  
  
/* Declare the required variables. */  
STATUS status;  
  
/* Set the bit order associated with slave 2 to LSB first. */
```



```
status = NU_SPI_Master_Set_Bit_Order(SPI_Dev_Handle, 2, SPI_LSB_FIRST);  
  
/* At this point status indicates if the bit order was successfully set to  
   LSB first. */
```

Related Topics

[NU_SPI_Master_Get_Configuration](#)

[NU_SPI_Master_Set_Configuration](#)

[Transfer Attribute Control](#)

NU_SPI_Master_Set_Configuration

This function sets the specified transfer attributes for the given slave of the specified SPI master device.

Usage

```
STATUS NU_SPI_Master_Set_Configuration (SPI_HANDLE spi_dev,  
                                       UINT16 address,  
                                       SPI_TRANSFER_CONFIG *config)
```

Arguments

- **spi_dev**
The specified SPI device.
- **address**
Address of the slave.
- **config**
SPI configuration structure containing the desired transfer attributes.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
Nucleus SPI device handle is not valid.
- **SPI_INVALID_ADDRESS**
Specified slave address is not valid for the specified SPI device.
- **SPI_INVALID_PARAM_POINTER**
Null given instead of a variable pointer.
- **SPI_INVALID_MODE**
Specified SPI mode is not valid.
- **SPI_INVALID_BIT_ORDER**
Specified bit order is invalid.
- **SPI_UNSUPPORTED_BAUD_RATE**
The specified baud rate is not supported.
- **SPI_UNSUPPORTED_MODE**
Specified SPI mode is not supported.
- **SPI_UNSUPPORTED_BIT_ORDER**
Specified bit order is not supported.

- **SPI_UNSUPPORTED_TRANSFER_SIZE**

Specified transfer size is not supported.

Example

```
/* This example demonstrates setting all transfer attributes associated
   with a particular slave. */

SPI_HANDLE SPI_Dev_Handle;

/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device
   which has previously been initialized with Nucleus SPI NU_SPI_Start
   service call. */

/* Declare the required variables. */
STATUS      status;
SPI_TRANSFER_CONFIG config;

/* Choose the desired transfer attributes. */
config.spi_baud_rate      = 1000;
config.spi_clock_phase    = SPI_CPHA_0;
config.spi_clock_polarity = SPI_CPOL_0;
config.spi_bit_order      = SPI_MSB_FIRST;
config.spi_transfer_size  = 8;

/* Change the transfer attributes associated with slave 2. */
status = NU_SPI_Master_Set_Configuration(SPI_Dev_Handle, 2, &config);

/* At this point status indicates if the transfer attributes were
   successfully changed. */
```

Related Topics

[NU_SPI_Master_Get_Configuration](#)

[NU_SPI_Master_Set_Configuration](#)

[Transfer Attribute Control](#)

NU_SPI_Master_Set_SPI_Mode

This function sets the SPI mode for the specified slave of the specified SPI master device.

Usage

```
STATUS NU_SPI_Master_Set_SPI_Mode (SPI_HANDLE spi_dev,  
                                  UINT16      address,  
                                  UINT8       clock_phase,  
                                  UINT8       clock_polarity)
```

Arguments

- **spi_dev**
The specified SPI device.
- **address**
Address of the slave.
- **clock_phase**
The desired clock phase.
- **clock_polarity**
The desired clock polarity.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
Nucleus SPI device handle is not valid.
- **SPI_INVALID_ADDRESS**
Specified slave address is not valid for the specified SPI device.
- **SPI_INVALID_MODE**
Specified SPI mode is not valid.
- **SPI_UNSUPPORTED_MODE**
Specified SPI mode is not supported.

Example

```
/* This example demonstrates setting the SPI mode associated with a  
   particular slave. */  
  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */
```

```
/* Declare the required variables. */
STATUS          status;

/* Set the SPI mode for slave 2 to SPI mode 1 (CPOL0, CPHA1). */
status = NU_SPI_Master_Set_SPI_Mode(SPI_Dev_Handle, 2, SPI_CPOL_0,
                                     SPI_CPHA_1);

/* At this point status indicates if SPI mode was successfully changed. */
```

Related Topics

[NU_SPI_Master_Get_Configuration](#)

[NU_SPI_Master_Set_Configuration](#)

[Transfer Attribute Control](#)

NU_SPI_Master_Set_Transfer_Size

This function sets the transfer size for the specified slave of the specified SPI master device.

Usage

```
STATUS NU_SPI_Master_Set_Transfer_Size (SPI_HANDLE spi_dev,  
                                       UINT16      address,  
                                       UINT8       transfer_size)
```

Arguments

- **spi_dev**
The specified SPI device.
- **address**
Address of the slave.
- **transfer_size**
The desired transfer size.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
Nucleus SPI device handle is not valid.
- **SPI_INVALID_ADDRESS**
Specified slave address is not valid for the specified SPI device.
- **SPI_UNSUPPORTED_TRANSFER_SIZE**
Specified transfer size is not supported.

Example

```
/* This example demonstrates setting the bit order associated  
   with a particular slave.. */  
  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */  
  
/* Declare the required variables. */  
STATUS status;  
  
/* Set the transfer size associated with slave 2 to 7. */  
status = NU_SPI_Master_Set_SPI_Mode(SPI_Dev_Handle, 2, 7);  
  
/* At this point status indicates if transfer size was successfully  
   changed. */
```

Related Topics

[NU_SPI_Master_Get_Configuration](#)

[NU_SPI_Master_Set_Configuration](#)

[Transfer Attribute Control](#)

NU_SPI_Slave_Get_Configuration

This function gets the transfer attributes for the specified SPI slave device.

Usage

```
STATUS NU_SPI_Slave_Get_Configuration (SPI_HANDLE spi_dev,  
                                       SPI_TRANSFER_CONFIG *config)
```

Arguments

- **spi_dev**
The specified SPI device.
- **config**
Pointer to a SPI configuration structure which will get the queried transfer attributes information.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
Nucleus SPI device handle is not valid.
- **SPI_INVALID_PARAM_POINTER**
Null given instead of a variable pointer.

Example

```
/* This example demonstrates querying the transfer attributes associated  
with an SPI slave device. */  
  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
which has previously been initialized with Nucleus SPI NU_SPI_Start  
service call. */  
  
/* Declare the required variables. */  
STATUS status;  
SPI_TRANSFER_CONFIG config;  
  
/* Query the transfer attributes associated with the device. */  
status = NU_SPI_Slave_Get_Configuration(SPI_Dev_Handle, &config);  
  
/* At this point status indicates if the transfer attributes query was  
successful. */
```

Related Topics

[NU_SPI_Slave_Set_Configuration](#)

[Transfer Attribute Control](#)

NU_SPI_Slave_Set_Bit_Order

This function sets the bit order for the specified SPI slave device.

Usage

```
STATUS NU_SPI_Slave_Set_Bit_Order (SPI_HANDLE spi_dev,  
                                  UINT8      bit_order)
```

Arguments

- **spi_dev**
The specified SPI device.
- **bit_order**
The desired bit order.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
Nucleus SPI device handle is not valid.
- **SPI_INVALID_BIT_ORDER**
Specified bit order is invalid.
- **SPI_UNSUPPORTED_BIT_ORDER**
Specified bit order is not supported.

Example

```
/* This example demonstrates setting the bit order. */  
  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */  
  
/* Declare the required variables. */  
STATUS status;  
  
/* Set the bit order to LSB first. */  
status = NU_SPI_Slave_Set_Bit_Order(SPI_Dev_Handle, SPI_LSB_FIRST);  
  
/* At this point status indicates if the bit order was successfully set to  
   LSB first. */
```

Related Topics

[NU_SPI_Slave_Get_Configuration](#)

[NU_SPI_Slave_Set_Configuration](#)

Transfer Attribute Control

NU_SPI_Slave_Set_Configuration

This function sets the specified transfer attributes for the specified SPI slave device.

Usage

```
STATUS NU_SPI_Slave_Set_Configuration (SPI_HANDLE      spi_dev,  
                                       SPI_TRANSFER_CONFIG *config)
```

Arguments

- `spi_dev`
The specified SPI device.
- `config`
SPI configuration structure containing the desired transfer attributes.

Return Values

- `NU_SUCCESS`
Service completed successfully.
- `SPI_INVALID_HANDLE`
Nucleus SPI device handle is not valid.
- `SPI_INVALID_PARAM_POINTER`
Null given instead of a variable pointer.
- `SPI_INVALID_MODE`
Specified SPI mode is not valid.
- `SPI_INVALID_BIT_ORDER`
Specified bit order is invalid.
- `SPI_UNSUPPORTED_MODE`
Specified SPI mode is not supported.
- `SPI_UNSUPPORTED_BIT_ORDER`
Specified bit order is not supported.
- `SPI_UNSUPPORTED_TRANSFER_SIZE`
Specified transfer size is not supported.

Example

```
/* This example demonstrates setting all transfer attributes associated  
   with an SPI slave device. */  
  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start
```

```
    service call. */

/* Declare the required variables. */
STATUS status;
SPI_TRANSFER_CONFIG config;

/* Chose the desired transfer attributes. */
config.spi_clock_phase    = SPI_CPHA_0;
config.spi_clock_polarity = SPI_CPOL_0;
config.spi_bit_order      = SPI_MSB_FIRST;
config.spi_transfer_size  = 8;

/* Change the transfer attributes associated with the device. */
status = NU_SPI_Slave_Set_Configuration(SPI_Dev_Handle, 2, &config);

/* At this point status indicates if the transfer attributes were
   successfully changed. */
```

Related Topics

[NU_SPI_Slave_Get_Configuration](#)

[Transfer Attribute Control](#)

NU_SPI_Slave_Set_SPI_Mode

This function sets the SPI mode for the specified SPI slave device.

Usage

```
STATUS NU_SPI_Slave_Set_SPI_Mode (SPI_HANDLE spi_dev,  
                                UINT8      clock_phase,  
                                UINT8      clock_polarity)
```

Arguments

- **spi_dev**
The specified SPI device.
- **clock_phase**
The desired clock phase.
- **clock_polarity**
The desired clock polarity.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_MODE**
Specified SPI mode is not valid.
- **SPI_UNSUPPORTED_MODE**
Specified SPI mode is not supported.

Example

```
/* This example demonstrates setting the SPI mode. */  
  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */  
  
/* Declare the required variables. */  
STATUS status;  
  
/* Set the SPI mode to SPI mode 1 (CPOL0, CPHA1). */  
status = NU_SPI_Slave_Set_SPI_Mode(SPI_Dev_Handle, SPI_CPOL_0,  
                                   SPI_CPHA_1);  
  
/* At this point status indicates if SPI mode was successfully changed. */
```

Related Topics

[NU_SPI_Slave_Get_Configuration](#)

[NU_SPI_Slave_Set_Configuration](#)

[Transfer Attribute Control](#)

NU_SPI_Slave_Set_Transfer_Size

This function sets the transfer size for the specified SPI slave device.

Usage

```
STATUS NU_SPI_Slave_Set_Transfer_Size (SPI_HANDLE spi_dev,  
                                       UINT8      transfer_size)
```

Arguments

- `spi_dev`
The specified SPI device.
- `transfer_size`
The desired transfer size.

Return Values

- `NU_SUCCESS`
Service completed successfully.
- `SPI_INVALID_HANDLE`
Nucleus SPI device handle is not valid.
- `SPI_UNSUPPORTED_TRANSFER_SIZE`
Specified transfer size is not supported.

Example

```
/* This example demonstrates setting transfer size. */  
  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */  
  
/* Declare the required variables. */  
STATUS status;  
  
/* Set the transfer size to 7. */  
status = NU_SPI_Slave_Set_SPI_Mode(SPI_Dev_Handle, 2, 7);  
  
/* At this point status indicates if transfer size was successfully  
   changed. */
```

Related Topics

[NU_SPI_Slave_Get_Configuration](#)

[NU_SPI_Slave_Set_Configuration](#)

[Transfer Attribute Control](#)

Data Transfer

SPI supports variable transfer size (though this depends upon hardware). Nucleus SPI provides a separate set of functions for each range of transfer size:

- 8 Bit version for 1-bit to 8-bit transfer size.
- 16 Bit version for 1-bit to 16-bit transfer size.
- 32 Bit version for 1-bit to 32-bit transfer size.

Data Transfer Functions in Interrupt-Driven Mode

Nucleus SPI does not buffer received data during SPI transfers into any internal buffer. The received data is directly stored in the Rx buffer you specify when a data transfer is requested. The Rx buffer must remain alive until the completion of the data transfer (for example, the transfer complete notification is received). This means the buffer memory cannot be freed or reused until the transfer completion notification is received. As a result, the Rx buffer does not contain received data when the Nucleus SPI service used to place a transfer request returns. The complete received data will be available in the Rx buffer when the application receives a transfer complete notification from Nucleus SPI.

For applications that want to bypass buffer copy overhead completely, Nucleus SPI provides the option of user buffering. This works similar to the normal interrupt-driven mode, except that the no internal copy of the Tx data you specify is made. The Tx data buffer you specify must remain alive until the completion of the transfer.

To summarize:

- In interrupt-driven mode, with or without user buffering, the Rx buffer must remain alive until a transfer complete notification is received.
- If user buffering is being used in the interrupt-driven mode, then, the Tx buffer must also remain alive until a transfer complete notification is received.

Data Transfer Functions in Polling Mode

In polling mode, user buffers are directly used and no internal copying is performed. During the transfer of each data unit, the data to be transmitted is picked from the Tx data buffer you specify and the received data is placed in the specified Rx buffer. This process continues until all data units are transferred. As a result, when the Nucleus SPI service used to place the transfer request returns, the Rx buffer contains all the received data. No user callback is called by Nucleus SPI for devices using polling mode.

This section describes the following SPI transfer attribute control functions.

- [NU_SPI_Master_Duplex_8Bit](#)
- [NU_SPI_Master_Duplex_16Bit](#)
- [NU_SPI_Master_Duplex_32Bit](#)
- [NU_SPI_Master_Receive_8Bit](#)
- [NU_SPI_Master_Receive_16Bit](#)
- [NU_SPI_Master_Receive_32Bit](#)
- [NU_SPI_Master_Transmit_8Bit](#)
- [NU_SPI_Master_Transmit_16Bit](#)
- [NU_SPI_Master_Transmit_32Bit](#)
- [NU_SPI_Slave_8Bit_Duplex_Response](#)
- [NU_SPI_Slave_16Bit_Duplex_Response](#)
- [NU_SPI_Slave_32Bit_Duplex_Response](#)
- [NU_SPI_Slave_8Bit_Receive_Response](#)
- [NU_SPI_Slave_16Bit_Receive_Response](#)
- [NU_SPI_Slave_32Bit_Receive_Response](#)
- [NU_SPI_Slave_8Bit_Transmit_Response](#)
- [NU_SPI_Slave_16Bit_Transmit_Response](#)
- [NU_SPI_Slave_32Bit_Transmit_Response](#)

Related Topics

[Data Transfer](#)

[SPI Function Reference](#)

NU_SPI_Master_Duplex_8Bit

This function performs a duplex transfer of data units 1-bit to 8-bits in size with the given slave of the specified SPI master device.

Usage

```
STATUS NU_SPI_Master_Duplex_8Bit (SPI_HANDLE spi_dev,  
                                UINT16      address,  
                                UINT8       *tx_data,  
                                UINT8       *rx_buffer,  
                                UNSIGNED_INT length)
```

Arguments

- **spi_dev**
The specified SPI device.
- **address**
The specified slave with which transfer should be done.
- **tx_data**
Pointer to the buffer that contains data units to be transmitted.
- **rx_buffer**
Pointer to the buffer that will get the simultaneously received data units.
- **length**
Number of data units to be transferred. The buffer pointed to by rx_buffer must be able to hold this many data units.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
The specified handle is invalid.
- **SPI_INVALID_ADDRESS**
Specified slave address is not valid for the specified SPI device.
- **SPI_QUEUE_FULL**
Queue is full.
- **SPI_BUFFER_NOT_ENOUGH**
The buffer does not have enough free space for the specified data.

Example

```
/* This example demonstrates an 8-bit duplex transfer. */

SPI_HANDLE SPI_Dev_Handle;

/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device
   which has previously been initialized with Nucleus SPI NU_SPI_Start
   service call. */

/* Declare the required variables. */
STATUS status;
UINT8 tx_data[4];
UINT8 rx_buffer[4];

/* Setup some data to be transmitted. */
tx_data[0] = 'P';
tx_data[1] = 'L';
tx_data[2] = 'U';
tx_data[3] = 'S';

/* Perform a duplex transfer with slave number 1. */
status =
    NU_SPI_Master_Duplex_8Bit(SPI_Dev_Handle, 1, tx_data, rx_buffer, 4);

/* At this point status indicates if the duplex transfer request was
   successful. */
```

Related Topics

[NU_SPI_Master_Duplex_16Bit](#)

[NU_SPI_Master_Duplex_32Bit](#)

[Data Transfer](#)

NU_SPI_Master_Duplex_16Bit

This function performs a duplex transfer of data units 1-bit to 16-bits in size with the given slave of the specified SPI master device.

Usage

```
STATUS NU_SPI_Master_Duplex_16Bit (SPI_HANDLE spi_dev,  
                                  UINT16      address,  
                                  UINT16      *tx_data,  
                                  UINT16      *rx_buffer,  
                                  UNSIGNED_INT length)
```

Arguments

- **spi_dev**
The specified SPI device.
- **address**
The specified slave with which transfer should be done.
- **tx_data**
Pointer to the buffer that contains data units to be transmitted.
- **rx_buffer**
Pointer to the buffer that will get the simultaneously received data units.
- **length**
Number of data units to be transferred. The buffer pointed to by rx_buffer must be able to hold this many data units.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
The specified handle is invalid.
- **SPI_INVALID_ADDRESS**
Specified slave address is not valid for the specified SPI device.
- **SPI_QUEUE_FULL**
Queue is full.
- **SPI_BUFFER_NOT_ENOUGH**
The buffer does not have enough free space for the specified data.

Example

```
/* This example demonstrates a 16-bit duplex transfer. */

SPI_HANDLE SPI_Dev_Handle;

/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device
   which has previously been initialized with Nucleus SPI NU_SPI_Start
   service call. */

/* Declare the required variables. */
STATUS status;
UINT16 tx_data[4];
UINT16 rx_buffer[4];

/* Setup some data to be transmitted. */
tx_data[0] = 0x0011;
tx_data[1] = 0x2233;
tx_data[2] = 0x3344;
tx_data[3] = 0x4455;

/* Perform a duplex transfer with slave number 1. */
status =
    NU_SPI_Master_Duplex_16Bit(SPI_Dev_Handle, 1, tx_data, rx_buffer, 4);

/* At this point status indicates if the duplex transfer request was
   successful. */
```

Related Topics

[NU_SPI_Master_Duplex_8Bit](#)

[NU_SPI_Master_Duplex_32Bit](#)

[Data Transfer](#)

NU_SPI_Master_Duplex_32Bit

This function performs a duplex transfer of data units 1-bit to 32-bits in size with the given slave of the specified SPI master device.

Usage

```
STATUS NU_SPI_Master_Duplex_32Bit (SPI_HANDLE spi_dev,  
                                  UINT16      address,  
                                  UINT32      *tx_data,  
                                  UINT32      *rx_buffer,  
                                  UNSIGNED_INT length)
```

Arguments

- **spi_dev**
The specified SPI device.
- **address**
The specified slave with which transfer should be done.
- **tx_data**
Pointer to the buffer that contains data units to be transmitted.
- **rx_buffer**
Pointer to the buffer that will get the simultaneously received data units.
- **length**
Number of data units to be transferred. The buffer pointed to by rx_buffer must be able to hold this many data units.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
The specified handle is invalid.
- **SPI_INVALID_ADDRESS**
Specified slave address is not valid for the specified SPI device.
- **SPI_QUEUE_FULL**
Queue is full.
- **SPI_BUFFER_NOT_ENOUGH**
The buffer does not have enough free space for the specified data.

Example

```
/* This example demonstrates a 32-bit duplex transfer. */

SPI_HANDLE SPI_Dev_Handle;

/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device
   which has previously been initialized with Nucleus SPI NU_SPI_Start
   service call. */

/* Declare the required variables. */
STATUS status;
UINT32 tx_data[4];
UINT32 rx_buffer[4];

/* Setup some data to be transmitted. */
tx_data[0] = 0x00112233;
tx_data[1] = 0x44556677;
tx_data[2] = 0x8899AABB;
tx_data[3] = 0xCCDDEEFF;

/* Perform a duplex transfer with slave number 1. */
status =
    NU_SPI_Master_Duplex_32Bit(SPI_Dev_Handle, 1, tx_data, rx_buffer, 4);

/* At this point status indicates if the duplex transfer request was
   successful. */
```

Related Topics

[NU_SPI_Master_Duplex_8Bit](#)

[NU_SPI_Master_Duplex_16Bit](#)

[Data Transfer](#)

NU_SPI_Master_Receive_8Bit

This function is responsible for reception of data units 1-bit to 8-bits in size from the given slave of the specified SPI device.

Usage

```
STATUS NU_SPI_Master_Receive_8Bit (SPI_HANDLE spi_dev,  
                                  UINT16      address,  
                                  UINT8       *rx_buffer,  
                                  UNSIGNED_INT length)
```

Arguments

- **spi_dev**
The specified SPI device.
- **address**
The specified slave from which the data should be received.
- **rx_buffer**
Pointer to the buffer that will get the received data units.
- **length**
Number of data units to be received. The buffer pointed to by rx_buffer must be able to hold this many data units.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
The specified handle is invalid.
- **SPI_INVALID_ADDRESS**
Specified slave address is not valid for the specified SPI device.
- **SPI_QUEUE_FULL**
Queue is full.

Example

```
/* This example demonstrates an 8-bit reception. */  
  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */  
  
/* Declare the required variables. */  
STATUS status;
```



```
UINT8 rx_buffer[4];

/* Request the reception from slave number 1. */
status = NU_SPI_Master_Receive_8Bit(SPI_Dev_Handle, 1, rx_buffer, 4);

/* At this point status indicates if the reception request was
   successful. */
```

Related Topics

[NU_SPI_Master_Receive_16Bit](#)

[NU_SPI_Master_Receive_32Bit](#)

[Data Transfer](#)

NU_SPI_Master_Receive_16Bit

This function is responsible for reception of data units 1-bit to 16-bits in size from the given slave of the specified SPI device.

Usage

```
STATUS NU_SPI_Master_Receive_16Bit (SPI_HANDLE    spi_dev,  
                                   UINT16         address,  
                                   UINT16         *rx_buffer,  
                                   UNSIGNED_INT    length)
```

Arguments

- **spi_dev**
The specified SPI device.
- **address**
The specified slave from which the data should be received.
- **rx_buffer**
Pointer to the buffer that will get the received data units.
- **length**
Number of data units to be received. The buffer pointed to by rx_buffer must be able to hold this many data units.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
The specified handle is invalid.
- **SPI_INVALID_ADDRESS**
Specified slave address is not valid for the specified SPI device.
- **SPI_QUEUE_FULL**
Queue is full.

Example

```
/* This example demonstrates a 16-bit reception. */  
  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */  
  
/* Declare the required variables. */  
STATUS status;
```

```
UINT16 rx_buffer[4];

/* Request the reception from slave number 1. */
status = NU_SPI_Master_Receive_16Bit(SPI_Dev_Handle, 1, rx_buffer, 4);

/* At this point status indicates if the reception request was
   successful. */
```

Related Topics

[NU_SPI_Master_Receive_8Bit](#)

[NU_SPI_Master_Receive_32Bit](#)

[Data Transfer](#)

NU_SPI_Master_Receive_32Bit

This function is responsible for reception of data units 1-bit to 32-bits in size from the given slave of the specified SPI device.

Usage

```
STATUS NU_SPI_Master_Receive_32Bit (SPI_HANDLE    spi_dev,  
                                   UINT16         address,  
                                   UINT32         *rx_buffer,  
                                   UNSIGNED_INT    length)
```

Arguments

- **spi_dev**
The specified SPI device.
- **address**
The specified slave from which the data should be received.
- **rx_buffer**
Pointer to the buffer that will get the received data units.
- **length**
Number of data units to be received. The buffer pointed to by rx_buffer must be able to hold this many data units.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
The specified handle is invalid.
- **SPI_INVALID_ADDRESS**
Specified slave address is not valid for the specified SPI device.
- **SPI_QUEUE_FULL**
Queue is full.

Example

```
/* This example demonstrates a 32-bit reception. */  
  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */  
  
/* Declare the required variables. */  
STATUS status;
```

```
UINT32 rx_buffer[4];

/* Request the reception from slave number 1. */
status = NU_SPI_Master_Receive_32Bit(SPI_Dev_Handle, 1, rx_buffer, 4);

/* At this point status indicates if the reception request was
   successful. */
```

Related Topics

[NU_SPI_Master_Receive_8Bit](#)

[NU_SPI_Master_Receive_16Bit](#)

[Data Transfer](#)

NU_SPI_Master_Transmit_8Bit

This function transmits specified data units 1-bit to 8-bits in size to the given slave of the specified SPI master device.

Usage

```
STATUS NU_SPI_Master_Transmit_8Bit (SPI_HANDLE    spi_dev,  
                                   UINT16         address,  
                                   UINT8          *tx_data,  
                                   UNSIGNED_INT    length)
```

Arguments

- **spi_dev**
The specified SPI device.
- **address**
The specified slave to which the data should be transmitted.
- **tx_data**
Pointer to the buffer that contains data units to be transmitted.
- **length**
Number of data units to be transmitted.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
The specified handle is invalid.
- **SPI_INVALID_ADDRESS**
Specified slave address is not valid for the specified SPI device.
- **SPI_QUEUE_FULL**
Queue is full.
- **SPI_BUFFER_NOT_ENOUGH**
The buffer does not have enough free space for the specified data.

Example

```
/* This example demonstrates an 8-bit transmission. */  
  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */
```

```
/* Declare the required variables. */
STATUS status;
UINT8 tx_data[4];

/* Setup some data to be transmitted. */
tx_data[0] = 'P';
tx_data[1] = 'L';
tx_data[2] = 'U';
tx_data[3] = 'S';

/* Transmit to slave number 1. */
status = NU_SPI_Master_Transmit_8Bit(SPI_Dev_Handle, 1, tx_data, 4);

/* At this point status indicates if the transmission request was
   successful. */
```

Related Topics

[NU_SPI_Master_Transmit_16Bit](#)

[NU_SPI_Master_Transmit_32Bit](#)

[Data Transfer](#)

NU_SPI_Master_Transmit_16Bit

This function transmits specified data units 1-bit to 16-bits in size to the given slave of the specified SPI master device.

Usage

```
STATUS NU_SPI_Master_Transmit_16Bit (SPI_HANDLE    spi_dev,  
                                     UINT16         address,  
                                     UINT16         *tx_data,  
                                     UNSIGNED_INT    length)
```

Arguments

- **spi_dev**
The specified SPI device.
- **address**
The specified slave to which the data should be transmitted.
- **tx_data**
Pointer to the buffer that contains data units to be transmitted.
- **length**
Number of data units to be transmitted.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
The specified handle is invalid.
- **SPI_INVALID_ADDRESS**
Specified slave address is not valid for the specified SPI device.
- **SPI_QUEUE_FULL**
Queue is full.
- **SPI_BUFFER_NOT_ENOUGH**
The buffer does not have enough free space for the specified data.

Example

```
/* This example demonstrates a 16-bit transmission. */  
  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */
```



```
/* Declare the required variables. */
STATUS status;
UINT16 tx_data[4];

/* Setup some data to be transmitted. */
tx_data[0] = 0x0011;
tx_data[1] = 0x2233;
tx_data[2] = 0x3344;
tx_data[3] = 0x4455;

/* Transmit to slave number 1. */
status = NU_SPI_Master_Transmit_16Bit(SPI_Dev_Handle, 1, tx_data, 4);

/* At this point status indicates if the transmission request was
   successful. */
```

Related Topics

[NU_SPI_Master_Transmit_8Bit](#)

[NU_SPI_Master_Transmit_32Bit](#)

[Data Transfer](#)

NU_SPI_Master_Transmit_32Bit

This function transmits specified data units 1-bit to 32-bits in size to the given slave of the specified SPI master device.

Usage

```
STATUS NU_SPI_Master_Transmit_32Bit (SPI_HANDLE    spi_dev,  
                                     UINT16         address,  
                                     UINT32         *tx_data,  
                                     UNSIGNED_INT    length)
```

Arguments

- **spi_dev**
The specified SPI device.
- **address**
The specified slave to which the data should be transmitted.
- **tx_data**
Pointer to the buffer that contains data units to be transmitted.
- **length**
Number of data units to be transmitted.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
The specified handle is invalid.
- **SPI_INVALID_ADDRESS**
Specified slave address is not valid for the specified SPI device.
- **SPI_QUEUE_FULL**
Queue is full.
- **SPI_BUFFER_NOT_ENOUGH**
The buffer does not have enough free space for the specified data.

Example

```
/* This example demonstrates a 32-bit transmission. */  
  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */
```

```
/* Declare the required variables. */
STATUS status;
UINT32 tx_data[4];

/* Setup some data to be transmitted. */
tx_data[0] = 0x00112233;
tx_data[1] = 0x44556677;
tx_data[2] = 0x8899AABB;
tx_data[3] = 0xCCDDEEFF;

/* Transmit to slave number 1. */
status = NU_SPI_Master_Transmit_32Bit(SPI_Dev_Handle, 1, tx_data, 4);

/* At this point status indicates if the transmission request was
   successful. */
```

Related Topics

[NU_SPI_Master_Transmit_8Bit](#)

[NU_SPI_Master_Transmit_16Bit](#)

[Data Transfer](#)

NU_SPI_Slave_8Bit_Duplex_Response

This function sets the response for a duplex transfer of data units 1-bit to 8-bits in size on the specified SPI slave device.

Usage

```
STATUS NU_SPI_Slave_8Bit_Duplex_Response (SPI_HANDLE  spi_dev,  
                                          UINT8        *tx_data,  
                                          UINT8        *rx_buffer,  
                                          UNSIGNED_INT length)
```

Arguments

- **spi_dev**
The specified SPI device.
- **tx_data**
Pointer to the buffer that contains data units to be transmitted.
- **rx_buffer**
Pointer to the buffer that will get the simultaneously received data units.
- **length**
Number of data units to be transferred. The buffer pointed to by rx_buffer must be able to hold this many data units.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
The specified handle is invalid.
- **SPI_QUEUE_FULL**
Queue is full.
- **SPI_BUFFER_NOT_ENOUGH**
The buffer does not have enough free space for the specified data.

Example

```
/* This example demonstrates setting an 8-bit duplex response. */  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */
```

```
/* Declare the required variables. */
STATUS status;
UINT8 tx_data[4];
UINT8 rx_buffer[4];

/* Setup some data to be transmitted. */
tx_data[0] = 'P';
tx_data[1] = 'L';
tx_data[2] = 'U';
tx_data[3] = 'S';

/* Set an 8-bit duplex response. */
status = NU_SPI_Slave_8Bit_Duplex_Response(SPI_Dev_Handle, tx_data,
                                           rx_buffer, 4);

/* At this point status indicates if the duplex response was
   successfully set. */
```

Related Topics

[NU_SPI_Slave_16Bit_Duplex_Response](#)

[NU_SPI_Slave_32Bit_Duplex_Response](#)

[Data Transfer](#)

NU_SPI_Slave_16Bit_Duplex_Response

This function sets the response for a duplex transfer of data units 1-bit to 16-bits in size on the specified SPI slave device.

Usage

```
STATUS NU_SPI_Slave_16Bit_Duplex_Response (SPI_HANDLE spi_dev,  
                                           UINT16      *tx_data,  
                                           UINT16      *rx_buffer,  
                                           UNSIGNED_INT length)
```

Arguments

- **spi_dev**
The specified SPI device.
- **tx_data**
Pointer to the buffer that contains data units to be transmitted.
- **rx_buffer**
Pointer to the buffer that will get the simultaneously received data units.
- **length**
Number of data units to be transferred. The buffer pointed to by rx_buffer must be able to hold this many data units.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
The specified handle is invalid.
- **SPI_QUEUE_FULL**
Queue is full.
- **SPI_BUFFER_NOT_ENOUGH**
The buffer does not have enough free space for the specified data.

Example

```
/* This example demonstrates setting a 16-bit duplex response. */  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */
```

```
/* Declare the required variables. */
STATUS status;
UINT16 tx_data[4];
UINT16 rx_buffer[4];

/* Setup some data to be transmitted. */
tx_data[0] = 0x0011;
tx_data[1] = 0x2233;
tx_data[2] = 0x3344;
tx_data[3] = 0x4455;

/* Set a 16-bit duplex response. */
status = NU_SPI_Slave_16Bit_Duplex_Response(SPI_Dev_Handle, tx_data,
                                             rx_buffer, 4);

/* At this point status indicates if the duplex response was
   successfully set. */
```

Related Topics

[NU_SPI_Slave_8Bit_Duplex_Response](#)

[NU_SPI_Slave_32Bit_Duplex_Response](#)

[Data Transfer](#)

NU_SPI_Slave_32Bit_Duplex_Response

This function sets the response for a duplex transfer of data units 1-bit to 32-bits in size on the specified SPI slave device.

Usage

```
STATUS NU_SPI_Slave_32Bit_Duplex_Response (SPI_HANDLE spi_dev,  
                                           UINT32      *tx_data,  
                                           UINT32      *rx_buffer,  
                                           UNSIGNED_INT length)
```

Arguments

- **spi_dev**
The specified SPI device.
- **tx_data**
Pointer to the buffer that contains data units to be transmitted.
- **rx_buffer**
Pointer to the buffer that will get the simultaneously received data units.
- **length**
Number of data units to be transferred. The buffer pointed to by rx_buffer must be able to hold this many data units.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
The specified handle is invalid.
- **SPI_QUEUE_FULL**
Queue is full.
- **SPI_BUFFER_NOT_ENOUGH**
The buffer does not have enough free space for the specified data.

Example

```
/* This example demonstrates setting a 32-bit duplex response. */  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */
```



```
/* Declare the required variables. */
STATUS status;
UINT32 tx_data[4];
UINT32 rx_buffer[4];

/* Setup some data to be transmitted. */
tx_data[0] = 0x00112233;
tx_data[1] = 0x44556677;
tx_data[2] = 0x8899AABB;
tx_data[3] = 0xCCDDEEFF;

/* Set an 32-bit duplex response. */
status = NU_SPI_Slave_32Bit_Duplex_Response(SPI_Dev_Handle, tx_data,
                                             rx_buffer, 4);

/* At this point status indicates if the duplex response was
   successfully set. */
```

Related Topics

[NU_SPI_Slave_8Bit_Duplex_Response](#)

[NU_SPI_Slave_16Bit_Duplex_Response](#)

[Data Transfer](#)

NU_SPI_Slave_8Bit_Receive_Response

This function sets the response for reception of data units 1-bit to 8-bit s in size on the specified SPI slave device.

Usage

```
STATUS NU_SPI_Slave_8Bit_Receive_Response (SPI_HANDLE    spi_dev,  
                                           UINT8          *rx_buffer,  
                                           UNSIGNED_INT  length)
```

Arguments

- **spi_dev**
The specified SPI device.
- **rx_buffer**
Pointer to the buffer that will get the received data units.
- **length**
Number of data units to be received. The buffer pointed to by rx_buffer must be able to hold this many data units.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
The specified handle is invalid.
- **SPI_QUEUE_FULL**
Queue is full.

Example

```
/* This example demonstrates an 8-bit reception response. */  
  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */  
  
/* Declare the required variables. */  
STATUS status;  
UINT8 rx_buffer[4];  
  
/* Set the receive response. */  
status = NU_SPI_Slave_8Bit_Receive_Response(SPI_Dev_Handle, rx_buffer, 4);  
  
/* At this point status indicates if the reception response was  
   successfully set. */
```

Related Topics

[NU_SPI_Slave_16Bit_Receive_Response](#)

[NU_SPI_Slave_32Bit_Receive_Response](#)

[Data Transfer](#)

NU_SPI_Slave_16Bit_Receive_Response

This function sets the response for reception of data units 1-bit to 16-bits in size on the specified SPI slave device.

Usage

```
STATUS NU_SPI_Slave_16Bit_Receive_Response (SPI_HANDLE   spi_dev,  
                                             UINT16         *rx_buffer,  
                                             UNSIGNED_INT   length)
```

Arguments

- **spi_dev**
The specified SPI device.
- **rx_buffer**
Pointer to the buffer that will get the received data units.
- **length**
Number of data units to be received. The buffer pointed to by rx_buffer must be able to hold this many data units.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
The specified handle is invalid.
- **SPI_QUEUE_FULL**
Queue is full.

Example

```
/* This example demonstrates a 16-bit reception response. */  
  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */  
  
/* Declare the required variables. */  
STATUS status;  
UINT16 rx_buffer[4];  
  
/* Set the receive response. */  
status = NU_SPI_Slave_16Bit_Receive_Response(SPI_Dev_Handle, rx_buffer,  
                                              4);  
  
/* At this point status indicates if the reception response was  
   successfully set. */
```

Related Topics

[NU_SPI_Slave_8Bit_Receive_Response](#)

[NU_SPI_Slave_32Bit_Receive_Response](#)

[Data Transfer](#)

NU_SPI_Slave_32Bit_Receive_Response

This function sets the response for reception of data units 1-bit to 32-bits in size on the specified SPI slave device.

Usage

```
STATUS NU_SPI_Slave_32Bit_Receive_Response (SPI_HANDLE   spi_dev,  
                                            UINT32        *rx_buffer,  
                                            UNSIGNED_INT  length)
```

Arguments

- **spi_dev**
The specified SPI device.
- **rx_buffer**
Pointer to the buffer that will get the received data units.
- **length**
Number of data units to be received. The buffer pointed to by **rx_buffer** must be able to hold this many data units.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
The specified handle is invalid.
- **SPI_QUEUE_FULL**
Queue is full.

Example

```
/* This example demonstrates a 32-bit reception response. */  
  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */  
  
/* Declare the required variables. */  
STATUS status;  
UINT32 rx_buffer[4];  
  
/* Set the receive response. */  
status = NU_SPI_Slave_32Bit_Receive_Response(SPI_Dev_Handle, rx_buffer,  
                                              4);  
  
/* At this point status indicates if the reception response was  
   successfully set. */
```

Related Topics

[NU_SPI_Slave_8Bit_Receive_Response](#)

[NU_SPI_Slave_16Bit_Receive_Response](#)

[Data Transfer](#)

NU_SPI_Slave_8Bit_Transmit_Response

This function sets the response for transmission of the specified data units 1-bit to 8-bits in size on the specified SPI slave device.

Usage

```
STATUS NU_SPI_Slave_8Bit_Transmit_Response (SPI_HANDLE  spi_dev,  
                                           UINT8        *tx_data,  
                                           UNSIGNED_INT length)
```

Arguments

- **spi_dev**
The specified SPI device.
- **tx_data**
Pointer to the buffer that contains data units to be transmitted.
- **length**
Number of data units to be transmitted.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
The specified handle is invalid.
- **SPI_QUEUE_FULL**
Queue is full.
- **SPI_BUFFER_NOT_ENOUGH**
The buffer does not have enough free space for the specified data.

Example

```
/* This example demonstrates setting an 8-bit transmission response. */  
  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */  
  
/* Declare the required variables. */  
STATUS status;  
UINT8 tx_data[4];  
  
/* Setup some data to be transmitted. */  
tx_data[0] = 'P';  
tx_data[1] = 'L';  
tx_data[2] = 'U';
```



```
tx_data[3] = 'S';

/* Set the transmission response. */
status = NU_SPI_Slave_8Bit_Transmit_Response(SPI_Dev_Handle, tx_data, 4);

/* At this point status indicates if the transmission response was
   Successfully set. */
```

Related Topics

[NU_SPI_Slave_16Bit_Transmit_Response](#)

[NU_SPI_Slave_32Bit_Transmit_Response](#)

[Data Transfer](#)

NU_SPI_Slave_16Bit_Transmit_Response

This function sets the response for transmission of the specified data units 1-bit to 16-bits in size on the specified SPI slave device.

Usage

```
STATUS NU_SPI_Slave_16Bit_Transmit_Response (SPI_HANDLE  spi_dev,  
                                             UINT16        *tx_data,  
                                             UNSIGNED_INT  length)
```

Arguments

- **spi_dev**
The specified SPI device.
- **tx_data**
Pointer to the buffer that contains data units to be transmitted.
- **length**
Number of data units to be transmitted.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
The specified handle is invalid.
- **SPI_QUEUE_FULL**
Queue is full.
- **SPI_BUFFER_NOT_ENOUGH**
The buffer does not have enough free space for the specified data.

Example

```
/* This example demonstrates setting a 16-bit transmission response. */  
  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */  
  
/* Declare the required variables. */  
STATUS status;  
UINT16 tx_data[4];  
  
/* Setup some data to be transmitted. */  
tx_data[0] = 0x0011;  
tx_data[1] = 0x2233;  
tx_data[2] = 0x3344;
```

```
tx_data[3] = 0x4455;

/* Set the transmission response. */
status =
    NU_SPI_Slave_16Bit_Transmit_Response(SPI_Dev_Handle, tx_data, 4);

/* At this point status indicates if the transmission response was
   Successfully set. */
```

Related Topics

[NU_SPI_Slave_8Bit_Transmit_Response](#)

[NU_SPI_Slave_32Bit_Transmit_Response](#)

[Data Transfer](#)

NU_SPI_Slave_32Bit_Transmit_Response

This function sets the response for transmission of the specified data units 1-bit to 32-bits in size on the specified SPI slave device.

Usage

```
STATUS NU_SPI_Slave_32Bit_Transmit_Response (SPI_HANDLE  spi_dev,  
                                             UINT32        *tx_data,  
                                             UNSIGNED_INT  length)
```

Arguments

- **spi_dev**
The specified SPI device.
- **tx_data**
Pointer to the buffer that contains data units to be transmitted.
- **length**
Number of data units to be transmitted.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
The specified handle is invalid.
- **SPI_QUEUE_FULL**
Queue is full.
- **SPI_BUFFER_NOT_ENOUGH**
The buffer does not have enough free space for the specified data.

Example

```
/* This example demonstrates setting a 32-bit transmission response. */  
  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */  
  
/* Declare the required variables. */  
STATUS status;  
UINT32 tx_data[4];  
  
/* Setup some data to be transmitted. */  
tx_data[0] = 0x00112233;  
tx_data[1] = 0x44556677;  
tx_data[2] = 0x8899AABB;
```

```
tx_data[3] = 0xCCDDEEFF;

/* Set the transmission response. */
status = NU_SPI_Slave_32Bit_Transmit_Response(SPI_Dev_Handle, tx_data,
                                              4);

/* At this point status indicates if the transmission response was
   Successfully set. */
```

Related Topics

[NU_SPI_Slave_8Bit_Transmit_Response](#)

[NU_SPI_Slave_16Bit_Transmit_Response](#)

[Data Transfer](#)

Miscellaneous

This section describes the following miscellaneous functions:

- [NU_SPI_Ioctl_Driver](#)
- [NU_SPI_Master_Get_Callbacks](#)
- [NU_SPI_Master_Get_Driver_Mode](#)
- [NU_SPI_Master_Set_Callbacks](#)
- [NU_SPI_Master_Set_SS_Polarity](#)

Related Topics

[SPI Function Reference](#)

NU_SPI_Ioctl_Driver

This function provides an interface to the I/O control service of the underlying SPI hardware driver.

Usage

```
STATUS NU_SPI_Ioctl_Driver (SPI_HANDLE spi_dev,  
                           UINT16      address,  
                           INT          control_code,  
                           VOID         *control_info)
```

Arguments

- **spi_dev**
The specified SPI device.
- **address**
Address of the slave.
- **control_code**
Specifies the control operation to be performed.
- **control_info**
Information associated to/required by the control operation.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
Nucleus SPI device handle is not valid.
- **SPI_INVALID_IOCTL_OPERATION**
Unsupported operation requested for I/O control.
- **other**
Error codes depending upon the control operation implemented by the user.

Description

This function provides an interface that allows you to implement target-specific operations, such as setting different delays involved in SPI transfers. Operations such as setting the baud rate, SPI mode, bit order and transfer size are excluded. These settings are managed by Nucleus SPI and can only be changed through the corresponding Nucleus SPI function.

Example

```
/* This example demonstrates using I/O control. Assume that SET_DELAYS I/O  
   control operation has been implemented in the Nucleus SPI Driver. */
```

```
SPI_HANDLE SPI_Dev_Handle;

/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device
   which has previously been initialized with Nucleus SPI NU_SPI_Start
   service call. */

/* Declare the required variables. */
STATUS status;

/* An assumed structure that contains delay information. */
DELAYS_STRUCT delays;

/* Set delays using I/O control service. */
status = NU_SPI_Ioctl_Driver(SPI_Dev_Handle, 0, SET_DELAYS, &delays);

/* At this point status indicates if the I/O control operation was
   successful. */
```

Related Topics

[NU_SPI_Slave_Set_Configuration](#)

[Miscellaneous](#)

NU_SPI_Master_Get_Callbacks

This function gets the callback functions for the specified SPI slave device.

Usage

```
STATUS NU_SPI_Master_Get_Callbacks (SPI_HANDLE      spi_dev,  
                                   UINT8           address,  
                                   SPI_APP_CALLBACKS *callbacks)
```

Arguments

- **spi_dev**
The specified SPI device.
- **address**
The specified slave device address.
- **callbacks**
Pointer to the callback structure which will get the queried callback functions.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
Nucleus SPI device handle is not valid.
- **SPI_INVALID_ADDRESS**
Specified slave address is not valid for the specified SPI device.
- **SPI_INVALID_PARAM_POINTER**
Null given instead of a variable pointer.

Example

```
/* This example demonstrates setting transfer size. */  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */  
  
/* Declare the required variables. */  
STATUS status;  
SPI_APP_CALLBACKS callbacks;  
  
/* Get the callback functions for slave 1. */  
status = NU_SPI_Master_Get_Callbacks (SPI_Dev_Handle, 1,&callbacks);  
  
/* At this point status indicates if callback functions were  
   successfully changed. */
```

Related Topics

[NU_SPI_Master_Set_Callbacks](#)

[Miscellaneous](#)

NU_SPI_Master_Get_Driver_Mode

This function gets the driver mode of the specified SPI master device.

Usage

```
STATUS NU_SPI_Master_Get_Driver_Mode (SPI_HANDLE    spi_dev,  
                                     SPI_DRIVER_MODE *spi_driver_mode)
```

Arguments

- **spi_dev**
The specified SPI device.
- **spi_driver_mode**
Pointer to a Nucleus SPI driver mode that gets the queried driver mode information.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
Nucleus SPI device handle is not valid.

Example

```
/* This example demonstrates setting transfer size. */  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */  
  
/* Declare the required variables. */  
STATUS status;  
SPI_DRIVER_MODE spi_driver_mode;  
  
/* Set the slave-select polarity for slave 2 to HIGH. */  
status = NU_SPI_Slave_Get_Driver_Mode (SPI_Dev_Handle,&spi_driver_mode);  
  
/* At this point status indicates if the driver mode is returned  
   successfully. */
```

Related Topics

[NU_SPI_Master_Get_Configuration](#)

[Miscellaneous](#)

NU_SPI_Master_Set_Callbacks

This function sets the callback functions for the specified SPI slave device.

Usage

```
STATUS NU_SPI_Master_Set_Callbacks (SPI_HANDLE      spi_dev,  
                                   UINT8           address,  
                                   SPI_APP_CALLBACKS *callbacks)
```

Arguments

- **spi_dev**
The specified SPI device.
- **address**
The specified slave device address.
- **callbacks**
Pointer to the desired callback functions structure.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
Nucleus SPI device handle is not valid.
- **SPI_INVALID_ADDRESS**
Specified slave address is not valid for the specified SPI device.
- **SPI_INVALID_PARAM_POINTER**
Null given instead of a variable pointer.

Example

```
/* This example demonstrates setting transfer size. */  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */  
  
/* Declare the required variables. */  
STATUS status;  
SPI_APP_CALLBACKS callbacks;  
  
callbacks.spi_transfer_complete = Demo_SPI_Transfer_Complete;  
callbacks.spi_error = Demo_SPI_Error;  
  
/* Set the callback functions for slave 1. */  
status = NU_SPI_Master_Set_Callbacks (SPI_Dev_Handle, 1&callbacks);
```

```
/* At this point status indicates if callback functions were  
successfully changed. */
```

Related Topics

[NU_SPI_Master_Get_Callbacks](#)

[Miscellaneous](#)

NU_SPI_Master_Set_SS_Polarity

This function sets the slave-select polarity for the specified slave of the specified SPI device.

Usage

```
STATUS NU_SPI_Master_Set_SS_Polarity (SPI_HANDLE spi_dev,  
                                     UINT16      address,  
                                     UINT8       ss_polarity)
```

Arguments

- **spi_dev**
The specified SPI device.
- **address**
The specified slave device address.
- **ss_polarity**
The desired slave-select polarity.

Return Values

- **NU_SUCCESS**
Service completed successfully.
- **SPI_INVALID_HANDLE**
Nucleus SPI device handle is not valid.
- **SPI_INVALID_ADDRESS**
Specified slave address is not valid for the specified SPI device.
- **SPI_INVALID_SS_POLARITY**
Specified polarity is not valid.

Example

```
/* This example demonstrates setting transfer size. */  
SPI_HANDLE SPI_Dev_Handle;  
  
/* Assume that "SPI_Dev_Handle" contains device handle of an SPI device  
   which has previously been initialized with Nucleus SPI NU_SPI_Start  
   service call. */  
  
/* Declare the required variables. */  
STATUS status;  
  
/* Set the slave-select polarity for slave 2 to HIGH. */  
status = NU_SPI_Slave_Set_SS_Polarity (SPI_Dev_Handle, 2, SPI_SS_POL_HIGH);  
  
/* At this point status indicates if slave-select polarity was  
   successfully changed. */
```

Related Topics

[NU_SPI_Master_Set_Configuration](#)[Miscellaneous](#)

SPI Error Codes

Table 4-3 describes the error codes returned to the application by Nucleus SPI.

Table 4-3. Nucleus SPI Error Codes

Symbol	Value	Description
NU_SUCCESS	0	Successful completion.
SPI_OS_ERROR	1	Error occurred in underlying operating system kernel.
SPI_NULL_GIVEN_FOR_MEM_POOL	2	Memory pointer was not set properly and was pointing to null.
SPI_NULL_GIVEN_FOR_INIT	3	Initialization pointer is null.
SPI_REGISTRY_ERROR	4	System Registry Error
SPI_DEVICE_ERROR	5	SPI Device error
SPI_INVALID_PORT_ID	11	Invalid SPI driver port.
SPI_INVALID_DEVICE_ID	12	Invalid controller ID.
SPI_INVALID_HANDLE	13	Nucleus SPI device handle is invalid.
SPI_INVALID_HANDLE_POINTER	14	Specified pointer is not valid.
SPI_SHUTDOWN_ERROR	15	SPI controller could not be closed.
SPI_DEV_ALREADY_INIT	16	Nucleus SPI device is already initialized.
SPI_DRIVER_REGISTER_FAILED	17	Hardware driver registration with Nucleus SPI failed.
SPI_DRIVER_INIT_FAILED	18	The device initialization failed.
SPI_INVALID_DRIVER_MODE	19	Driver mode is not set to either interrupt driven or polling mode.
SPI_DRIVER_MODE_UNAVAILABLE	20	Specified driver mode is not available in current configuration.
SPI_INVALID_VECTOR	21	Invalid vector ID.
SPI_INVALID_PARAM_POINTER	22	Null given instead of a variable pointer.
SPI_INVALID_TRANSFER_LENGTH	23	Transfer length is zero.

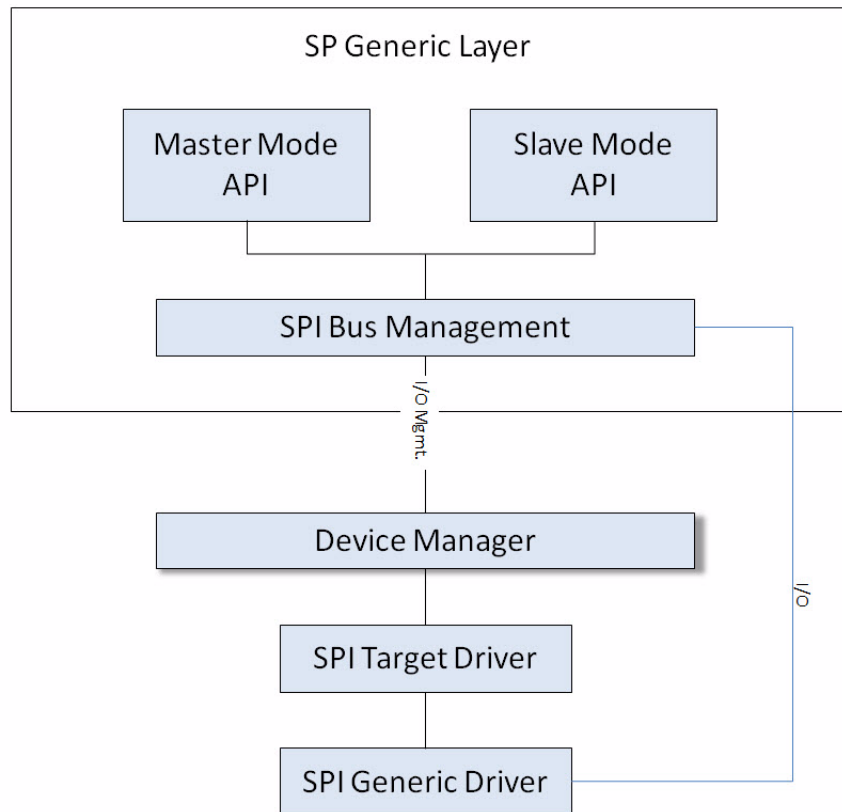
Table 4-3. Nucleus SPI Error Codes (cont.)

Symbol	Value	Description
SPI_INVALID_ADDRESS	24	Invalid slave address.
SPI_INVALID_MODE	25	Specified SPI mode is not valid.
SPI_INVALID_BIT_ORDER	26	Specified bit order is not valid.
SPI_ELEMENT_SIZE_NOT_ENOUGH	27	The buffer elements cannot hold data units of current transfer size.
SPI_INVALID_IOCTL_OPERATION	28	Unsupported operation requested for I/O control.
SPI_DEVICE_IN_USE	29	Nucleus SPI device is being used by some other user.
SPI_INVALID_SS_POLARITY	30	Invalid specified slave select polarity.
SPI_QUEUE_FULL	41	Data buffer is full.
SPI_BUFFER_NOT_ENOUGH	42	Buffer still has data.
SPI_INVALID_QUEUE_SIZE	43	More data was requested from buffer than available.
SPI_INVALID_BUFFER_SIZE	44	Transmission buffer size if zero.
SPI_GENERAL_HARDWARE_ERROR	101	An undocumented error occurred in SPI driver.
SPI_MASTER_MODE_NOT_SUPPORTED	102	Specified SPI device cannot function as an SPI master.
SPI_SLAVE_MODE_NOT_SUPPORTED	103	Specified SPI device cannot function as an SPI slave.
SPI_RX_OVERRUN	104	Rx overrun in SPI hardware.
SPI_TX_UNDERFLOW	105	Tx underflow occurred in the SPI slave.
SPI_UNSUPPORTED_BIT_ORDER	106	Specified bit order is not supported.
SPI_UNSUPPORTED_BAUD_RATE	107	Specified baud rate is not supported.
SPI_UNSUPPORTED_MODE	108	Specified SPI mode is not supported.
SPI_UNSUPPORTED_TRANSFER_SIZE	109	Specified transfer size is not supported.
SPI_UNSUPPORTED_SS_POLARITY	110	Specified slave-select polarity is not supported.

Lightweight SPI

Nucleus Lightweight SPI is designed to be easy to use. It provides a simple and well defined API for data I/O. The following block diagram gives an overview of Lightweight SPI.

Figure 4-4. Nucleus Lightweight SPI



The SPI generic layer consists of master and slave mode APIs and internal bus management functions. On top level there are API handlers. These APIs translate user requests into SPI driver calls for carrying I/O over SPI.

For master mode, bus management functions maintain a complete hierarchy of all busses (an SPI bus is synonymous to a SPI master controller) and their associated devices. For slave mode operations, bus hierarchy is not required but internal management of slave state machine.

For management operations like open, close and I/O control, the SPI generic layer communicates with underlying SPI driver using a standard device manager interface (open, close, and ioctl). I/O operations are an exception, these operations will bypass the device manager to achieve a better performance. SPI protocol specifics are contained in SPI driver, the generic layer provides only an abstraction for SPI target driver.

Note



Currently, Nucleus Lightweight SPI supports master mode only.

Theory of Operation

In master mode, upon discovering a new SPI master hardware controller, SPI generic layer will create a new SPI bus in the system. An application or a driver (using SPI) will register a slave device with SPI master. Slave registration requires the following configuration attributes about the device.

Attributes	Description		
Baud rate	Operational baud rate of slave device		
Slave select control	Defines whether slave select is application controlled or hardware controlled. 1 - Hardware controlled 2 - Application controlled		
Maximum transfer size	Maximum transfer size of device		
Mode	An SPI slave devices can be in one of four modes based on a combination of polarity and phase. Each slave device must specify one of the following modes.		
	Mode	CPOL	CPHA
	0	0	0
	1	0	1
	2	1	0
	3	1	1

The SPI generic layer will return a unique handle against each successfully registered slave device. The caller should save this handle and may use it to reference this device in the future.

Lightweight SPI Configuration

The metadata file for Nucleus Lightweight SPI is located at `os/connectivity/lwspi`. This metadata file defines the following configuration options:

- `num_spi_buses`

This option specifies the number of SPI buses in system. The value can range between from 1 to 10. The default value is 1.

- `num_spi_slaves`

This configures the number of SPI slaves per bus. The value can range from 1 to 255. The default value is 1.

- `int_mode_io_enable`

Use this configuration option to enable interrupt driven I/O. By default, the interrupt driven I/O is disabled.

- `err_check_enable`

Use this configuration option to enable error checking. Error checking is disabled by default.

Lightweight SPI Function Reference

This section describes services provided by Nucleus Lightweight SPI. These services can be divided into two categories: bus management services and I/O management services. This section provides details of the following APIs:

- [NU_SPI_Register_Slave](#)
- [NU_SPI_Unregister_Slave](#)
- [NU_SPI_Read](#)
- [NU_SPI_Write](#)
- [NU_SPI_Write_Read](#)

NU_SPI_Register_Slave

This API registers a new SPI slave device with an SPI bus. Applications must register each slave device with the respective bus before starting any I/O operations.

Usage

```
STATUS NU_SPI_Register_Slave( CHAR          *spi_bus,  
                             UINT32        baud_rate,  
                             UINT32        max_transfer_size,  
                             UINT32        mode_order_polarity,  
                             SPI_SS_CALLBACK ss_cb,  
                             NU_SPI_HANDLE *handle)
```

Arguments

- **spi_bus**
The SPI bus with which the device is associated. A SPI bus is identified by the “dev_name” option in the SPI device’s platform metadata file.
- **baud_rate**
The required operational baud rate of the device.
- **max_transfer_size**
The maximum supported transfer size of the device.
- **mode_order_polarity**
The logical or value for SPI mode, bit order, and slave select polarity.

SPI mode

The SPI transfer mode is a combination of phase and polarity supported by the device. Specify the mode according to the following table:

Mode	CPOL	CPHA
SPI_MODE_POL_LO_PHA_LO	0	0
SPI_MODE_POL_LO_PHA_HI	0	1
SPI_MODE_POL_HI_PHA_LO	1	0
SPI_MODE_POL_HI_PHA_HI	1	1

bit order

The bit order specifies which bit should be transferred first. To transfer the most significant bit (MSB) first, specify SPI_BO_MSB_FIRST. To transfer the least significant bit (LSB) first, specify SPI_BO_LSB_FIRST.

polarity

The slave select polarity specifies whether the slave select is active low or active high. To specify active low, use SPI_SS_POL_LO. To specify active high, use

SPI_SS_POL_HI. If the slave is driven by the software, specify SPI_SS_POL_DONT_CARE.

- **ss_cb**

A pointer to an application supplied slave select callback function. This is only valid if the application uses software driven slave select, otherwise it should be passed as NU_NULL.

- **handle**

This is an output argument containing a unique handle to the SPI slave device when the function returns.

Return Status

- **NU_SPI_BUS_SLOT_NOT_FOUND**

The specified SPI bus was not found.

- **NU_SPI_NO_FREE_SLAVE_SLOT**

There is no free slave slot available on SPI bus.

- **NU_SUCCESS**

The service completed successfully.

Related Topics

[Lightweight SPI Function Reference](#)

NU_SPI_Unregister_Slave

This function un-registers the slave device identified by NU_SPI_HANDLE. Once un-registered, no further I/O can take place with this slave device.

Usage

```
STATUS NU_SPI_Unregister_Slave(NU_SPI_HANDLE handle)
```

Arguments

- **handle**
The handle of the SPI slave session previously obtained through a call to [NU_SPI_Register_Slave](#) function.

Return Status

- **NU_SUCCESS**
The service completed successfully.
- **NU_SPI_INVLD_ARG**
Invalid handle.

Related Topics

[Lightweight SPI Function Reference](#)

NU_SPI_Read

This function reads data from a specified SPI device. The behavior of this function varies depending upon the value of the “io_type” argument.

Usage

```
STATUS NU_SPI_Read( NU_SPI_HANDLE handle,
                   UNSIGNED      io_type,
                   UINT8          *buffer,
                   UINT32         length)
```

Arguments

- **handle**
The handle to an already established session with an SPI device.
- **io_type**
The I/O type can be one of the following:
 - SPI_POLLED_IO**
Requested read operation uses poll-based I/O.
 - SPI_INTERRUPT_IO**
Requested read operation uses interrupt-driven I/O.
- **buffer**
A pointer to buffer where data is to be read.
- **length**
The length of data to be read from a SPI device.

Return Status

- **NU_SUCCESS**
The service completed successfully.

Description

If the caller specifies “io_type” as **SPI_POLLED_IO**, then the driver will complete the read operation in polling mode. This allows all data to be read within the calling context.

If the caller specifies “io_type” as **SPI_INTERRUPT_IO**, then the driver will complete the read operation in a interrupt-driven manner. Using this option, all data will be read within the context of SPI master controller driver HISR/LISR, but the API will suspend the calling context until the read operation finishes, hence completion of the read operation will still be reported synchronously.

This function will block until the read request is completed with success or until an error occurs during the transfer. If a read request is already in progress on the bus, then the call to this function will suspend the calling context until the bus is free and the transfer can progress.

Related Topics

[Lightweight SPI Function Reference](#)

NU_SPI_Write

This function writes data to a specified SPI device. The behavior of this function varies depending upon the value of “io_type” argument.

Usage

```
STATUS NU_SPI_Write( NU_SPI_HANDLE handle,
                    UNSIGNED      io_type,
                    UINT8         *buffer,
                    UINT32        length)
```

Arguments

- **handle**
The handle to an already established session with an SPI device.
- **io_type**
The I/O type can be one of the following:
 - SPI_POLLED_IO**
Requested read operation uses poll-based I/O.
 - SPI_INTERRUPT_IO**
Requested read operation uses interrupt-driven I/O.
- **buffer**
A pointer to buffer where data is to be read.
- **length**
The length of data to be read from a SPI device.

Return Status

- **NU_SUCCESS**
The service completed successfully.

Description

If the caller specifies “io_type” as **SPI_POLLED_IO**, then the driver will complete write operation in polling mode. This allows all data to be written within the calling context.

If the caller specifies “io_type” as **SPI_INTERRUPT_IO**, then the driver will complete the write operation in an interrupt-driven manner. Using this option, all data will be written within the context of the SPI master controller driver HISR, but the API will suspend the calling context until the write operation finishes, hence completion of write operation will still be reported synchronously.

This function will block until the write request is completed with success or an until error occurs during the transfer. If a write request is already in progress on the bus, then the call to this function will suspend the calling context until bus is free and transfer can progress.

Related Topics

[Lightweight SPI Function Reference](#)

NU_SPI_Write_Read

This function writes to specified SPI device and then reads from that device. The behavior of this function varies depending upon the value of “io_type” argument.

Usage

```
STATUS NU_SPI_Write_Read( NU_SPI_HANDLE handle,
                          UNSIGNED      io_type,
                          UINT8         *buffer_tx,
                          UINT8         *buffer_rx,
                          UINT32        io_length)
```

Arguments

- **handle**
The handle to an already established session with an SPI device.
- **io_type**
The I/O type can be one of the following:
 - SPI_POLLED_IO**
Requested read operation uses poll-based I/O.
 - SPI_INTERRUPT_IO**
Requested read operation uses interrupt-driven I/O.
- **buffer_tx**
A pointer to the buffer containing data to be written.
- **buffer_rx**
A pointer to the buffer containing the data to be read.
- **io_length**
The length of data to be exchanged from the SPI device.

Return Status

- **NU_SUCCESS**
The service completed successfully.

Description

If the caller passes “io_type” as **SPI_POLLED_IO**, then the driver will complete the I/O operation in polling mode. This allows all data will be transferred within the calling context.

If the caller passes “io_type” as **SPI_INTERRUPT_IO**, then the driver will complete the I/O operation in an interrupt-driven manner. Using this option, all data will be transferred within the context of SPI master controller driver HISR, but the API will suspend the calling context until

the I/O operation finishes, hence completion of the I/O operation will still be reported synchronously.

This function will block until the I/O request is completed with success or until an error occurs during the transfer. If an I/O request is already in progress on the bus, then the call to this function will suspend the calling context until the bus is free.

Related Topics

[Lightweight SPI Function Reference](#)

Chapter 5

Serial Driver

This chapter describes the Nucleus serial driver software module and the requirements for using it.

Caution



To guarantee future support and compatibility for your application, use only documented Nucleus interfaces, structures, macros, and so on.

Serial Driver Module Overview

Nucleus serial driver is a serial bus implementation that includes three sets of functions:

- **Driver Functions** — Provide the basic functions that must be present in a Nucleus serial driver. In an actual serial driver, additional functions are usually provided to increase the driver functionally and make it easy to use. Examples of these types of functions are provided in this chapter.
- **I/O Services (SIO) Functions** — Provide a common UART for SIO using an ANSI C style of character input and output. Four simple functions are available for interfacing with the UART and no pointer to the UART port is necessary. The SIO services are implemented as a layer over the Nucleus serial driver services.
- **Middleware Functions** — Provide a set of functions to open, close and use serial devices. The serial middleware abstracts all device-related interaction and provides buffering capability to improve performance.

To see examples of some the services provided by the Nucleus serial driver module, refer to [Nucleus Serial Driver Examples](#).

Driver Functions

[Table 5-1](#) summarizes the serial driver functions:

Table 5-1. Serial Driver Functions

Driver Function	Description
NU_SD_Data_Ready	Checks the receive buffer for characters.
NU_SD_Put_Char	Writes a character out to the serial port.
NU_SD_Put_String	Writes a null-terminated string out to the serial port.
NU_SD_Put_Stringn	Writes a n number of characters in a string out to the serial port.

Related Topics

[Serial Driver Module Overview](#)

NU_SD_Data_Ready

This function checks to see if there are any characters in the receive buffer. A status value is returned indicating if characters are present or not.

Usage

```
STATUS NU_SD_Data_Ready(SERIAL_SESSION *port)
```

Arguments

- port
Pointer to the port structure.

Return Values

- NU_TRUE
Indicates data in the receive buffer.
- NU_FALSE
Indicates no data in the receive buffer.

Example

```
/* Declare a new serial port */  
SERIAL_SESSION *port;  
INT ch;  
while (NU_SD_Data_Ready(port))  
{  
    ch = NU_Serial_Getchar(port);  
    NU_SD_Put_Char(ch, port, NU_FALSE)  
}
```

Related Topics

[Driver Functions](#)

NU_SD_Put_Char

This function writes a character out to the serial port.

Usage

```
INT NU_SD_Put_Char (UINT8          ch,  
                   SERIAL_SESSION *port,  
                   BOOLEAN        caller_putstring)
```

Arguments

- **ch**
The character to be sent to the port.
- **port**
Pointer to the port structure.
- **caller_putstring**
Flag to determine if the caller is from the Put_String.

Return Values

- **ch**
Character that was written to the serial port.
- **NU_EOF**
Write failed.

Description

A buffer test is performed to determine if there are any characters in the buffer that have not been transmitted. If there is room in the transmit buffer, the character is added to the buffer. Once a check for full buffer or buffer wrap has been performed, the next character in the transmit buffer is sent out the port.

Note



This service is not protected by a semaphore. Multiple threads could cause undesired results in the transmission of data. The use of the [NU_SD_Put_String](#) service is recommended.

Example

```
/* Declare a new serial port */  
SERIAL_SESSION *port;  
CHAR ch = 'A';  
  
/* Get Serial I/O port */  
port = NU_SIO_Get_Port();  
  
/* Call Serial Driver function */  
NU_SD_Put_Char(ch, port, NU_FALSE);
```


Related Topics

[Driver Functions](#)

NU_SD_Put_String

This function writes a null-terminated string out to the serial port.

Usage

```
INT NU_SD_Put_String (CHAR          str[],  
                     SERIAL_SESSION *port)
```

Arguments

- str
The string of characters to be sent to the port.
- port
Pointer to the port structure.

Return Values

- ch
Successfully transmitted the null-terminated string to the serial port.
- NU_EOF
Transmission failed.

Description

The function will obtain a semaphore for the specified port and will then call the function [NU_SD_Put_Char](#). Once the function has returned from `NU_SD_Put_Char` the entire string has been transmitted, the semaphore will be released.

Example

```
/* Declare a new serial port */  
SERIAL_SESSION* port;  
NU_SD_Put_String("Print this string of chars. \n\n\r", port);
```

Related Topics

[Driver Functions](#)

NU_SD_Put_Stringn

This function writes n number of characters in a string out to the serial port.

Usage

```
INT NU_SD_Put_Stringn (CHAR          str[],  
                      SERIAL_SESSION *port,  
                      UNSIGNED       count)
```

Arguments

- str[]
String to be written to the serial port.
- port
Serial port to send the string to.
- count
Number of bytes in a string to be written.

Return Values

- INT
Successfully transmitted n characters to the serial port.
- NU_EOF
Transmission failed.

Example

```
/* Declare a new serial port */  
SERIAL_SESSION *port;  
  
/* Get Serial I/O port */  
port = NU_SIO_Get_Port();  
  
/* Send the first 10 bytes of the string out to the serial port. */  
NU_SD_Put_Stringn("Only print the first 10 bytes of this string. \n\n\r",  
port, 10);
```

Related Topics

[Driver Functions](#)

I/O Services (SIO) Functions

[Table 5-2](#) summarizes the I/O Services (SIO) functions:

Table 5-2. Serial Driver SIO Functions

SIO Function	Description
NU_SIO_Get_Port	Returns a pointer to the port used by SIO.
NU_SIO_Getchar	Reads a character from the SIO port.
NU_SIO_Putchar	Transmits a character to the SIO port.
NU_SIO_Puts	Transmits a string to the SIO port.

Related Topics

[Serial Driver Module Overview](#)

NU_SIO_Get_Port

This function returns a pointer to the port used by SIO.

Usage

```
NU_SERIAL_PORT* NU_SIO_Get_Port (VOID)
```

Return Values

- **NU_SERIAL_PORT**
Pointer to the port structure.
- **NU_NULL**
SIO is not initialized.

Description

A test is performed to determine if SIO (NU_SIO_Initialized) is initialized. If initialized, a pointer to the port is returned, otherwise NU_NULL is returned.

This function is useful for applications that want to call the Serial Driver functions directly. See the [Serial Driver Module Overview](#) section of this chapter for more information on the three types of function sets.

Example

```
/* Pointer to the SIO serial port */
NU_SERIAL_PORT* port;

/* Get Serial I/O port */
port = NU_SIO_Get_Port();

/* Call Serial Driver function */
NU_SD_Put_String("Hello, World!\n", port);
```

Related Topics

[I/O Services \(SIO\) Functions](#)

NU_SIO_Getchar

This function is like the Standard I/O `getchar()` function where the serial data is read from STDIN. It cannot be used to read in binary data as End Of File character checking is performed on the data stream.

Usage

```
INT NU_SIO_Getchar (VOID)
```

Return Values

- `ch`
Character that was read from the SIO port.
- `NU_EOF`
End of the file or any other error was found.

Example

```
INT ch;  
  
/* Read a character from the SIO port */  
ch = NU_SIO_Getchar();
```

Related Topics

[I/O Services \(SIO\) Functions](#)

NU_SIO_Putchar

This function transmits a character to the SIO port. The function returns an integer (the value transmitted). A test is performed to determine if SIO (NU_SIO_Initialized) is initialized. If initialized, [NU_SD_Put_Char](#) is called. If serial I/O is not initialized, then NU_EOF is returned.

Usage

```
INT NU_SIO_Putchar (INT ch)
```

Arguments

- **ch**
Character for serial output.

Return Values

- **INT**
Character transmitted to the buffer.
- **NU_EOF**
Transmission failed.

Example

```
INT ch;  
  
ch = 'x';  
  
/* Write a character to the SIO port */  
NU_SIO_Putchar (ch);
```

Related Topics

[I/O Services \(SIO\) Functions](#)

NU_SIO_Puts

This function transmits a string to the SIO port. The function returns an integer. A test is performed to determine if serial I/O (NU_SIO_Initialized) is initialized. If initialized, [NU_SD_Put_String](#) is called. If SIO is not initialized, then NU_EOF (-1) is returned.

Usage

```
INT NU_SIO_Puts (const CHAR *s)
```

Arguments

- s
String for serial output.

Return Values

- Non-negative INT
Successfully transmitted the string to the SIO port.
- NU_EOF (-1)
Transmission failed.

Example

```
/* Write a string and carriage return to the SIO port */  
NU_SIO_Puts ("Hello, World!");
```

Related Topics

[I/O Services \(SIO\) Functions](#)

Middleware Functions

Table 5-3 summarizes the Middleware functions:

Table 5-3. Serial Driver Middleware Functions

Middleware Function	Description
NU_Serial_Close	Closes a serial device.
NU_Serial_Getchar	Reads a character from the selected serial port.
NU_Serial_Get_Configuration	Returns the attributes for the current serial port configuration.
NU_Serial_Get_Read_Mode	Returns the read mode configuration.
NU_Serial_Get_Write_Mode	Returns the write mode configuration.
NU_Serial_Open	Initializes a serial communications port.
NU_Serial_Putchar	Writes a character out to the serial port
NU_Serial_Puts	Writes a null-terminated string out to the serial port, obtains a semaphore for the specified port and calls the function NU_SD_Put_Char .
NU_Serial_Set_Configuration	Sets the new serial port configuration attributes.
NU_Serial_Set_Read_Mode	Sets the read mode configuration.
NU_Serial_Set_Write_Mode	Sets the write mode configuration.

Related Topics

[Serial Driver Module Overview](#)

NU_Serial_Close

This function closes a serial device.

Usage

```
STATUS NU_Serial_Close(SERIAL_SESSION *port)
```

Arguments

- port
Pointer to the serial port to be closed.

Return Values

- NU_SUCCESS
Successful completion of the service.
- NU_SERIAL_INVALID_SESSION
Invalid handle.
- Other error status
Returns the error status from one of the following APIs:

[NU_Obtain_Semaphore](#)

[DVC_Dev_Close](#)

[DVC_Dev_Ioctl](#)

Example

```
DV_DEV_LABEL  serial_device = {SERIAL_USART0_LABEL};  
SERIAL_SESSION port;  
  
/* Open a device */  
status = NU\_Serial\_Open(&serial_device, &port);  
  
/* Use the device; NU_Serial_Getchar/Putchar/Puts */  
...  
...  
  
/* Close the device */  
status = NU_Serial_Close(&port);
```

Related Topics

[Serial Demo](#)

[Middleware Functions](#)

NU_Serial_Getchar

This function reads a character from the selected serial port. It can be used to read in binary data as no special character checking (like End Of File) is performed on the data stream.

Note

This function replaced NU_SD_Get_Char.

Usage

```
INT NU_Serial_Getchar(SERIAL_SESSION *port)
```

Arguments

- port
Pointer to the serial port from which to get the character.

Return Values

- ch
Character that was read from the hardware.
- NU_SERIAL_SESSION_UNAVAILABLE
Port is NULL.
- NU_SERIAL_NO_CHAR_AVAIL
No characters are available from the selected serial port.
- Other error status

Returns the error status from one of the following APIs:

[DVC_Dev_Read](#)

[NU_Obtain_Semaphore](#)

[NU_Retrieve_Events](#)

Example

```
/* Serial device port */  
SERIAL_SESSION port;  
INT ch;  
ch = NU_Serial_Getchar(&port);
```

Related Topics

[Middleware Functions](#)

NU_Serial_Get_Configuration

This function returns the attributes for the current serial port configuration.

Usage

```
STATUS  NU_Serial_Get_Configuration (SERIAL_SESSION *port,  
                                     UINT32         *baud_rate,  
                                     UINT32         *tx_mode,  
                                     UINT32         *rx_mode,  
                                     UINT32         *data_bits,  
                                     UINT32         *stop_bits,  
                                     UINT32         *parity,  
                                     UINT32         *flow_ctrl)
```

Arguments

- port
Pointer to the serial session handle.
- baud_rate
Pointer to the return serial baud rate value.
- tx_mode
Pointer to the return serial Tx mode value.
- rx_mode
Pointer to the return serial Rx mode value.
- data_bits
Pointer to the return serial data bits value.
- stop_bits
Pointer to the return serial stop bits value.
- parity
Pointer to the return serial parity value.
- flow_ctrl
Pointer to the return serial flow control value.

Return Values

- NU_SUCCESS
Successful completion of the service.
- Other error status
Returns the error status from the following API:

[DVC_Dev_Ioctl](#)

Example

```
NU_SERIAL_PORT  *serial_port;
UINT32          baud_rate;
UINT32          tx_mode;
UINT32          rx_mode;
UINT32          data_bits;
UINT32          stop_bits;
UINT32          parity;
UINT32          flow_ctrl;

/* Retrieve STDIO serial port handle. */
serial_port = NU_SIO_Get_Port();

/* Retrieve the current serial port attributes. */
status = NU_Serial_Get_Configuration (serial_port, &baud_rate,
                                     &tx_mode, &rx_mode, &data_bits,
                                     &stop_bits, &parity, &flow_ctrl);
```

Related Topics

[Middleware Functions](#)

NU_Serial_Get_Read_Mode

This function returns the read mode configuration.

Usage

```
STATUS NU_Serial_Get_Read_Mode(SERIAL_SESSION *port,  
                               UNSIGNED         *mode)
```

Arguments

- **port**
Pointer to the serial port from which to read the mode configuration.
- **mode**
Pointer where mode is returned.

Return Values

- **NU_SUCCESS**
Successful completion of the service.
- **NU_SERIAL_SESSION_UNAVAILABLE**
Invalid handle.
- **NU_SERIAL_ERROR**
Device is in polled mode.

Related Topics

[Middleware Functions](#)

NU_Serial_Get_Write_Mode

This function returns the write mode configuration.

Usage

```
STATUS NU_Serial_Get_Write_Mode(SERIAL_SESSION *port,  
                                UNSIGNED         *mode)
```

Arguments

- port
Pointer to the serial port from which to write the mode configuration.
- mode
Pointer where mode is returned.

Return Values

- NU_SUCCESS
Successful completion of the service.
- NU_SERIAL_SESSION_UNAVAILABLE
Invalid handle.
- NU_SERIAL_ERROR
Device is in polled mode.

Related Topics

[Middleware Functions](#)

NU_Serial_Open

This function initializes a serial communications port. The function finds all devices with the specified label and opens, configures and enables the first available device.

Usage

```
STATUS NU_Serial_Open(DV_DEV_LABEL  *name,  
                      SERIAL_SESSION *port)
```

Arguments

- **name**
Pointer to a GUID of a serial device. You can set this to NULL to point to any arbitrary serial device.
- **port**
Pointer to the serial port to be initialized.

Return Values

- **NU_SUCCESS**
Successful completion of the service.
- **NU_SERIAL_DEV_NOT_FOUND**
Cannot find a device with passed-in name.
- **NU_SERIAL_SESSION_UNAVAILABLE**
Ran out of available sessions.
- **Other error status**
Returns the error status from one of the following APIs:

[NU_Obtain_Semaphore](#)

[NU_Register_LISR](#)

[DVC_Dev_ID_Open](#)

[DVC_Dev_ID_Get](#)

[DVC_Dev_Ioctl](#)

Example

```
/* This is some Serial device's label */  
DV_DEV_LABEL  serial_device = {SERIAL_USART0_LABEL};  
SERIAL_SESSION COM_Port_Session;  
STATUS        status;  
  
/* Open device */  
status = NU_Serial_Open(&serial_device, &COM_Port_Session);
```


Related Topics

[Serial Demo](#)

[Middleware Functions](#)

NU_Serial_Putchar

This function writes a character out to the serial port.

Usage

```
INT NU_Serial_Putchar(SERIAL_SESSION *port,  
                      INT             ch)
```

Arguments

- port
Pointer to the serial port where a character will be written.
- ch
The character to be sent to the port.

Return Values

- ch
Character that was transmitted.
- NU_EOF
Transmission failed.

Description

A buffer test is performed to determine if there are any characters in the buffer that have not been transmitted. If there is room in the transmit buffer, the character is added to the buffer. Once a check for full buffer or buffer wrap has been performed, the next character in the transmit buffer is sent out the port.

Note



This service is not protected by a semaphore. Multiple threads could cause undesirable results in the transmission of data. The [NU_Serial_Puts](#) service is recommended.

Example

```
/* Serial device port */  
SERIAL_SESSION port;  
INT ch = 'A';  
NU_Serial_Putchar (&port, ch);
```

Related Topics

[Middleware Functions](#)

NU_Serial_Puts

This function writes a null-terminated string out to the serial port, obtains a semaphore for the specified port and then calls the function [NU_SD_Put_Char](#). Once the `NU_SD_Put_Char` function indicates the string has been transmitted, the semaphore is released.

Usage

```
INT NU_Serial_Puts(SERIAL_SESSION *port,  
                  const CHAR      *str)
```

Arguments

- `port`
Pointer to the serial port where a null-terminated string will be written.
- `str`
The string of characters to be sent to the port.

Return Values

- Non-negative value
String was transmitted.
- `NU_EOF`
Transmission failed.

Example

```
/* Serial session port */  
SERIAL_SESSION port;  
NU_Serial_Puts(&port, "Print this string of chars. \n\n\r");
```

Related Topics

[Serial Demo](#)

[Middleware Functions](#)

NU_Serial_Set_Configuration

This function sets and configures the serial driver to work with the provided new serial port attributes.

Usage

```
STATUS  NU_Serial_Set_Configuration (SERIAL_SESSION *port,  
                                     UINT32         baud_rate,  
                                     UINT32         tx_mode,  
                                     UINT32         rx_mode,  
                                     UINT32         data_bits,  
                                     UINT32         stop_bits,  
                                     UINT32         parity,  
                                     UINT32         flow_ctrl)
```

Arguments

- **port**
Pointer to the serial session handle.
- **baud_rate**
New serial baud rate value.
- **tx_mode**
New serial Tx mode value.
- **rx_mode**
New serial Rx mode value.
- **data_bits**
New serial data bits value.
- **stop_bits**
New serial stop bits value.
- **parity**
New serial parity value.
- **flow_ctrl**
New serial flow control value.

Return Values

- **NU_SUCCESS**
Successful completion of the service.
- **Other error status**
Returns the error status from one of the following APIs:

[DVC_Dev_Ioctl](#)

Example

```
NU_SERIAL_PORT  *serial_port;
UINT32          baud_rate;
UINT32          tx_mode;
UINT32          rx_mode;
UINT32          data_bits;
UINT32          stop_bits;
UINT32          parity;
UINT32          flow_ctrl;

/* Retrieve STDIO serial port handle. */
serial_port = NU_SIO_Get_Port();

/* Retrieve the current serial port attributes. */
status = NU_Serial_Get_Configuration (serial_port, &baud_rate,
                                     &tx_mode, &rx_mode, &data_bits,
                                     &stop_bits, &parity, &flow_ctrl);

/* Set baud-rate to 115200. */
baud_rate = 115200;

/* Set new serial port attributes. */
status = NU_Serial_Set_Configuration (serial_port, baud_rate, tx_mode,
                                     rx_mode, data_bits, stop_bits,
                                     parity, flow_ctrl);
```

Related Topics

[Middleware Functions](#)

NU_Serial_Set_Read_Mode

This function sets the read mode configuration. Depending on the timeout, this service may block.

Usage

```
STATUS NU_Serial_Set_Read_Mode(SERIAL_SESSION *port,  
                               UNSIGNED mode,  
                               UNSIGNED timeout)
```

Arguments

- **port**
Pointer to the serial port where the read mode configuration will be set.
- **mode**
Specifies the read mode configuration.
- **timeout**
NU_NO_SUSPEND, NU_SUSPEND or (1-4,294,967,293)

Return Values

- **NU_SUCCESS**
Successful completion of the service.
- **NU_SERIAL_SESSION_UNAVAILABLE**
Invalid handle.
- **NU_SERIAL_ERROR**
Device is in polled mode.
- **NU_Obtain_Semaphore error status**
Returns the error status from [NU_Obtain_Semaphore](#).

Related Topics

[Middleware Functions](#)

NU_Serial_Set_Write_Mode

This function sets the write mode configuration. Depending on the timeout, this service may block.

Usage

```
STATUS NU_Serial_Set_Write_Mode(SERIAL_SESSION *port,  
                                UNSIGNED         mode,  
                                UNSIGNED         timeout)
```

Arguments

- **port**
Pointer to the serial port where the write mode configuration will be set.
- **mode**
Specifies the read write configuration.
- **timeout**
NU_NO_SUSPEND, NU_SUSPEND or (1-4,294,967,293)

Return Values

- **NU_SUCCESS**
Successful completion of the service.
- **NU_SERIAL_SESSION_UNAVAILABLE**
Invalid handle.
- **NU_SERIAL_ERROR**
Device is in polled mode.
- **NU_Obtain Semaphore error status**
Returns the error status from [NU_Obtain_Semaphore](#).

Related Topics

[Middleware Functions](#)

Nucleus Serial Driver Examples

The following examples demonstrate the functions provided by the Nucleus serial driver software module.

- [Serial Demo](#)

Related Topics

[Serial Driver Module Overview](#)

Serial Demo

This example demonstrates the “Nucleus Serial Demo” sample project.

Purpose & Goals

This sample application illustrates the use of Serial Middleware API's to use a UART device on the target. The simple application does the following:

- **Print_Time_Task:** Searches for available serial devices and prints a message to the STDIO interface.

Prerequisites

To get started with the Serial Demo project, create a new C project, select the Nucleus Serial Demo project, and read the README file included with the demo that is placed in your workspace.

Components Used

This application uses the following components:

- nu.os.kern.plus
- nu.os.kern.devmgr
- nu.os.svcs.appinit
- nu.os.drvr.serial

API Reference

This example uses the following services:

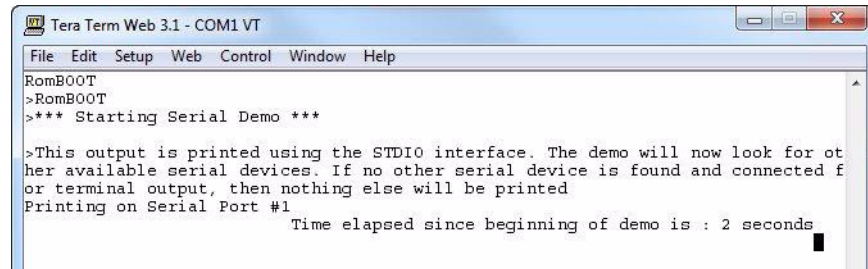
Table 5-4. Serial Demo Services

Service	Function Using Service
NU_Allocate_Memory (<i>Nucleus Kernel Guide</i>)	Application_Initialize
NU_Create_Task (<i>Nucleus Kernel Guide</i>)	Application_Initialize
NU_Serial_Open	Print_Time_Task
NU_Serial_Puts	Print_Time_Task
NU_Serial_Close	Print_Time_Task
NU_Sleep (<i>Nucleus Kernel Guide</i>)	Print_Time_Task
NU_Retrieve_Clock (<i>Nucleus Kernel Guide</i>)	Print_Time_Task

Results

[Figure 5-1](#) shows the output of this example viewed in a serial terminal (like TeraTerm) while running on a hardware platform.

Figure 5-1. Serial Demo Output



The screenshot shows a web-based terminal window titled "Tera Term Web 3.1 - COM1 VT". The window has a menu bar with "File", "Edit", "Setup", "Web", "Control", "Window", and "Help". The terminal output is as follows:

```
RomBOOT
>RomBOOT
>*** Starting Serial Demo ***

>This output is printed using the STDIO interface. The demo will now look for ot
her available serial devices. If no other serial device is found and connected f
or terminal output, then nothing else will be printed
Printing on Serial Port #1
Time elapsed since beginning of demo is : 2 seconds
```

Related Topics

[Nucleus Serial Driver Examples](#)

Chapter 6

DMA Device Driver

This chapter describes the Nucleus Direct Memory Access (DMA) Device Driver software module and the requirements for using it.

Caution



To guarantee future support and compatibility for your application, use only documented Nucleus interfaces, structures, macros, and so on.

DMA Device Driver Module Overview

The Nucleus DMA device driver provides the interface to the DMA controller. The driver supports the following data transfers:

- Memory to memory
- Peripheral to memory
- Memory to peripheral

The DMA Device Driver also supports multiple DMA devices, multiple channels per DMA device and multiple users per channel. Data transfers can be made synchronously or asynchronously.

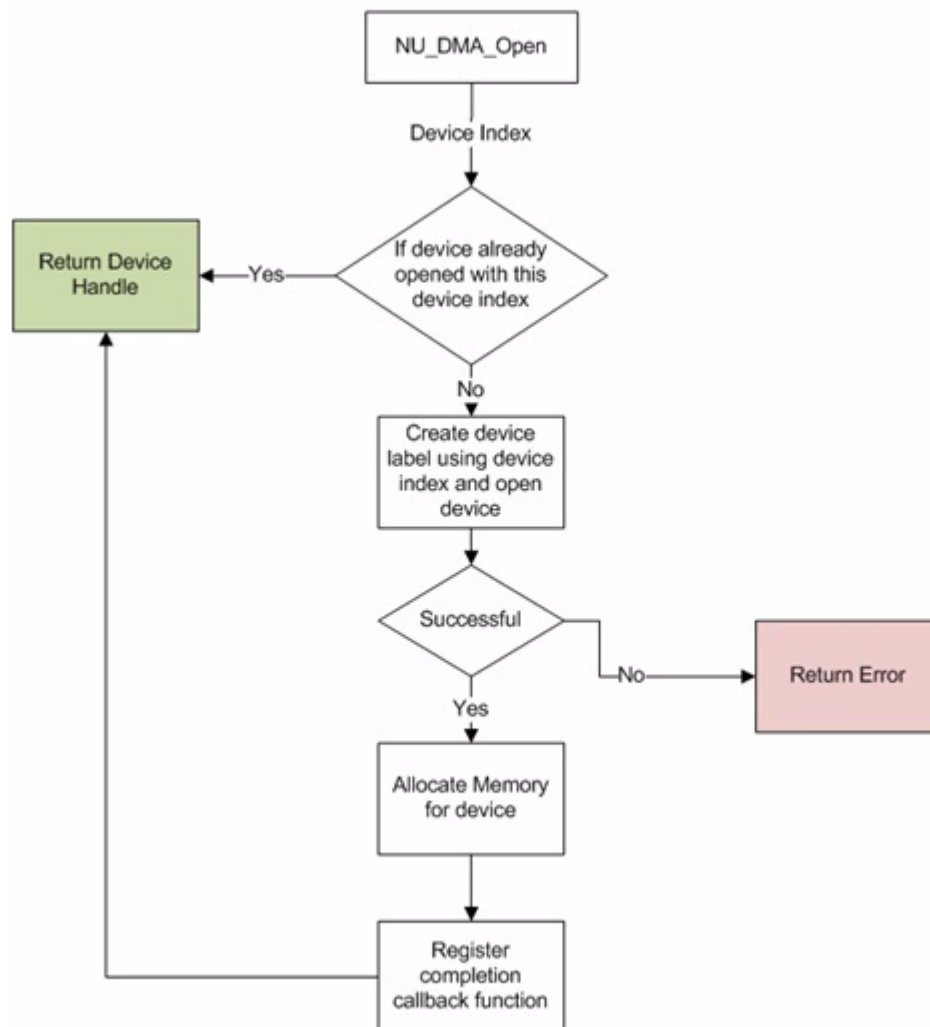
DMA Device Driver Operation

This section describes the major operations performed by the DMA driver, including [DMA Initialization](#), [DMA Channel Acquisition](#), and [DMA Data Transfer](#).

DMA Initialization

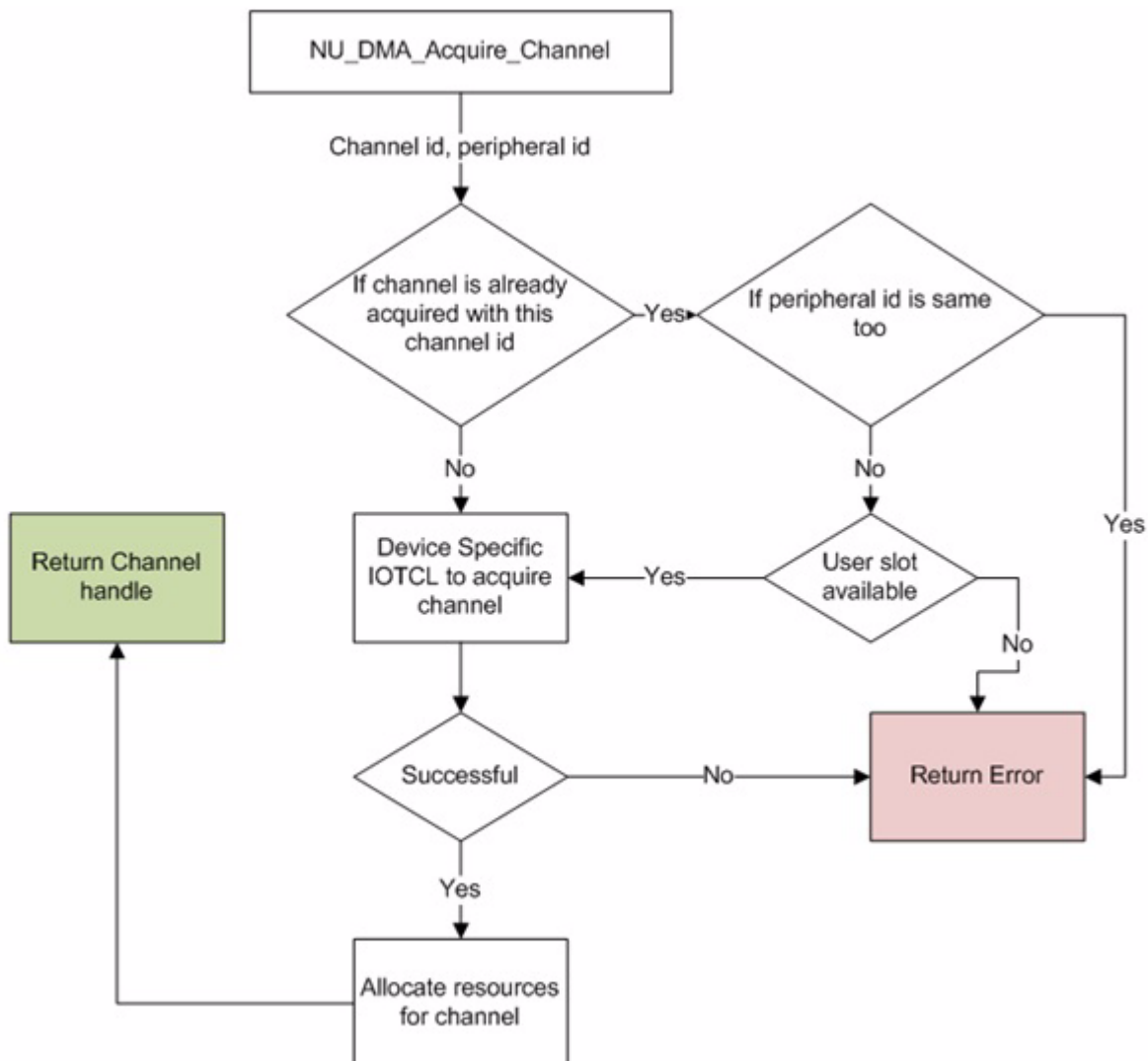
The DMA driver initialization is performed by the [NU_DMA_Open](#) API. The API is responsible for opening the DMA driver against a certain DMA device ID. It initializes the internal DMA driver structures, creates event handlers, registers completion call-backs and makes calls to the target specific layer for hardware initialization. [Figure 6-1](#) shows the process flow for DMA initialization.

Figure 6-1. DMA Initialization Process Flow



DMA Channel Acquisition

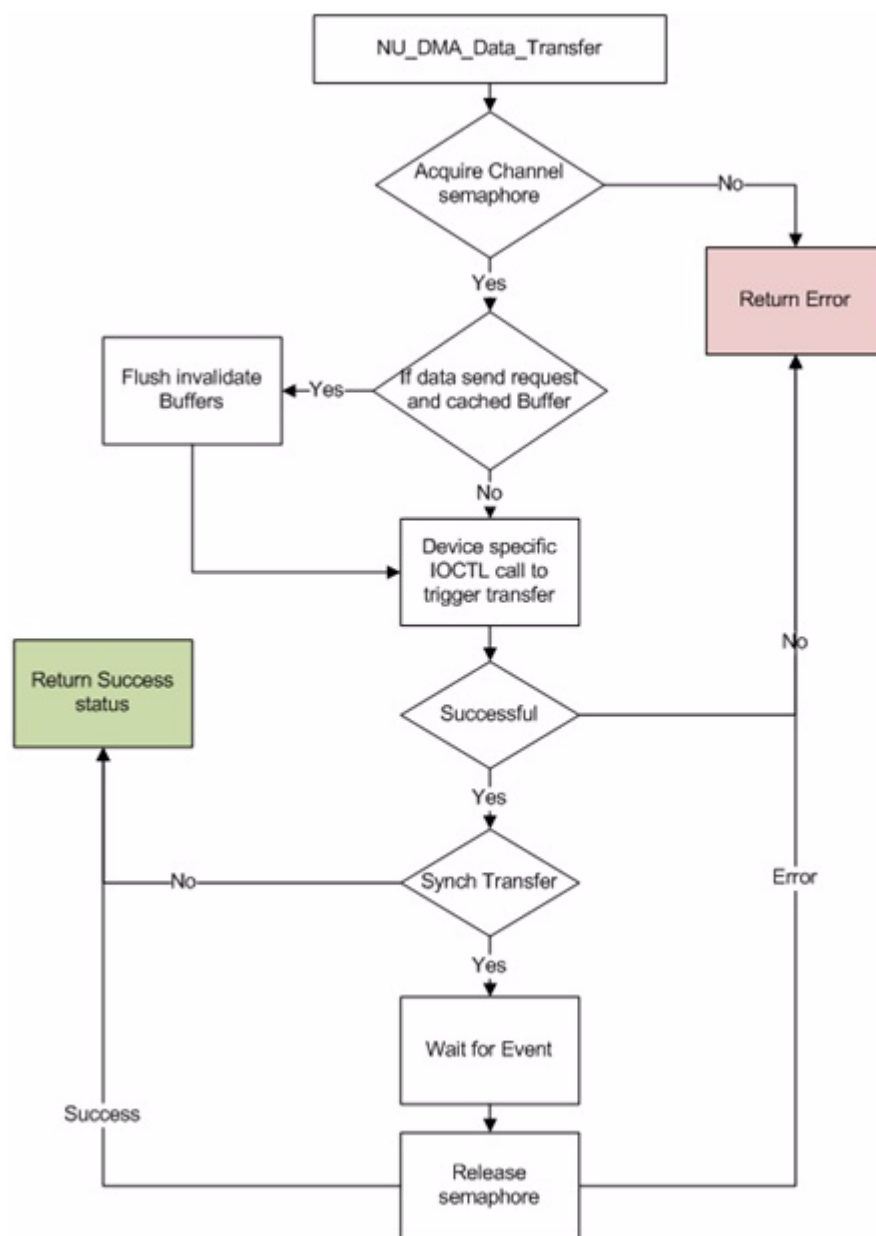
The DMA channel acquisition operation performed by API `NU_DMA_Acquire_Channel` acquires a channel for a certain peripheral. This operation allows multiple users (each with its own peripheral ID) to share a hardware DMA channel. Figure 6-2 shows the process flow for DMA channel acquisition.

Figure 6-2. DMA Channel Acquisition Process Flow

DMA Data Transfer

The DMA driver API `NU_DMA_Data_Transfer` is responsible for triggering a data transfer on an already acquired channel. This API handles cached and un-cached buffers, calling target specific functions to configure the hardware as per the transfer type attributes provided to the API. Figure 6-3 shows the process flow for DMA data transfers.

Figure 6-3. DMA Data Transfer Process Flow



DMA Device Driver Data Structures

This section describes the following data structures associated with the Nucleus [DMA Device Driver APIs](#):

- [DMA_ADDRESS_TYPE](#)
- [DMA_REQUEST_TYPE](#)
- [DMA_REQ](#)

DMA_ADDRESS_TYPE

DMA_ADDRESS_TYPE is an enum typedef data structure that defines the DMA peripheral address types.

Structure Definition

```
typedef enum
{
    DMA_ADDRESS_INCR,
    DMA_ADDRESS_DECR,
    DMA_ADDRESS_FIXED,
} DMA_ADDRESS_TYPE;
```

The members of the structure are defined in [Table 6-1](#).

Table 6-1. DMA_ADDRESS_TYPE

Member	Description
DMA_ADDRESS_INCR	Incremental DMA address type
DMA_ADDRESS_DECR	Decremental DMA address type
DMA_ADDRESS_FIXED	Fixed (FIFO based) DMA address type

Related Topics

[NU_DMA_Acquire_Channel](#)

[DMA Device Driver APIs](#)

DMA_REQUEST_TYPE

DMA_REQUEST_TYPE is an enum typedef data structure that defines the DMA transfer types.

Structure Definition

```
typedef enum
{
    DMA_FREE,
    DMA_SYNC_SEND,
```

```
DMA_SYNC_RECEIVE,  
DMA_ASYNC_SEND,  
DMA_ASYNC_RECEIVE,  
DMA_SYNC_MEM_TRANS,  
DMA_ASYNC_MEM_TRANS  
} DMA_REQUEST_TYPE;
```

The members of the structure are defined in [Table 6-2](#).

Table 6-2. DMA_REQUEST_TYPE

Member	Description
DMA_FREE	Reserved define for transfers not falling into any other category. For future usage.
DMA_SYNC_SEND	Synchronous memory to peripheral transfers.
DMA_SYNC_RECEIVE	Synchronous peripheral to memory transfers.
DMA_ASYNC_SEND	Asynchronous memory to peripheral transfers.
DMA_ASYNC_RECEIVE	Asynchronous peripheral to memory transfers.
DMA_SYNC_MEM_TRANS	Synchronous memory to memory transfers.
DMA_ASYNC_MEM_TRANS	Asynchronous memory to memory transfers.

Related Topics

[NU_DMA_Data_Transfer](#)

[DMA Device Driver APIs](#)

DMA_REQ

DMA_REQ is a typedef name for the `_dma_req_struct` data structure. This data structure defines the DMA request structure for data transfers.

Structure Definition

```
typedef struct _dma_req_struct  
{  
    VOID *src_ptr;  
    VOID *dst_ptr;  
    UINT32 length;  
    VOID *req_reserve;  
} DMA_REQ;
```

The members of the structure are defined in [Table 6-3](#).

Table 6-3. DMA_REQ

Member	Description
src_ptr	Pointer to the source buffer.
dst_ptr	Pointer to the destination buffer.

Table 6-3. DMA_REQ

Member	Description
length	Size of the data transfer.
req_reserve	Reserved field for handling hardware specific requirements.

Related Topics

[NU_DMA_Acquire_Channel](#)

[NU_DMA_Data_Transfer](#)

[DMA Device Driver APIs](#)

DMA Device Driver APIs

Table 6-4 summarizes the DMA Device Driver APIs:

Table 6-4. DMA Device Driver APIs

Driver Function	Description
NU_DMA_Open	This function initializes the DMA driver.
NU_DMA_Close	This function closes the DMA device.
NU_DMA_Acquire_Channel	This function acquires the specified DMA hardware channel.
NU_DMA_Release_Channel	This function releases the specified DMA channel.
NU_DMA_Reset_Channel	This function resets the specified DMA channel.
NU_DMA_Data_Transfer	This function triggers a DMA data transfer.

Related Topics

[DMA Device Driver Data Structures](#)

NU_DMA_Open

This function is used to open the DMA device. The device can be opened by multiple users.

Usage

```
STATUS NU_DMA_Open(UINT8          dma_dev_id,  
                   DMA_DEVICE_HANDLE *dma_handle_ptr)
```

Arguments

- **dma_dev_id**
The ID specific the DMA device to be opened. It is used to distinguish between multiple DMA devices. The device id is specified in the DMA device registry portion of the platform file.
- **dma_handle_ptr**
Pointer to the location where the device handle will be returned.

Return Values

- **NU_SUCCESS**
Open request was completed successfully.
- **NU_INVALID_POINTER**
Invalid pointer.
- **NU_DMA_DEVICE_NOT_FREE**
No DMA device control block is free.

Example

```
STATUS status;  
DMA_DEVICE_HANDLE dma_handle;  
  
/* Open DMA device. */  
status = NU_DMA_Open(0, &dma_handle);
```

Related Topics

[NU_DMA_Close](#)

NU_DMA_Close

This function is used to close the DMA device.

Usage

```
STATUS NU_DMA_Close(DMA_DEVICE_HANDLE dma_handle)
```

Arguments

- `dma_handle`
DMA device handle.

Return Values

- `NU_SUCCESS`
Close request was completed successfully.
- `NU_DMA_INVALID_HANDLE`
Device handle is not valid.

Example

```
STATUS status;  
DMA_DEVICE_HANDLE dma_handle;  
...  
/* Close DMA device. */  
status = NU_DMA_Close(dma_handle);
```

Related Topics

[NU_DMA_Open](#)

NU_DMA_Acquire_Channel

This function is used to acquire the DMA channel. It uses the hardware channel ID to distinguish between different DMA channels and maps the channel ID to the actual DMA controller channel number. Multiple users can open the same DMA channel by using different peripheral IDs associated with that channel.

Usage

```
STATUS NU_DMA_Acquire_Channel(DMA_DEVICE_HANDLE dma_handle,  
                             DMA_CHAN_HANDLE  *chan_handle_ptr,  
                             UINT16          hw_chan_id,  
                             UINT16          peri_id,  
                             DMA_ADDRESS_TYPE add_type,  
                             UINT16          burst_size,  
                             VOID (*completion_callback)  
                             ((DMA_CHAN_HANDLE, DMA_REQ*, UINT32, STATUS))
```

Arguments

- **dma_handle**
DMA device handle.
- **chan_handle_ptr**
Pointer to the location where the channel handle is returned.
- **hw_chan_id**
Hardware channel id. The channel id is mapped on the actual DMA controller channel.
- **peri_id**
Peripheral associated with the channel. The peripheral id can be logical or physical. Multiple users can access the same channel with different peripheral ids.
- **add_type**
Peripheral address type. The address type can be `DMA_ADDRESS_FIXED` for FIFO, `DMA_ADDRESS_INCR` for incrementing addresses, or `DMA_ADDRESS_DECR` for decrementing addresses.

Data structure [DMA_ADDRESS_TYPE](#) is defined in the [DMA Device Driver Data Structures](#) section of this chapter.
- **burst_size**
DMA transfer burst size in bytes.
- **completion_callback**
Completion callback function pointer. The callback function is triggered in asynchronous data transfer mode either on successful completion of the data transfer with `NU_SUCCESS` or if the transfer was unsuccessful and an error code was received.

Return Values

- **NU_SUCCESS**
Request completed successfully.
- **NU_DMA_INVALID_HANDLE**
Device handle is not valid.
- **NU_DMA_CHANNEL_ALREADY_OPEN**
Channel is already open with the same channel and peripheral id.
- **NU_DMA_CHANNEL_MAX_REACHED**
Maximum channel limit has been reached.
- **NU_DMA_CHANNEL_MAX_USER_REACHED**
Maximum limit of users per channel has been reached.

Example

```
STATUS status;
DMA_DEVICE_HANDLE dma_handle;
DMA_CHAN_HANDLE chan_handle;
UINT8 hw_chan_id = 7;
UINT8 peri_id = 0;
UINT16 burst_size = 4;

/* Open Device. */
...
/* Acquire channel. */
status = NU_DMA_Acquire_Channel(dma_handle, &chan_handle,
                                hw_chan_id, peri_id,
                                DMA_ADDRESS_INCR, burst_size,
                                comp_callback);
...
/* Transfer completion callback function. */
VOID comp_callback(DMA_CHAN_HANDLE chan_handle, DMA_REQ *req,
                  UINT32 length, STATUS status)
```

Related Topics

[NU_DMA_Release_Channel](#)

[DMA Device Driver Data Structures](#)

[DMA_REQ](#)

[DMA_ADDRESS_TYPE](#)

NU_DMA_Release_Channel

This function is used to release the DMA channel.

Usage

```
STATUS NU_DMA_Release_Channel (DMA_CHAN_HANDLE chan_handle)
```

Arguments

- `chan_handle`
DMA channel handle.

Return Values

- `NU_SUCCESS`
Request completed successfully.
- `NU_DMA_INVALID_HANDLE`
Channel handle is not valid.

Example

```
STATUS status;  
DMA_CHAN_HANDLE chan_handle ;  
.....  
/* Release channel. */  
status = NU_DMA_Release_Channel (chan_handle);
```

Related Topics

[NU_DMA_Acquire_Channel](#)

NU_DMA_Reset_Channel

This function is used to reset the DMA channel. It can be used in case of some error on the channel.

Usage

```
STATUS NU_DMA_Reset_Channel (DMA_CHAN_HANDLE chan_handle)
```

Arguments

- `chan_handle`
DMA channel handle.

Return Values

- `NU_SUCCESS`
Request completed successfully.
- `NU_DMA_INVALID_HANDLE`
Channel handle is not valid.

Example

```
STATUS status;  
DMA_CHAN_HANDLE chan_handle ;  
.....  
/* Reset channel. */  
status = NU_DMA_Reset_Channel (chan_handle);
```

Related Topics

[NU_DMA_Acquire_Channel](#)

NU_DMA_Data_Transfer

This function is used to trigger the data transfer on the DMA channel. Data transfer can be synchronous or asynchronous. This function call can be blocking or non-blocking.

Usage

```
STATUS NU_DMA_Data_Transfer(DMA_CHAN_HANDLE chan_handle,  
                            DMA_REQ * dma_req_ptr,  
                            UINT32 total_requests,  
                            UINT8 is_cached,  
                            DMA_REQUEST_TYPE req_type,  
                            UNSIGNED suspend)
```

Arguments

- **chan_handle**
DMA channel handle.
- **dma_req_ptr**
Pointer to the DMA request array. A DMA request contains the source address, destination address and data transfer length. It can be either a pointer to a single request element or a request array containing multiple transfer requests.
Data structure [DMA_REQ](#) is defined in the [DMA Device Driver Data Structures](#) section of this chapter.
- **total_requests**
Number of elements in the transfer request array.
- **is_cached**
Specifies either the data buffers are cached or uncached. A NU_TRUE value indicates the buffers are cached; NU_FALSE indicates the buffers are uncached. If the argument is set to NU_TRUE the DMA driver flushes the cache and invalidates the buffers; otherwise, the user will handle the buffers according to their requirement.
- **req_type**
Specifies the DMA request type.
Data structure [DMA_REQUEST_TYPE](#) is defined in the [DMA Device Driver Data Structures](#) section of this chapter.
- **Suspend**
Following are the suspension options:
 - NU_NO_SUSPEND: Return if transfer can be completed immediately.
 - NU_SUSPEND: Infinite suspension.
 - Any other value: Specifies the suspension timeout in ticks.

Return Values

- **NU_SUCCESS**
Request completed successfully.
- **NU_DMA_INVALID_HANDLE**
Channel handle is not valid.
- **NU_DMA_DRIVER_ERROR**
Some error from underlying driver.
- **NU_TIMEOUT**
Timeout on service.

Example

```
STATUS status;
DMA_DEVICE_HANDLE dma_handle;
DMA_CHAN_HANDLE chan_handle;
DMA_REQ dma_req;

/* Open Device. */
.....

/* Acquire Channel. */
.....
dma_req.src_ptr = src_addr;
dma_req.dst_ptr = dst_addr;
dma_req.length= length;

/* Start data transfer. */
status = NU_DMA_Data_Transfer(chan_handle, &dma_req, 1,
                             NU_TRUE, DMA_SYNC_MEM_TRANS, NU_SUSPEND);
```

Related Topics

[DMA_REQUEST_TYPE](#)

[DMA_REQ](#)

[DMA Device Driver Data Structures](#)

Chapter 7

USB Communications

This chapter describes the Nucleus USB communication class driver module and the requirements for using it.

Caution



To guarantee future support and compatibility for your application, use only documented Nucleus interfaces, structures, macros, and so on.

USB Communication Class Module Overview

Nucleus USB communications class software module is an implementation of the middleware necessary to develop a USB communication driver or a communications application. This middleware is implemented in accordance with the *Universal Serial Bus Class Definitions for Communication Devices* specification. All models of communication devices have some part common between them related to USB and communication class.

The USB communications class driver module contains all the components needed to develop a communication user driver or a communication application as follows:

- **Host Component**— For more information, see [“USB Host Component”](#) on page 267.
- **Function Component** — For more information, see [“USB Function Component”](#) on page 319.

USB Host Component

The Nucleus USB Host Communications Class Driver enables support for all communication device models. The current version has implementation for ACM and ECM devices such as abstract control model and Ethernet control model, which are the most widely used type of communication device models in embedded applications. All different models of communication devices have some part common between them related to USB and communication class. The model specific part is due to the different nature of data I/O between communication devices the USB host and the communication channels.

Nucleus USB Host Communications Class Driver implements both common class and model specific functionality while providing the application with a simplified USB independent API. Then the communication user driver acts as the middleware, which only implements the partial or complete modular behavior of a communication device. Common class API and model specific API of a device model should not be changed with the introduction of any of new

device models. Then the communication user driver acts as the middleware, which only implements the partial or complete modular behavior of a communication device. Common class API and model specific API of a device model should not be changed with the introduction of any of new device models.

Host Component Control Block

Like all other service components in Nucleus USB Host, the Nucleus USB Host Communications Class Driver is also identified in the system by its control block. This control block is named as `NU_USBH_COM`.

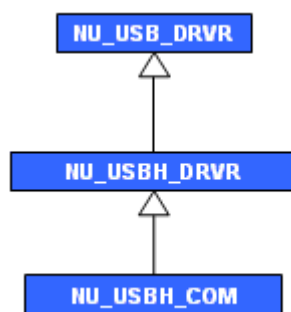
Host Component Initialization

`NU_USBH_COM` is created and initialized in the system by the application using the API call, [NU_USBH_COM_Create](#), by passing a reference, a name, a memory pool and pointer to host stack control block. The communication driver uses this memory pool internally to store information connected to communication devices and other bookkeeping needs. The stack pointer registers itself as an interface with the host stack.

Host Component Hierarchy

[Figure 7-1](#) shows the component hierarchy diagram of the Nucleus USB Host Communications Class Driver. The Nucleus USB Host Communications Class Driver is a specialization of the base host class driver. It implements common class driver functionality, unique to the USB communication class. Additional services required to access communication devices are also provided.

Figure 7-1. Host Component Hierarchy



Due to this inheritance philosophy, Nucleus USB Host stack invokes base class driver API services on references to communication [extended] class driver control blocks. The stack obtains communication [extended] class driver control blocks through registration; the stack uses them as normal references to [base] class driver control blocks.

Host Comm Class Match Spec

According to the *USB Communication Class Specification*, communication class code is defined in the interface descriptor. Consequently, the Nucleus USB Host Communications Class Driver is an interface driver and owns an interface on a device. But all USB communication devices have two interfaces, which are communication and data interface. Communication interface is used for control operations and event report while data interface is used for sending and receiving data packets. Since Nucleus USB Host Communication Class Driver is an interface driver it is made to own the communication interface by application while in addition to this the data interface is owned by the Nucleus USB Host Communications Class Driver itself.

The Nucleus USB Host Communications Class Driver is an interface level driver. As stated earlier, this class driver handles all of the defined device models on the basis of subclass. Therefore, it only provides bInterfaceClass in match spec. The class code for communication class is 0x02. For more information about match spec, refer to “[Function Component Match Spec](#)”.

The following is a code example from the initialization routine of the communication class created, providing the match spec and dispatch tables are included as parameters to the function.

```
status = _NU_USBH_DRV_CREATE(
    /* pointer to control block of communication class host driver */
    (NU_USB_DRV *) pcb_com_drvr,

    /* name for driver component */
    "USBH_CD",

    /* match_flag instruct to create it as a class driver only*/
    USB_MATCH_CLASS,

    /* not a device drv so all related fields are 0 */
    0, 0, 0, 0,

    /* USB communication class code */
    0x02,

    /* covers all subclasses and protocols of USB communication class */
    0, 0,

    /* pointer to dispatch table of communication class host driver */
    &USBh_com_dispatch
);
status = _NU_USBH_DRV_CREATE(
    /* pointer to control block of communication class host's data
       driver */
    (NU_USB_DRV *) pcb_com_drvr->pcb_data_drvr,

    /* name for driver component */
    "USBH_DD",

    /* match_flag instruct to create it as a class driver only*/
    USB_MATCH_CLASS,
```

```
/* not a device driver so all related fields are 0 */
0, 0, 0, 0,

/* USB communication class's data interface code */
0x0A,

/* covers all subclasses and protocols of USB communication class */
0, 0,

/* pointer to dispatch table of communication class host driver */
&usbh_com_dispatch
);
```

Communications Class Function Reference

Nucleus USB Host Communications Class Driver has two interfaces, one with a lower layer or Nucleus USB Host Stack, the other is with an upper layer known as a user or application.

This section lists the user, or application, functions.

Note



These functions may lead to the suspension of the calling task momentarily, therefore exercise care when calling these services in an ISR context.

- [NU_USBH_COM_Clear_Comm_Feature](#)
- [NU_USBH_COM_Create](#)
- [NU_USBH_COM_Data_Transfer](#)
- [NU_USBH_COM_Get_Comm_Feature](#)
- [NU_USBH_COM_Get_Encap_Resp](#)
- [NU_USBH_COM_Send_Encap_Cmd](#)
- [NU_USBH_COM_Set_Comm_Feature](#)
- [_NU_USBH_COM_Delete](#)

NU_USBH_COM_Clear_Comm_Feature

The user driver calls this function to clear a specific feature at the attached communication device.

Usage

```
STATUS NU_USBH_COM_Clear_Comm_Feature (NU_USBH_COM_DEVICE *pcb_curr_dev,  
                                       UINT16 feature_sel)
```

Arguments

- `pcb_curr_dev`
Pointer to the control block of the communication device.
- `feature_sel`
Feature selector.

Return Values

- `NU_SUCCESS`
Indicates successful completion.
- `NU_USB_INVLD_ARG`
Indicates a control block is deleted before completion.
- `NU_USB_COM_XFER_ERR`
Indicates data transfer does not complete successfully due to internal error.
- `NU_USB_COM_XFER_FAILED`
Indicates command failed by the communication.

Example

```
/* Calling API function to send cmd data assuming that user  
 * and communication driver have been already created and an  
 * attached communication device has been reported to user  
 * driver */  
  
NU_USB_COM_Clear_Comm_Feature (&global_cb_dev, 0x1234);
```

Related Topics

[Communications Class Function Reference](#)

NU_USBH_COM_Create

This function initializes the Nucleus USB Host Communications Class Driver and registers it with a host stack automatically.

Usage

```
STATUS NU_USBH_COM_Create (NU_USBH_COM    *pcb_com_drvr,  
                           CHAR            *p_name,  
                           NU_MEMORY_POOL *p_memory_pool,  
                           NU_USB_STACK   *usb_stack)
```

Arguments

- **pcb_com_drvr**
Pointer to the control block of the communication driver.
- **p_name**
Pointer to the start of the USB object's name.
- **p_memory_pool**
Starting in Nucleus 3.2, this argument is no longer required and is present only for backward compatibility. NU_NULL should be passed in for p_memory_pool.
- **usb_stack**
Pointer to the USB stack.

Return Values

- **NU_SUCCESS**
Successful initialization.
- **NU_INVALID_SEMAPHORE**
Indicates the semaphore pointer is invalid.
- **DELETED**
The semaphore was deleted while the task was suspended.
- **NU_UNAVAILABLE**
Indicates the semaphore is unavailable.
- **NU_INVALID_SUSPEND**
Indicates that this API is called from a non-task thread.

Description

This function is designed for use by system integrators to create communication class driver components on the host subsystem. The data interface driver for the communication device is internally created and registered with the host stack.

Example

```
NU_USBH_COM    cb_comm_drvr;
NU_MEMORY_POOL com_memory_pool;
STATUS         status;

/* Creating memory pool to used by communication driver */
status = NU_Create_Memory_Pool (&com_memory_pool, "COMMEM",
                                first_available_memory, 6*1024*1024, 50,
                                NU_FIFO);

if (status != NU_SUCCESS) /* Exception Handler */
{
    Application_Error_Handler (status);
}

/* Calling API function to create communication class host driver */
status = NU_USBH_COM_Create(&cb_com_drvr, name, &com_memory_pool,
                            usbh_stack); /* already created stack */

if (status != NU_SUCCESS) /* Exception Handler */
{
    Application_Error_Handler (status);
}
```

Related Topics

[Communications Class Function Reference](#)

NU_USBH_COM_Data_Transfer

This function transfers data across USB to and from a communication device.

Usage

```
STATUS NU_USBH_COM_Data_Transfer (NU_USBH_COM_DEVICE *pcb_curr_device,  

                                  NU_USBH_COM_XBLOCK *pcb_xblock)
```

Arguments

- **pcb_curr_device**
 Pointer to the control block of the target communication device.
- **pcb_xblock**
 Pointer to the communication data transfer control block.

Return Values

- **NU_SUCCESS**
 Indicates successful completion.
- **NU_USBH_COM_XFER_ERR**
 Indicates data transfer does not complete successfully due to internal error.
- **NU_USBH_COM_PEMPT**
 Indicates paper empty status at communication.
- **NU_USBH_COM_XFER_FAILED**
 Indicates command failed by the communication.

NU_USBH_COM_XBLOCK

Table 7-1. NU_USBH_COM_XBLOCK

Data Type	Element Name	Description
VOID*	p_data_buf	Pointer to data buffer.
UINT32	data_length	Buffer length in bytes.
UINT32	transfer_length	Variable to hold actual amount of data transferred or received. In some cases the actual length received over USB is less than what was specified in data_length by the user. On successful return this variable holds the actual length in bytes transferred over USB.

Table 7-1. NU_USBH_COM_XBLOCK (cont.)

Data Type	Element Name	Description
UINT32	direction	Direction of data transfer NU_USBH_COM_DATA_OUT for transferring data from host to communication device NU_USBH_COM_DATA_IN for receiving data from communication device to host.

Description

This function takes transfer information in the form of a block.

While transmitting the data, no operation should be performed on the provided buffer until this routine returns. Similarly, while receiving, valid data is present in the provided buffer only after the successful return of this routine.

Example

```
UINT8          comm_data[0x8000];
NU_USBH_COM_XBLOCK cb_xfer;
/* Calling API function to send communication data assuming that user
   and communication driver have been already created and an
   attached communication device has been reported to user
   driver */

cb_xfer.p_data_buf = comm_data;
cb_xfer.data_length = 0x8000;
cb_xfer.direction = NU_USBH_COM_DATA_OUT;
NU_USB_COM_Data_Transfer (&global_cb_comm_device, &cb_xfer);
```

Related Topics

[Communications Class Function Reference](#)

NU_USBH_COM_Get_Comm_Feature

The user driver calls this function gets specific feature data from the attached communication device.

Usage

```
STATUS NU_USBH_COM_Get_Comm_Feature (NU_USBH_COM_DEVICE *pcb_curr_dev,
                                     VOID *p_data_buf,
                                     UINT32 data_length,
                                     UINT16 feature_sel)
```

Arguments

- **pcb_curr_dev**
Pointer to the control block of the communication device.
- **p_data_buf**
Pointer to the hold state data.
- **data_length**
Byte length for above buffer.
- **feature_sel**
Feature selector.

Return Values

- **NU_SUCCESS**
Indicates successful completion.
- **NU_USB_INVLD_ARG**
Indicates a control block is deleted before completion.
- **NU_USB_COM_XFER_ERR**
Indicates data transfer does not complete successfully due to internal error.
- **NU_USB_COM_XFER_FAILED**
Indicates command failed by the communication.

Example

```
UINT8 FEATURE_data[0x100];
/* Calling API function to get feature data assuming that user and
   communication driver have been already created and an attached
   communication device has been reported to user driver */

NU_USB_COM_Get_Comm_Feature (&global_cb_dev, FEATURE_data, 0x100,
                             0x1234);
```

Related Topics

[Communications Class Function Reference](#)

NU_USBH_COM_Get_Encap_Resp

The user driver calls this function to get the response of a previously issued command to the attached communication device. The purpose of this request varies from one device model to another.

Usage

```
STATUS NU_USBH_COM_Get_Encap_Resp (NU_USBH_COM_DEVICE *pcb_curr_dev,  
                                   VOID                *p_data_buf,  
                                   UINT32              data_length)
```

Arguments

- `pcb_curr_dev`
Pointer to the control block of the communication device.
- `p_data_buf`
Pointer to hold protocol based data.
- `data_length`
Byte length for above buffer.

Return Values

- `NU_SUCCESS`
Indicates successful completion.
- `NU_USB_INVLD_ARG`
Indicates some control block(s) is (are) deleted before completion.
- `NU_USB_COM_XFER_ERR`
Indicates data transfer does not complete successfully due to internal error.
- `NU_USB_COM_XFER_FAILED`
Indicates command failed by the communication device.

Example

```
UINT8 CMD_data[0x100];  
/* Calling API function to get response data assuming that user and  
   communication driver have been already created and an attached  
   communication device has been reported to user driver */  
  
NU_USB_COM_Get_Encap_Resp (&global_cb_dev, CMD_data, 0x100);
```

Related Topics

[Communications Class Function Reference](#)

NU_USBH_COM_Send_Encap_Cmd

The user driver calls this function to send a command to the attached communication device. The purpose of this request varies from one device model to another.

Usage

```
STATUS NU_USBH_COM_Send_Encap_Cmd (NU_USBH_COM_DEVICE *pcb_curr_dev,
                                   VOID *p_data_buf,
                                   UINT32 data_length)
```

Arguments

- **pcb_curr_dev**
Pointer to the control block of the communication device.
- **p_data_buf**
Pointer to the protocol based command to be sent.
- **data_length**
Byte length for above buffer.

Return Values

- **NU_SUCCESS**
Indicates successful completion.
- **NU_USB_INVLD_ARG**
Indicates a control block is deleted before completion.
- **NU_USB_COM_XFER_ERR**
Indicates data transfer does not complete successfully due to internal error.
- **NU_USB_COM_XFER_FAILED**
Indicates command failed by the communication.

Example

```
UINT8 CMD_data[0x100];

/* Calling API function to send cmd. data assuming that user
   and communication driver have been already created and an
   attached communication device has been reported to user driver */

NU_USB_COM_Send_Encap_Cmd (&global_cb_dev, CMD_data, 0x100);
```

Related Topics

[Communications Class Function Reference](#)

NU_USBH_COM_Set_Comm_Feature

The user driver calls this function to set a specific feature on the attached communication device. Feature data is provided through a buffer while the feature is specified through a 16-bit value.

Usage

```
STATUS NU_USBH_COM_Set_Comm_Feature (NU_USBH_COM_DEVICE *pcb_curr_dev,  
                                     VOID *p_data_buf,  
                                     UINT32 data_length,  
                                     UINT16 feature_sel)
```

Arguments

- `pcb_curr_dev`
Pointer to the control block of the communication device.
- `p_data_buf`
Pointer to the state data to be sent.
- `data_length`
Byte length for above buffer.
- `feature_sel`
Feature Selector.

Return Values

- `NU_SUCCESS`
Indicates successful completion.
- `NU_USB_INVLD_ARG`
Indicates a control block is deleted before completion.
- `NU_USB_COM_XFER_ERR`
Indicates data transfer does not complete successfully due to internal error.
- `NU_USB_COM_XFER_FAILED`
Indicates command failed by the communication.

Example

```
UINT8 FEATURE_data[0x100];  
/* Calling API function to send cmd data assuming that user  
   and communication driver have been already created and an  
   attached communication device has been reported to user driver */  
  
NU_USB_COM_Set_Comm_Feature (&global_cb_dev, FEATURE_data, 0x100,  
                             0x1234);
```


Related Topics

[Communications Class Function Reference](#)

_NU_USBH_COM_Delete

This function deletes the previously created communications class driver. No driver or attached communication device will be available at the completion of this call.

Usage

```
STATUS _NU_USBH_COM_Delete (VOID *cb)
```

Arguments

- **cb**
Pointer to the communications class driver control block.

Return Values

- **NU_SUCCESS**
Indicates successful completion.

Description

This function is indirectly called as the application always calls [NU_USB_Delete](#), then the actual routine call is made through the USB dispatch table. This entry must be made in the initialization of the class driver's dispatch table.

Example

```
STATUS status;  
/* Calling API function to delete communication class host driver with its  
   globally declared control block */  
  
status = NU_USB_Delete(&global_cb_com_drvr);  
  
/* This function is called through the dispatch table */
```

Related Topics

[Communications Class Function Reference](#)

ACM Function Reference

This section provides a description of the API exported by NU_USBH_COM's abstract control component.

- [NU_USBH_COM_Get_Line_Coding](#)
- [NU_USBH_COM_Send_Break](#)
- [NU_USBH_COM_Set_Ctrl_LS](#)
- [NU_USBH_COM_Set_Line_Coding](#)

NU_USBH_COM_Get_Line_Coding

The user driver calls this function to get the line characteristics of the communication device. Data received over USB is in little endian format and forms a line coding structure defined by USB Communications Class specifications.

Usage

```
STATUS NU_USBH_COM_Get_Line_Coding (NU_USBH_COM_DEVICE *pcb_curr_dev,  
                                   VOID *p_data_buf,  
                                   UINT32 data_length)
```

Arguments

- **pcb_curr_dev**
Pointer to the control block of the communication device.
- **p_data_buf**
Pointer to the hold line characteristics structure. Received data over USB is always in little endian format. If the data forms any structure it is the user's responsibility to convert the little endian data to match to the target endianness for structure elements.
- **data_length**
Size of the line characteristics structure in bytes.

Return Values

- **NU_SUCCESS**
Indicates successful completion.
- **NU_USB_INVLD_ARG**
Indicates a control block is deleted before completion.
- **NU_USB_COM_XFER_ERR**
Indicates data transfer does not complete successfully due to internal error.
- **NU_USB_COM_XFER_FAILED**
Indicates command failed by the communication.

Example

```
UINT8 line_data[0x100];  
  
/* Calling API function to get line data assuming that user and  
   communication driver have been already created and an attached  
   communication device has been reported to user driver */  
  
NU_USB_COM_Get_Line_Coding (&global_cb_dev, line_data, 0x100);
```

Related Topics

[ACM Function Reference](#)

NU_USBH_COM_Send_Break

The user driver calls this function to break time on the communication device.

Usage

```
STATUS NU_USBH_COM_Send_Break (NU_USBH_COM_DEVICE *pcb_curr_dev,  
                               UINT16 break_time)
```

Arguments

- `pcb_curr_dev`
Pointer to the control block of the communication device.
- `break_time`
Break duration in milliseconds.

Return Values

- `NU_SUCCESS`
Indicates successful completion.
- `NU_USB_INVLD_ARG`
Indicates a control block is deleted before completion.
- `NU_USB_COM_XFER_ERR`
Indicates data transfer does not complete successfully due to internal error.
- `NU_USB_COM_XFER_FAILED`
Indicates command failed by the communication.

Example

```
/* Calling API function to send cmd data assuming that user and  
communication driver have been already created and an attached  
communication device has been reported to user driver */  
  
NU_USB_COM_Send_Break (&global_cb_dev, 0x100);
```

Related Topics

[ACM Function Reference](#)

NU_USBH_COM_Set_Ctrl_LS

The user driver calls this function to the control line status of the communication device. Line status bitmap is defined by USB Communications Class specifications.

Usage

```
STATUS NU_USBH_COM_Set_Ctrl_LS (NU_USBH_COM_DEVICE* pcb_curr_dev,
                               UINT16 LS_bmp)
```

Arguments

- `pcb_curr_dev`
Pointer to the control block of the communication device.
- `LS_bmp`
Line Status bitmap.

Return Values

- `NU_SUCCESS`
Indicates successful completion.
- `NU_USB_INVLD_ARG`
Indicates a control block is deleted before completion.
- `NU_USB_COM_XFER_ERR`
Indicates data transfer does not complete successfully due to internal error.
- `NU_USB_COM_XFER_FAILED`
Indicates command failed by the communication.

Example

```
/* Calling API function to send cmd data assuming that user and
communication driver have been already created and an attached
communication device has been reported to user driver */

NU_USB_COM_Set_Ctrl_LS (&global_cb_dev, 0x100);
```

Related Topics

[ACM Function Reference](#)

NU_USBH_COM_Set_Line_Coding

The user driver calls this function to set the line characteristics of the communication device. Line characteristics are provided in line coding structure defined by the communication class specifications through a VOID pointer.

Usage

```
STATUS NU_USBH_COM_Set_Line_Coding (NU_USBH_COM_DEVICE *pcb_curr_dev,  
                                    VOID *p_data_buf,  
                                    UINT32 data_length)
```

Arguments

- `pcb_curr_dev`
Pointer to control block of communication device.
- `p_data_buf`
Pointer to line characteristics structure. Data is always transmitted in little endian format over USB. Nucleus USB Host Communications Class Driver handles this endianness issue while transmitting structure data to a communication device.
- `data_length`
Size of line characteristics structure in bytes.

Return Values

- `NU_SUCCESS`
Indicates successful completion.
- `NU_USB_INVLD_ARG`
Indicates a control block is deleted before completion.
- `NU_USB_COM_XFER_ERR`
Indicates data transfer does not complete successfully due to internal error.
- `NU_USB_COM_XFER_FAILED`
Indicates command failed by the communication.

Example

```
UINT8 Line_data[0x100];  
/* Calling API function to send cmd data assuming that user and  
   communication driver have been already created and an  
   attached communication device has been reported to user driver */  
  
NU_USB_COM_Set_Line_Coding (&global_cb_dev, line_data, 0x100);
```

Related Topics

[ACM Function Reference](#)

ECM Function Reference

This section provides description about the API exported by NU_USBH_COM's Ethernet control component.

- [NU_USBH_COM_Get_ETH_Power_Filter](#)
- [NU_USBH_COM_Get_ETH_Static](#)
- [NU_USBH_COM_Set_ETH_Mul_Filter](#)
- [NU_USBH_COM_Set_ETH_Packet_Filter](#)
- [NU_USBH_COM_Set_ETH_Power_Filter](#)

NU_USBH_COM_Get_ETH_Power_Filter

The user driver calls this function to get power management pattern filters of Ethernet networking device.

Usage

```
STATUS NU_USBH_COM_Get_ETH_Power_Filter (
                                         NU_USBH_COM_DEVICE *pcb_curr_dev,
                                         VOID                *p_data_buf,
                                         UINT32              data_length,
                                         UINT16              filter_num)
```

Arguments

- `pcb_curr_dev`
Pointer to control block of communication device.
- `p_data_buf`
Pointer to hold power filter data.
- `data_length`
Byte length for above buffer, usually 2 bytes.
- `filter_num`
Filter Number to set.

Return Values

- `NU_SUCCESS`
Indicates successful completion.
- `NU_USB_INVLD_ARG`
Indicates a control block is deleted before completion.
- `NU_USB_COM_XFER_ERR`
Indicates data transfer does not successfully complete due to internal error.
- `NU_USB_COM_XFER_FAILED`
Indicates command failed by the communication.

Example

```
UINT8 Filter_data[0x100];
/* Calling API function to get filter data assuming that user and
   communication driver have been already created and an attached comm.
   device has been reported to user driver */

NU_USB_COM_Get_ETH_Power_Filter (&global_cb_dev, Filter_data,
                                0x100, 0x01);
```

Related Topics

[ECM Function Reference](#)

NU_USBH_COM_Get_ETH_Static

The user driver calls this function to get static address of Ethernet networking device.

Usage

```
STATUS NU_USBH_COM_Get_ETH_Static (NU_USBH_COM_DEVICE *pcb_curr_dev,  
                                  UINT16               feature_selector,  
                                  UINT32               *feature)
```

Arguments

- `pcb_curr_dev`
Pointer to control block of communication device.
- `feature_selector`
Feature to be read.
- `feature`
Pointer to hold 4 bytes for feature.

Return Values

- `NU_SUCCESS`
Indicates successful completion.
- `NU_USB_INVLD_ARG`
Indicates a control block is deleted before completion.
- `NU_USB_COM_XFER_ERR`
Indicates data transfer does not successfully complete due to internal error.
- `NU_USB_COM_XFER_FAILED`
Indicates command failed by the communication.

Example

```
UINT32 feature_hold;  
/* Calling API function to send cmd data assuming that user and  
   communication driver have been already created and an attached comm.  
   device has been reported to user driver */  
  
NU_USB_COM_Set_ETH_Static (&global_cb_dev, 0x100, &feature_hold);
```

Related Topics

[ECM Function Reference](#)

NU_USBH_COM_Set_ETH_Mul_Filter

The user driver calls this function to set multicast filters of an Ethernet networking device. The buffer here should contain a list of the filters in network byte order.

Usage

```
STATUS NU_USBH_COM_Set_ETH_Mul_Filter (NU_USBH_COM_DEVICE *pcb_curr_dev,
                                       VOID                *p_data_buf,
                                       UINT32               data_length)
```

Arguments

- **pcb_curr_dev**
Pointer to control block of communication device.
- **p_data_buf**
Pointer to multicast filter data to be sent.
- **data_length**
Byte length for above buffer.

Return Values

- **NU_SUCCESS**
Indicates successful completion.
- **NU_USB_INVLD_ARG**
Indicates a control block is deleted before completion.
- **NU_USB_COM_XFER_ERR**
Indicates data transfer does not successfully complete due to internal error.
- **NU_USB_COM_XFER_FAILED**
Indicates command failed by the communication.

Example

```
UINT8 Filter_data[0x100];
/* Calling API function to send cmd data assuming that user and
   communication driver have been already created and an attached comm.
   device has been reported to user driver */

NU_USB_COM_Set_ETH_Mul_Filter(&global_cb_dev, Filter_data, 0x100);
```

Related Topics

[ECM Function Reference](#)

NU_USBH_COM_Set_ETH_Packet_Filter

The user driver calls this function to set packet filters of Ethernet networking device.

Usage

```
STATUS NU_USBH_COM_Set_ETH_Packet_Filter (
                                         NU_USBH_COM_DEVICE *pcb_curr_dev,
                                         UINT16              filter_bmp)
```

Arguments

- **pcb_curr_dev**
Pointer to control block of communication device
- **filter_bmp**
Bit map representing all filters to set. This bitmap is defined in USB Communications Class specifications.

Return Values

- **NU_SUCCESS**
Indicates successful completion.
- **NU_USB_INVLD_ARG**
Indicates a control block is deleted before completion.
- **NU_USB_COM_XFER_ERR**
Indicates data transfer does not successfully complete due to internal error.
- **NU_USB_COM_XFER_FAILED**
Indicates command failed by the communication.

Example

```
/* Calling API function to send cmd data assuming that user and
   communication driver have been already created and an attached comm.
   device has been reported to user driver */

NU_USB_COM_Set_ETH_Packet_Filter (&global_cb_dev, 0x100);
```

Related Topics

[ECM Function Reference](#)

NU_USBH_COM_Set_ETH_Power_Filter

The user driver calls this function to set Ethernet power management pattern filters of Ethernet networking device. The buffer here should contain the structure for power management defined by USB Communications Class specifications.

Usage

```
STATUS NU_USBH_COM_Set_ETH_Power_Filter (
                                NU_USBH_COM_DEVICE *pcb_curr_dev,
                                VOID                *p_data_buf,
                                UINT32              data_length,
                                UINT16              filter_num)
```

Arguments

- **pcb_curr_dev**
Pointer to control block of communication device.
- **p_data_buf**
Pointer to power filter data to be sent.
- **data_length**
Byte length for above buffer.
- **filter_num**
Filter Number to set.

Return Values

- **NU_SUCCESS**
Indicates successful completion.
- **NU_USB_INVLD_ARG**
Indicates a control block is deleted before completion.
- **NU_USB_COM_XFER_ERR**
Indicates data transfer does not successfully complete due to internal error.
- **NU_USB_COM_XFER_FAILED**
Indicates command failed by the communication.

Example

```
UINT8 Filter_data[0x100];

/* Calling API function to send cmd data assuming that user and
   communication driver have been already created and an
   attached comm. device has been reported to user driver */

NU_USB_COM_Set_ETH_Power_Filter (&global_cb_dev, Filter_data,
                                0x100, 0x01);
```

Related Topics

[ECM Function Reference](#)

Host Communications User Driver

User driver selection is based on the communication device model so there should be as many user drivers as there are device models for communication class. There may be further classification within a user driver on the basis of the protocol implemented. For example, an ACM user driver may have support for a virtual serial or RNDIS device.

Writing a User Driver for NU_USBH_COM

To develop a user driver, refer to “[Developing a Communication User Driver](#)” on page 345. The driver may also require following the application interface. The transfer API provided by NU_USBH_COM should be used for transferring all communication data to device and other control operations.

The following is an example of a user driver, which gets registered with NU_USBH_COM. The following is an example of the initialization for a communication user driver.

```
STATUS _NU_USBH_COM_USER_Create(NU_USBH_COM_USER *pcb_user_drvr,
                                CHAR                *p_name,
                                NU_MEMORY_POOL        *p_memory_pool,
                                NU_USBH_COM_USER_HDL  *p_handler,
                                NU_USBH_COM_USER_DISPATCH *usbh_com_user_dispatch)
{
    STATUS status;
    /* Saving user driver handlers to report device connection and
       disconnection */

    pcb_user_drvr->p_hndl_table = p_handler;

    /* Create base. */
    status = _NU_USBH_USER_Create (pcb_user_drvr, "COM-USER",
                                   p_memory_pool,
                                   0x06, /* ECM SubClass */
                                   0x00, /* Protocol is don't care */
                                   /* Globally declared communication user
                                      dispatch table */
                                   &usbh_com_user_dispatch);

    if (NU_SUCCESS != status)
        Comm_Err_Handler();
}
```

A communication user usually overrides the connect, disconnect and delete callbacks and does not need to override any of the other base functionality of NU_USBH_USER, so it fills in the dispatch table with default values. Also, functions for connection and interrupt asynchronous response are added to the base functionality. The dispatch table is given as follows.

```
const NU_USBH_COM_USER_DISPATCH usbh_com_com_eth_dispatch = {
{
    {
        NU_USBH_COM_USER_Delete,
        _NU_USB_Get_Name,
```

```
        _NU_USB_Get_Object_Id  
    },  
    NU_NULL,  
    _NU_USBH_COM_ETH_Disconnect_Handler  
},  
_NU_USBH_USER_Wait,  
NU_NULL,  
NU_NULL,  
NU_NULL  
},  
_NU_USBH_COM_ETH_Connect_Handler,  
_NU_USBH_COM_ETH_Intr_Handler  
};
```

The following is a code example for the new connection callback by our user.

```
STATUS _NU_USBH_COM_ETH_Connect_Handler(NU_USB_USER *pcb_user,  
                                         NU_USB_DRV_R *pcb_drvr,  
                                         VOID *pcb_curr_device,  
                                         VOID *information)  
{  
    STATUS status;  
    NU_USBH_COM_ETH*      pcb_eth_drvr = (NU_USBH_COM_ETH*)pcb_user;  
    NU_USBH_COM_ETH_DEVICE* pcb_eth_dev;  
    VOID* pointer;  
    VOID* temp_ptr = information;  
  
    /* Since the function is over written by Communication user driver,  
       therefore calling the base behavior for its internal  
       requirements. */  
  
    _NU_USBH_USER_Connect(pcb_user,pcb_drvr,pcb_curr_device);  
    /* Sending connection notification to application. */  
  
    status = NU_Allocate_Memory (pcb_eth_drvr->parent.memory_pool,  
                                (VOID **) &pcb_eth_dev,  
                                sizeof (NU_USBH_COM_ETH_DEVICE),  
                                NU_SUSPEND);  
  
    memset (pcb_eth_dev, 0, sizeof (NU_USBH_COM_ETH_DEVICE));  
  
    pcb_eth_dev->inform_str = *((NU_USBH_COM_ECM_INFORM*)information);  
    pcb_eth_dev->user_device.usb_device = pcb_curr_device;  
    pcb_eth_dev->user_device.user_drvr  = (NU_USBH_COM_ETH*)pcb_user;  
    pcb_eth_dev->user_device.class_drvr = (NU_USBH_COM*)pcb_drvr;  
  
    CSC_Place_On_List (  
        (CS_NODE **) & pcb_eth_drvr->pcb_first_device,  
        (CS_NODE *)  pcb_eth_dev);  
  
    /* Notify application of connection */  
    pcb_eth_drvr->p_hndl_table->Connect_Handler(pcb_user,pcb_eth_dev,  
                                                information);  
  
    return (status);  
}
```

A communication user driver has two interface layers. One is the class driver and the other is the application interface. Since the interface with the class driver should be the same for all user drivers irrespective of the protocol handled in those so Nucleus USB Host Communications Class Driver provides a base driver for the purpose. You can use this base driver as a starting point for the new user driver and add the protocol specific part straight away.

USB Host Communications User Base Driver

This user driver acts as a base component to any communication user driver. It handles almost the entire interface with the communication class driver. It models the common part for data transfer for all possible user drivers or applications based on different device models. This component is provided to speed up the application or user driver development process through presenting a basic framework.

Since Nucleus USB Host Communications Class Driver selects the user driver on the basis of device model, this component must be specialized to own a certain device model. This can be achieved through its create function.

NU_USBH_COM_USER

Like all other service components in Nucleus USB Host, the Nucleus USB Host Communication User Driver is also identified in the system by its control block. This control block is named as NU_USBH_COM_USER.

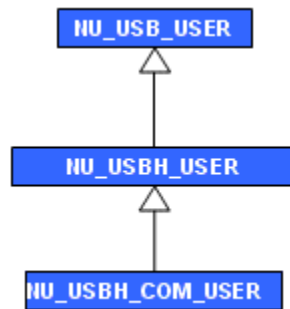
Initialization

NU_USBH_COM_USER is created and initialized in the system by the application using the API call, [_NU_USBH_COM_USER_Create](#), by passing a reference, a name, a memory pool, a subclass code for registration with class driver and few function pointers for event report. It uses this memory pool internally to store information connected to communication devices and other bookkeeping needs.

Component Hierarchy

[Figure 7-2](#) shows the component hierarchy diagram of the Nucleus USB Host Communication User Driver. As apparent from the figure, the Nucleus USB Host Communication User Driver is a specialization of the base host user driver.

Figure 7-2. Host Communications User Driver Component Hierarchy



Host Communications User Base Driver Functions

These function calls may lead to the suspension of the calling task momentarily so the application should take care in calling these routines from an ISR context.

- `_NU_USBH_COM_USER_Create`
- `_NU_USBH_COM_USER_Delete`
- `NU_USBH_COM_USER_Create_Polling`
- `NU_USBH_COM_USER_Delete_Polling`
- `NU_USBH_COM_USER_Get_Data`
- `NU_USBH_COM_USER_HDL`
- `NU_USBH_COM_USER_Register`
- `NU_USBH_COM_USER_Send_Data`
- `NU_USBH_COM_USER_Start_Polling`
- `NU_USBH_COM_USER_Stop_Polling`
- `NU_USBH_COM_USER_Wait`
- `*Connect_Handler`
- `*Data_Handler`
- `*Disconnect_Handler`
- `*Event_Handler`

_NU_USBH_COM_USER_Create

The application calls this function to create the main control block for the user driver. It allocates the memory required for its internal operations, creates required OS resources and initializes data structures.

Usage

```
STATUS _NU_USBH_COM_USER_Create (
    NU_USBH_COM_USER      *pcb_user_drvr,
    CHAR                  *p_name,
    NU_MEMORY_POOL        *p_memory_pool,
    NU_USBH_COM_USER_HDL  *p_handler,
    NU_USBH_COM_USER_DISPATCH *usbh_com_user_dispatch)
```

Arguments

- **pcb_user_drvr**
Pointer to the control block of the user driver.
- **p_name**
Pointer to the drivers name.
- **p_memory_pool**
Starting in Nucleus 3.2, this argument is no longer required and is present only for backward compatibility. NU_NULL should be passed in for p_memory_pool.
- **p_handler**
Pointer to the structure which holds pointers for application call back functions.
- **usbh_com_user_dispatch**
Pointer to the user driver dispatch table.

Return Values

- **NU_SUCCESS**
Indicates successful completion.
- **NU_USB_INVLD_ARG**
Indicates a control block is deleted before completion.

Related Topics

[Host Communications User Base Driver Functions](#)

_NU_USBH_COM_USER_Delete

The application calls this function to delete the user driver.

Usage

```
STATUS _NU_USBH_COM_USER_Delete (VOID *pcb_user_drvr)
```

Arguments

- `pcb_user_drvr`
Pointer to the control block of the user driver.

Return Values

- `NU_SUCCESS`
Indicates successful completion.
- `NU_USB_INVLD_ARG`
Indicates a control block is deleted before completion.

Description

This function is indirectly called as application always calls [NU_USB_Delete](#). Then the actual routine call is made through the USB dispatch table. Therefore this entry must be made during the initialization of the user driver's dispatch table.

Example

```
/* Already created user driver control block is cb_user_drvr */  
_NU_USB_Delete (&cb_user_drvr);  
/* Function is called through the dispatch table */
```

Related Topics

[Host Communications User Base Driver Functions](#)

NU_USBH_COM_USER_Create_Polling

The application calls this function to receive any data from the communication device or start automatic receiving. This call creates only the polling task, you must call start polling function to receive data.

Usage

```
STATUS NU_USBH_COM_USER_Create_Polling (
                                NU_USBH_COM_USER      *pcb_user_drvr,
                                NU_USBH_COM_USR_DEVICE *pcb_user_device)
```

Arguments

- `pcb_user_drvr`
Pointer to the control block of the user driver.
- `pcb_user_device`
Pointer to the control block of the user device.

Return Values

- `NU_SUCCESS`
Indicates successful completion.
- `NU_USB_INVLD_ARG`
Indicates a control block is deleted before completion.

Example

```
STATUS          status;
NU_USBH_COM_USER*  pcb_user_drvr;
NU_USBH_COM_USR_DEVICE* pcb_user_dev;

/* pcb_user_drvr and pcb_user_device are initialized prior to the call. */
status = NU_USBH_COM_USER_Create_Polling(pcb_user_drvr, pcb_user_dev);
```

Related Topics

[Host Communications User Base Driver Functions](#)

NU_USBH_COM_USER_Delete_Polling

The application calls this function to stop receiving data from the communication device.

Usage

```
STATUS NU_USBH_COM_USER_Delete_Polling (  
    NU_USBH_COM_USR_DEVICE *pcb_user_device)
```

Arguments

- `pcb_user_device`
Pointer to the control block of the user device.

Return Values

- `NU_SUCCESS`
Indicates successful completion.
- `NU_USB_INVLD_ARG`
Indicates a control block is deleted before completion.

Example

```
STATUS          status;  
NU_USBH_COM_USR_DEVICE* pcb_user_dev;  
  
/* pcb_user_device is initialized prior to the call. */  
status = NU_USBH_COM_USER_Delete_Polling(pcb_user_dev);
```

Related Topics

[Host Communications User Base Driver Functions](#)

NU_USBH_COM_USER_Get_Data

This function gets packet data from the communication device.

Usage

```
STATUS NU_USBH_COM_USER_Get_Data (
    NU_USBH_COM_USR_DEVICE *pcb_curr_device,
    UINT8 *p_file_data,
    UINT32 file_size,
    UINT32 actual_length)
```

Arguments

- **pcb_curr_device**
Pointer to the communication user device.
- **p_file_data**
Pointer to the data buffer to get.
- **file_size**
Data buffer size.
- **actual_length**
On USB it is often the case that you get lesser data then requested. This variable holds the actual received byte length which could be equal or less than the requested length.

Return Values

- **NU_SUCCESS**
Indicates successful completion.
- **NU_USB_INVLD_ARG**
Indicates a control block is deleted before completion.
- **NU_USB_COM_XFER_ERR**
Indicates data transfer does not complete successfully due to internal error.
- **NU_USB_COM_XFER_FAILED**
Indicates command failed by the communication.

Example

```
UINT8 com_data[100];
UINT32 actual_length = 0x00;

/* Already reported pointer of curr_device */
NU_USB_COM_User_Send_Data (curr_device, com_data, 100, &actual_length);
```

Related Topics

[Host Communications User Base Driver Functions](#)

NU_USBH_COM_USER_HDL

This is a structure containing different function pointers of application required and called by the communication user driver to report certain events. An application must implement all of these functions if there is any chance of calling any from the user driver.

Table 7-2. NU_USBH_COM_USER_HDL

Return Type	Name	Description
VOID	*Connect_Handler	Called by the user driver to report the connection of a specific device.
VOID	*Disconnect_Handler	Called by the user driver to report the disconnection of a specific device.
VOID	*Data_Handler	User driver has an automated incoming data poll mechanism. This function is called when such data is present in the received buffer.
VOID	*Event_Handler	Called by the user driver to report the application with some interrupt data received from device.

NU_USBH_COM_USER_Register

This function registers the user driver with the class one.

Usage

```
STATUS NU_USBH_COM_USER_Register (NU_USBH_COM_USER *pcb_user_drvr,  
                                  NU_USBH_COM      *pcb_com_drvr)
```

Arguments

- `pcb_user_drvr`
Pointer to the control block of the user driver.
- `pcb_com_drvr`
Pointer to the control block communication driver.

Return Values

- `NU_SUCCESS`
Indicates successful completion.
- `NU_USB_INVLD_ARG`
Indicates a control block is deleted before completion.

Example

```
/* cb_com_drvr is pointer for already created communication driver control  
   block and cb_user_drvr is for user driver */  
  
NU_USBH_COM_USER_Register (&cb_user_drvr, &cb_com_drvr);
```

Related Topics

[Host Communications User Base Driver Functions](#)

NU_USBH_COM_USER_Send_Data

This function sends packet data on the communication device.

Usage

```
STATUS NU_USBH_COM_USER_Send_Data (
    NU_USBH_COM_USR_DEVICE *pcb_curr_device,
    UINT8                  *p_file_data,
    UINT32                  file_size)
```

Arguments

- **pcb_curr_device**
Pointer to the communication user device.
- **p_file_data**
Pointer to the data buffer to send.
- **file_size**
Data buffer size.

Return Values

- **NU_SUCCESS**
Indicates successful completion.
- **NU_USB_INVLD_ARG**
Indicates a control block is deleted before completion.
- **NU_USB_COM_XFER_ERR**
Indicates data transfer does not complete successfully due to internal error.
- **NU_USB_COM_XFER_FAILED**
Indicates command failed by the communication.

Example

```
UINT8 com_data[100];

/* Already reported pointer of curr_device */
NU_USB_COM_User_Send_Data (curr_device, com_data, 100);
```

Related Topics

[Host Communications User Base Driver Functions](#)

NU_USBH_COM_USER_Start_Polling

The application calls this function after creating the polling to start receiving the incoming data.

Usage

```
STATUS NU_USBH_COM_USER_Start_Polling (
    NU_USBH_COM_USR_DEVICE *pcb_user_device)
```

Arguments

- `pcb_user_device`
 Pointer to the control block of the user device.

Return Values

- `NU_SUCCESS`
 Indicates successful completion.
- `NU_USB_INVLD_ARG`
 Indicates a control block is deleted before completion.

Description

Before calling this routine application must set the buffer information in the xfer block of user device as shown in the following example. Received data is reported through `Data_Handler` callback routine.

Example

```
STATUS status;
UINT8 RX_Array[0x200]= {0x00};
NU_USBH_COM_USR_DEVICE* pcb_user_dev;

/* pcb_user_device is initialized prior to the call. */
pcb_user_dev->rx_block.p_data_buf = &RX_Array;
pcb_user_dev->rx_block.data_length = 0x200;

status = NU_USBH_COM_USER_Start_Polling(pcb_user_dev);
```

Related Topics

[Host Communications User Base Driver Functions](#)

NU_USBH_COM_USER_Stop_Polling

The application calls this function to momentarily stop receiving data from the device. This may be for synchronizing the send and receive sides.

Usage

```
STATUS NU_USBH_COM_USER_Stop_Polling (  
    NU_USBH_COM_USR_DEVICE *pcb_user_device)
```

Arguments

- `pcb_user_device`
Pointer to the control block of the user device.

Return Values

- `NU_SUCCESS`
Indicates successful completion.
- `NU_USB_INVLD_ARG`
Indicates a control block is deleted before completion.

Example

```
STATUS status;  
NU_USBH_COM_USR_DEVICE* pcb_user_dev;  
  
/* pcb_user_device is initialized prior to the call. */  
status = NU_USBH_COM_USER_Stop_Polling(pcb_user_dev);
```

Related Topics

[Host Communications User Base Driver Functions](#)

NU_USBH_COM_USER_Wait

This function provides services for user level threads to wait for a particular device to be connected.

Usage

```
STATUS NU_USBH_COM_USER_Wait (NU_USBH_COM_USER *pcb_user_drvr,
                              UNSIGNED          suspend,
                              VOID              **handle_out)
```

Arguments

- **pcb_user_drvr**
Pointer to the control block of the user driver.
- **suspend**
Suspension option.
- **handle_out**
Pointer to a location to hold void pointer.

Return Values

- **NU_SUCCESS**
Indicates successful completion.
- **NU_TIMEOUT**
Indicates timeout on suspension.
- **NU_NOT_PRESENT**
Indicates event flags are not present.
- **NU_USB_INTERNAL_ERROR**
Indicates an internal error in USB subsystem

Description

The thread goes into a state specified by the suspension option if the device is not yet connected. This service returns a device handle as the output when the call is successful, which can be used by applications to use other services of this user driver. When a thread first calls this service, it checks if some device belonging to the user is already connected and gives out its handle. Otherwise, it waits for the device to be connected. If this service is called again from the same thread, it checks for the next available device and waits if it is not yet connected. This can help applications in traversing the devices to find a suitable device.

Example

```
/* cb_user_drvr is an already created control block of User  
   user driver */  
  
VOID* dev_ptr;  
NU_USB_COM_User_Wait (&cb_user_drvr, NU_SUSPEND, &dev_ptr);
```

Related Topics

[Host Communications User Base Driver Functions](#)

*Connect_Handler

This function is called when a new communication device specific to the user driver is connected on USB. This function is called from a HISR so there should be no SUSPEND activity in this call implementation.

Usage

```
VOID (*Connect_Handler) (NU_USB_USER *pcb_user_drvr,  
                        VOID          *device,  
                        VOID          *information)
```

Arguments

- `pcb_user_drvr`
Pointer to the control block of the user driver.
- `device`
Pointer to the connected device. This is type cast by the specific user driver. If there is no other layer then this is of the type `NU_USBH_COM_USR_DEVICE`.
- `information`
Pointer to the connected device's information block. This is type cast by the specific user driver.

Return Values

- `VOID`

Related Topics

[Host Communications User Base Driver Functions](#)

***Data_Handler**

This routine fetches data from the specified the device.

Usage

```
VOID (*Data_Handler)(VOID *device,  
    NU_USBH_COM_XBLOCK *xblock)
```

Arguments

- device
Pointer to the connected device. This is type cast by the specific user driver. If there is no any other layer then this is of the type NU_USBH_COM_USR_DEVICE.
- xblock
Pointer to the transfer block containing the received data and buffer information.

Return Values

- VOID

NU_USBH_COM_XBLOCK

Table 7-3. NU_USBH_COM_XBLOCK

Data Type	Element Name	Description
VOID*	p_data_buf	Pointer to data buffer.
UINT32	data_length	Buffer length in bytes.
UINT32	transfer_length	Variable to hold actual amount of data transferred.
UINT32	direction	Direction of data transfer NU_USBH_COM_DATA_OUT from host to communication device. NU_USBH_COM_DATA_IN receives data from the communication device to the host.

Description

The base user driver, NU_USBH_COM_USER, has a built in mechanism for polling the attached function for any incoming data for the Host.

Whenever data is received from a communications device, the callback of Data_Handler is called with device and buffer information. On return from this routine, new incoming data is fetched from the device. The application should either free the buffer or update its information before returning from this routine.

- This function can be created by calling the [NU_USBH_COM_USER_Create_Polling](#) API call.

- This function can be turned on temporarily by calling the [NU_USBH_COM_USER_Start_Polling](#) API call.
- This function can be turned off temporarily by calling the [NU_USBH_COM_USER_Stop_Polling](#) API call.
- This function can be deleted by calling the [NU_USBH_COM_USER_Delete_Polling](#) API call.

Related Topics

[Host Communications User Base Driver Functions](#)

***Disconnect_Handler**

This function is called when a connected communication device owned by the user driver is disconnected over USB. This function is called from a HISR so there should be no SUSPEND activity in this call implementation.

Usage

```
VOID (*Disconnect_Handler) (NU_USB_USER *pcb_user_drvr,  
                             VOID *device)
```

Arguments

- `pcb_user_drvr`
Pointer to the control block of the user driver.
- `device`
Pointer to the connected device. This is type cast by the specific user driver. If there is no other layer then this is of the type `NU_USBH_COM_USR_DEVICE`.

Return Values

- `VOID`

Related Topics

[Host Communications User Base Driver Functions](#)

*Event_Handler

This function is called from a HISR so there should be no SUSPEND activity in this call implementation.

Usage

```
VOID (*Event_Handler)(VOID *device,  
    NU_USBH_COM_XBLOCK *xblock)
```

Arguments

- device
Pointer to the connected device. This is type cast by the specific user driver. If there is not any other layer then this is of the type NU_USBH_COM_USR_DEVICE.
- xblock
Pointer to the transfer block containing the event data and buffer information.

Return Values

- VOID

Example

```
STATUS          status;  
NU_USBH_COM_USER cb_user_drvr;  
  
const NU_USBH_COM_USER_HDL func_table = { Connect_Device_User_Dummy,  
                                           Disconnect_Device_User_Dummy,  
                                           Rcvd_Packet_Handler,  
                                           Event_Handler };  
  
const NU_USBH_COM_USER_DISPATCH usbh_com_user_dispatch = {  
    {  
        {  
            _NU_USBH_COM_USER_Delete,  
            _NU_USB_Get_Name,  
            _NU_USB_Get_Object_Id  
        },  
        NU_NULL,  
        _NU_USBH_COM_ETH_Disconnect_Handler  
    },  
    _NU_USBH_USER_Wait,  
    _NU_NULL,  
    _NU_NULL,  
    _NU_NULL  
},  
    _NU_USBH_COM_ETH_Connect_Handler,  
    _NU_USBH_COM_ETH_Intr_Handler  
};
```

```
/* System_Memory is already created through NU_Create_Memory_Pool */
status = _NU_USBH_COM_USER_Create(&cb_user_driver, "UserDRV",
                                   &System_Memory, &func_table,
                                   &usbh_com_user_dispatch
);
```

Related Topics

[Host Communications User Base Driver Functions](#)

USB Function Component

Due to a variety of communication devices, USB communication class specification does not define the behavior of USB devices completely. It just gives a framework on which any communication device can be designed. According to USB specification, communication devices need at least two types of interfaces.

- **Communication class interface:** This interface works as master interface for one or more slave interfaces. This consists of a control endpoint 0, which works as the management element. This interface optionally may have a bulk or interrupt endpoint, which works as the notification element.
- **Data class interface:** The communication class interface is not used for any data transfer. For data transfer, communication devices may use other classes of interfaces like HID, Audio, and so on. If the data transfer does not match with any other class then one or more data class interfaces are used. Data class interface consists of a pair of bulk or isochronous endpoints.

Nucleus USB Function Communications Class Driver enables communication devices with one communication class interface with management and notification element and one data class interface with a pair of bulk endpoints (Bulk IN and Bulk OUT).

For different type of communication device categories, specification defines numerous control models like USB POTS Models (Direct Line Control model and Abstract control model), USB ISDN Model (Multi-channel Model, USB CAPI model) and USB Networking Models (Ethernet Networking Control Model and ATM Networking Control Model). Each of these control models is assigned different subclass codes in the interface descriptor of communication interface class. No subclasses are defined for the data interface class. Protocols for both the communication interface class and data interface class are non-USB so no protocol implementations are provided in the class drivers. These will be handled at the user driver level.

According to the communication class specification, the communication class interface class code and data class interface class codes are defined in interface descriptors. Consequently, separate interface level drivers for both interfaces are necessary.

- **NU_USBF_COMM:** This component is class driver for Communication Interface class and it acts as master interface. Any user driver is registered with this driver only. All the standard and class specific requests should be directed to this driver.
- **NU_USBF_COMM_DATA:** This component is the class driver for the Data Interface class. No user driver is registered with it. It uses the user of master interface as its own.
- Each of these drivers owns an interface on the communication device. The terms subclass and protocol codes, used above, are defined by the `bInterface Subclass` and `bInterfaceProtocol` fields of the device interface descriptor.

Function Component Control block

Like all other service components in Nucleus USB, Nucleus USB Function Communication Class Driver is also identified in the system by its control block. This control block is termed as NU_USBF_COMM.

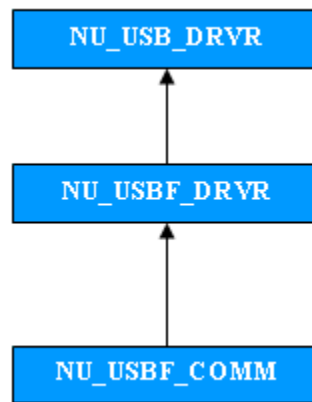
Function Component Initialization

NU_USBF_COMM is initialized in the system using the API call [NU_USBF_COMM_Create](#) by passing a reference and a name to the communication interface class control block.

Function Component Hierarchy

[Figure 7-3](#) shows the class hierarchy diagram of the Nucleus USB Function Communication Class Driver. As shown in [Figure 7-3](#), the Nucleus USB Function Communication Class Driver is a specialization of the base device class driver.

Figure 7-3. Comm Class Driver Function Component Hierarchy



Due to this inheritance philosophy, the Nucleus USB Function Stack invokes base class driver API services on references to the communication interface [extended] class driver control blocks. The stack obtains the communication interface [extended] class driver control blocks through registration; the stack uses them as normal references to [base] class driver control blocks.

Function Component Match Spec

The communication class interface is an interface level driver. As stated earlier, this class handles the management and notification for all subclasses and protocols that is handled at the user driver level. Therefore, it only provides bInterfaceClass in match spec. The class code for the communication interface as defined by the USB Forum is 2.

The following is a code example from the initialization routine of the communication class interface. The communication class interface makes calls to the base class driver created, providing the match spec and including the dispatch tables as parameters to the function.

```
/* Invoke the parents create function */
status = _NU_USBFR_DRV_Create (&cb->parent, name,
                               (USB_MATCH_CLASS), 0, 0, 0, 0,
                               (COMMF_CLASS), 0, 0,
                               &usbf_comm_dispatch);
```

Class Driver Configuration Parameters

The following are the configuration macros for the communication class interface. These macros can be found in the header file *nu_usbfr_comm_imp.h*. The default values set for these macros are suitable for most of the applications. However, they can be changed to suit the application needs as required.

Table 7-4. Class Driver Configuration Parameters

Macro	Suitable Range of Values	Description
COMMF_MAX_CMD_LENGTH	Depends on application	Maximum length of control OUT data. Default value is 256.
COMMF_MAX_NOTIF_DATA_SIZE	Depends on application	Maximum size of notification data in addition to 8 byte header. Default value is 8.
COMMF_MAX_DEVICES	Depends on application	Depends on Function Controller and application Maximum number of communication functions intended by application. Default value is four. Each function requires Bulk IN, Bulk OUT and Interrupt IN endpoint in function controller.
COMMF_HISR_MAX_STACK_SIZE	Depends on application	Depends on application/number of devices Stack size of class driver HISR. All devices supported by this driver share this HISR. Default value is 1024.

NU_USBF_COMM_DATA

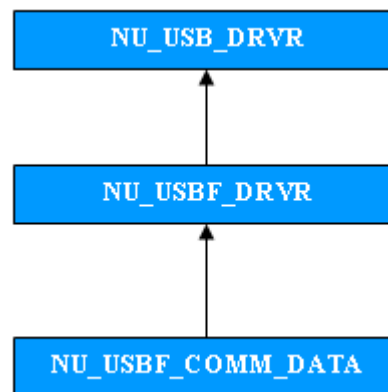
Like all other service components in Nucleus USB, the Nucleus USB Function Data Interface Class Driver is also identified in the system by its control block. This control block is termed as NU_USBF_COMM_DATA.

NU_USBF_COMM_DATA is initialized in the system using the API call [NU_USBF_COMM_DATA_Create](#) by passing a reference and a name to the Data Interface Class driver control block.

Comm Class Component Hierarchy

[Figure 7-4](#) shows the class hierarchy diagram of the Nucleus USB Function Data Interface Class Driver. As apparent from the figure, the Nucleus USB Function Data Interface Class Driver is a specialization of base device class driver. It implements common class driver functionality in a way unique to the USB data interface class.

Figure 7-4. Function Communications Data Interface Driver Component Hierarchy



Due to this inheritance philosophy, the Nucleus USB Function Stack invokes base class driver API services on references to the data interface [extended] class driver control blocks. The stack obtains the data interface [extended] class driver control blocks through registration; the stack uses them as normal references to [base] class driver control blocks.

Comm Class Match Spec

The data interface class is an interface level driver. As stated earlier, this class handles the data transfer for all subclasses and protocols that are handled at the user driver level. Therefore, it only provides bInterfaceClass in match spec. The class code for the data interface as defined by the USB Forum is 0x0A.

The following is a code example from the initialization routine of the data interface class driver. The data interface class driver makes calls to the base class driver created, providing the match spec and including the dispatch tables as parameters to the function.

```
/* Invoke the parents create function */
status = _NU_USBF_DRV_Create (&cb->parent, name,
                             (USB_MATCH_CLASS), 0, 0, 0, 0,
                             (COMMF_DATA_CLASS), 0, 0,
                             &usbf_data_dispatch);
```

Nucleus USB Function Communications User Base Driver

The Nucleus USB Communications Class Driver supports various subclasses defined by the *USB Communication Devices Specification*. For different control models given above, separate subclasses are defined. For each subclass, new user driver must be implemented. Communication devices are recognized by its communication interface (master interface), therefore all user drivers are registered with the communication interface class driver only. User drivers use the API of both the NU_USBF_COMM and NU_USBF_COMM_DATA.

NU_USBF_USER_COMM

All subclass drivers for the Communication class driver are direct descendents of the NU_USBF_USER_COMM component. The following is its dispatch table.

```
typedef struct _nu_usbf_user_comm_dispatch
{
    NU_USBF_USER_DISPATCH dispatch;
    STATUS (*DATA_Connect) (NU_USB_USER *user,
                           NU_USB_DRV *class_driver,
                           VOID *handle);

    STATUS (*DATA_Disconnect) (NU_USB_USER *user,
                              NU_USB_DRV *class_driver,
                              VOID *handle);

    STATUS (*Wait) (NU_USBF_USER_COMM *user, UNSIGNED suspend,
                  VOID **handle_out);
}
NU_USBF_USER_COMM_DISPATCH;
```

DATA_Connect is connect-callback function for the data interface class driver. Upon receiving this callback, the user should reset all its statistics and buffers, do the necessary initialization to start normal operation. Default implementation is [NU_USBF_USER_COMM_DATA_Connect](#). All user drivers should call this function in their Connect function.

DATA_Disconnect is disconnect-callback function of the data interface class driver. This is called when the configuration is changed in device. Upon receiving this callback, user should

reverse all the initialization done in the Connect callback and stop normal operation. Default implementation is `_NU_USBF_USER_COMM_Disconnect()`. All user drivers should call this default in their `Data_Disconnect` callback.

Wait is a function used by application to wait for proper initialization of both class drivers. This function waits for the connect callback of both communication interface and data interface class driver. Default implementation is [NU_USBF_USER_COMM_Wait](#). If some user driver wants to give their own implementation, it should call the parent's function in their function.

For further instructions on developing user drivers for the Nucleus USB Function software, refer to [“Developing a Communication User Driver”](#) on page 345.

Communications Interface Function Reference

The following function reference contains all Nucleus USB Function Communication Class Driver services.

- [NU_USBF_COMM_Cancel_Io](#)
- [NU_USBF_COMM_Create](#)
- [NU_USBF_COMM_Send_Notification](#)

NU_USBFS_COMM_Cancel_Io

This function cancels ongoing IO on the communication interface.

Usage

```
STATUS NU_USBFS_COMM_Cancel_Io (VOID *cb,
                                USBFS_COMM_USER_NOTIFICATION *notif,
                                VOID *handle)
```

Arguments

- **cb**
 Pointer to the communication user driver control block.
- **notif**
 Pointer to notification structure. The notification structure has the following fields. User identification is provided through the handle input.

notification
 1 byte Notification code identifying individual notification.

notif_value
 2 byte value further illustrating a specific notification. This is wValue field of the standard notification header. For details, refer to the *USB Class Definition of Communication Devices*.

data
 Pointer to buffer of data associated with Notification.

length
 Length of associated data.
- **handle**
 Handle for this notification.

Return Values

- **NU_SUCCESS**
 Indicates successful submission.
- **NU_USB_INVLD_ARG**
 Some input argument(s) are invalid.

Related Topics

[Communications Interface Function Reference](#)

NU_USBFS_COMM_Create

This function initializes the USB communication interface driver component. The class driver is characterized by the class code of communication and match specs.

Usage

```
STATUS NU_USBFS_COMM_Create (NU_USBFS_COMM  *cb,  
                             CHAR             *name,  
                             NU_MEMORY_POOL  *pool)
```

Arguments

- **cb**
Pointer to the communication driver control block.
- **name**
Pointer to a seven-character name for the communication driver. The name must be null-terminated.
- **pool**
Starting in Nucleus 3.2, this argument is no longer required and present only for backward compatibility. NU_NULL should be passed in for pool.

Return Values

- **NU_SUCCESS**
Indicates successful initialization.
- **NU_USB_INVLD_ARG**
Indicates an invalid argument.

Related Topics

[Communications Interface Function Reference](#)

NU_USBF_COMM_Send_Notification

This function submits the notification request to the hardware driver.

Usage

```
STATUS NU_USBF_COMM_Send_Notification (
                                VOID                                *cb,
                                USBF_COMM_USER_NOTIFICATION *notif,
                                VOID                                *handle)
```

Arguments

- **cb**
 Pointer to the communication user driver control block.
- **notif**
 Pointer to notification structure. The notification structure has the following fields. User identification is provided through the handle input.

notification
 1 byte Notification code identifying individual notification.

notif_value
 2 byte value further illustrating a specific notification. This is wValue field of the standard notification header. For details, refer to the *USB Class Definition of Communication Devices*.

data
 Pointer to buffer of data associated with Notification.

length
 Length of associated data.
- **handle**
 Handle for this notification.

Return Values

- **NU_SUCCESS**
 Indicates successful submission. Upon successful completion of notification, the Tx_Done user callback is called by the class driver.
- **NU_USB_INVLD_ARG**
 Some input argument(s) are invalid.
- **NU_USB_NOT_SUPPORTED**
 This device does not support notification.

Related Topics

[Communications Interface Function Reference](#)

Data Interface Function Reference

The following function reference contains all Nucleus USB Function data interface class driver services.

- [NU_USBF_COMM_DATA_Cancel_Io](#)
- [NU_USBF_COMM_DATA_Config_Xfers](#)
- [NU_USBF_COMM_DATA_Create](#)
- [NU_USBF_COMM_DATA_Dis_Reception](#)
- [NU_USBF_COMM_DATA_Get_Rcvd](#)
- [NU_USBF_COMM_DATA_Rbg_Create](#)
- [NU_USBF_COMM_DATA_Reg_Rx_Buffer](#)
- [NU_USBF_COMM_DATA_Send](#)

NU_USBF_COMM_DATA_Cancel_Io

This function cancels all TX operations submitted to data interface of communication class driver. It has no effect on RX operations.

Usage

```
STATUS NU_USBF_COMM_DATA_Cancel_Io (NU_USBF_COMM_DATA *cb,  
                                     VOID *handle)
```

Arguments

- **cb**
Pointer to the USB driver control block.
- **handle**
Identification handle for this device.

Return Values

- **NU_SUCCESS**
Indicates success.
- **NU_USB_INVLD_ARG**
Indicates that in_pipe's reference is not initialized properly. Device is in disconnected state.

Related Topics

[Data Interface Function Reference](#)

NU_USBF_COMM_DATA_Config_Xfers

This function configures different data transfer parameters for the user driver specified by handle input. The host must call this function at least once after configuration.

Usage

```
STATUS NU_USBF_COMM_DATA_Config_Xfers (NU_USBF_COMM_DATA *data_drvr,
                                       VOID                *handle,
                                       COMMF_DATA_CONF      *conf)
```

Arguments

- **data_drvr**
 Pointer to the data interface driver control block.
- **handle**
 Handle for this current transfer.
- **conf**
 Pointer to transfer configuration structure. You must initialize the following fields of conf structure:

cmpltd_rx_buffer_list

This is user/application allocated array for holding the buffer pointers in queue of completed receive transfers.

cmpltd_rx_len_list

This is user/application allocated array for holding the buffer lengths in queue of completed receive transfers.

max_cmpltd_rx_buffers

Number of buffers in queue of completed receive transfers.

pend_tx_buffer_list

This is user/application allocated array for holding the buffer pointer in queue of pending transmit transfers.

pend_tx_len_list

This is user/application allocated array for holding the buffer lengths in queue of pending transmit transfers.

max_pend_tx_buffers

Number of buffers in queue of pending transmit transfers.

cmpltd_tx_buffer_list

This is user/application allocated array for holding the buffer pointer in queue of completed transmit transfers.

`cmpltd_tx_len_list`

This is user/application allocated array for holding the buffer lengths in queue of completed receive transfers.

`max_cmpltd_tx_buffers`

Number of buffers in queue of completed transmit transfers.

`delineate`

Flag shows if class driver should delineate the transmit transfers. In case of delineation, class driver will send a zero length packet with the transfer multiple of maximum packet size of Bulk IN endpoint.

Return Values

- `NU_SUCCESS`
Indicates success.
- `NU_USBH_INVLD_ARG`
Indicates an invalid argument.

Related Topics

[Data Interface Function Reference](#)

NU_USBFS_COMM_DATA_Create

This function initializes the USB data interface driver component. The class driver is characterized by the class code of the data interface and match specs.

Usage

```
STATUS NU_USBFS_COMM_DATA_Create (NU_USBFS_COMM_DATA *cb,
                                  CHAR                  *name)
```

Arguments

- **cb**
 Pointer to the Data Interface driver control block.
- **name**
 Pointer to a seven-character name for the communication driver. The name must be null-terminated.

Return Values

- **NU_SUCCESS**
 Indicates successful initialization.
- **NU_USB_INVLD_ARG**
 Indicates an invalid argument.

Related Topics

[Data Interface Function Reference](#)

NU_USBF_COMM_DATA_Dis_Reception

This function clears the list of available receive buffer groups and cancels any ongoing data reception for the user driver identified by the handle input.

Usage

```
STATUS NU_USBF_COMM_DATA_Dis_Reception (NU_USBF_COMM_DATA *data_drvr,  
                                         VOID                *handle)
```

Arguments

- **data_drvr**
Pointer to the data interface driver control block.
- **handle**
Handle for this current cancellation.

Return Values

- **NU_SUCCESS**
Indicates success.
- **NU_USB_INVLD_ARG**
Indicates an invalid argument.

Related Topics

[Data Interface Function Reference](#)

NU_USBF_COMM_DATA_Get_Rcvd

This function de-queues a received data buffer from the completed receive buffers queue. User identification is provided through the handle input. This function expects to be called by the user driver once it receives the New_Transfer callback.

Usage

```
STATUS NU_USBF_COMM_DATA_Get_Rcvd (NU_USBF_DVR *cb,
                                     UINT8      **data_out,
                                     UINT16     *data_len_out,
                                     VOID        *handle)
```

Arguments

- **cb**
Pointer to the data interface driver control block.
- **data_out**
Pointer to hold data buffer pointer.
- **data_len_out**
Pointer to hold length.
- **handle**
Handle for this current transfer.

Return Values

- **NU_SUCCESS**
Indicates success.
- **NU_USB_INVLD_ARG**
Indicates an invalid argument.

Related Topics

[Data Interface Function Reference](#)

NU_USBF_COMM_DATA_Rbg_Create

This function initializes the specified receive buffer group.

Usage

```
STATUS NU_USBF_COMM_DATA_Rbg_Create (COMMF_RX_BUF_GROUP *buff_grp,  
                                     COMMF_BUFF_FINISHED *callback,  
                                     UINT8 ** buff_array,  
                                     UINT32 num_of_buffs,  
                                     UINT32 buff_size)
```

Arguments

- **buff_grp**
Pointer to the receive buffer group.
- **callback**
Pointer to callback function which will be called once all buffers in this group are consumed.
- **buff_array**
Array containing receive buffer pointers.
- **num_of_buffs**
Number of buffers in this buffer group.
- **buff_size**
Size of buffer in this buffer group.

Return Values

- **NU_SUCCESS**
Indicates success.

Description

After initialization, this buffer group should be added to the list of available buffer groups using the [NU_USBF_COMM_DATA_Reg_Rx_Buffer](#) function call. You must specify the previously allocated array containing the receive buffer pointers, the size of each buffer, and the number of buffers in this group.

Related Topics

[Data Interface Function Reference](#)

NU_USBF_COMM_DATA_Reg_Rx_Buffer

This function adds the specified buffer group in the list of available receive buffer groups of the data interface class driver.

Usage

```
STATUS NU_USBF_COMM_DATA_Reg_Rx_Buffer (NU_USBF_COMM_DATA *data_drvr,
                                         VOID *handle,
                                         COMMF_RX_BUF_GROUP *buffer_group)
```

Arguments

- **data_drvr**
Pointer to the data interface driver control block.
- **handle**
Handle for this current buffer group.
- **buffer_group**
Pointer to receive buffer group.

Return Values

- **NU_SUCCESS**
Indicates success.
- **NU_USB_INVLD_ARG**
Indicates an invalid argument.

Description

If no receive transfer is in progress, this function will submit the request to receive data from the Host. Specified buffer group **buffer_group** should be initialized using the [NU_USBF_COMM_DATA_Rbg_Create](#) before calling this function. User driver is identified with the handle input. See the description of [NU_USBF_COMM_DATA_Rbg_Create](#) function for more details.

Related Topics

[Data Interface Function Reference](#)

NU_USBFS_COMM_DATA_Send

This function submits the data transmission request to hardware driver.

Usage

```
STATUS NU_USBFS_COMM_DATA_Send (VOID    *cb,  
                                UINT8    *buffer,  
                                UINT16   length,  
                                VOID      *handle)
```

Arguments

- **cb**
Pointer to the data interface driver control block.
- **buffer**
Pointer to transmit buffer.
- **length**
Length of transmit data.
- **handle**
Handle for this current transfer.

Return Values

- **NU_SUCCESS**
Indicates successful submission.
- **NU_USB_INVLD_ARG**
Indicates an invalid argument.

Description

Upon successful completion of transfer, the class driver calls the Tx_Done user callback. User identification is provided through the handle input. Multiple transmit requests can be given using this function call repeatedly. The class driver provides facility of queuing up the transfers.

Related Topics

[Data Interface Function Reference](#)

Nucleus USB Function Communications User Function Reference

The following function reference contains all Nucleus USB Function base communication user driver services.

- [_NU_USBF_USER_COMM_Create](#)
- [NU_USBF_USER_COMM_DATA_Connect](#)
- [NU_USBF_USER_COMM_DATA_Discon](#)
- [NU_USBF_USER_COMM_Delete](#)
- [NU_USBF_USER_COMM_Wait](#)

_NU_USB_USER_COMM_Create

This function initializes the USB communication base user component.

The user driver is characterized by the bInterfaceSubclass and bInterfaceSubclass code. This function initializes all the data to their default values.

Usage

```
STATUS_NU_USB_USER_COMM_Create (NU_USB_USER_COMM *cb,  
                                CHAR                *name,  
                                UINT8               bInterfaceSubclass,  
                                UINT8               bInterfaceProtocol,  
                                BOOLEAN              reqrd_data,  
                                const VOID          *dispatch)
```

Arguments

- **cb**
Pointer to the communication user control block.
- **name**
Pointer to a seven-character name for the communication driver. The name must be null-terminated.
- **bInterfaceSubclass**
Subclass code associated with this user driver.
- **bInterfaceProtocol**
Protocol code associated with this user driver.
- **reqrd_data**
Flag indicating data interface existence for this user driver.
- **dispatch**
Pointer to dispatch table of user driver.

Return Values

- **NU_SUCCESS**
Indicates successful initialization.

Related Topics

[Nucleus USB Function Communications](#)
[User Function Reference](#)

NU_USB_USER_COMM_DATA_Connect

This function is the Connect callback function of the data interface class driver. The user driver is expected to reset all its counters and buffers and to restart its normal operation after this callback.

Usage

```
STATUS NU_USB_USER_COMM_DATA_Connect (NU_USB_USER *cb,
                                     NU_USB_DRV *class_driver,
                                     VOID *handle)
```

Arguments

- **cb**
Pointer to the user driver control block.
- **class_driver**
Pointer to connection sending class driver
- **handle**
Pointer to handle of current driver. It normally points to communication device structure.

Return Values

- **NU_SUCCESS**
Indicates success.

Related Topics

[Nucleus USB Function Communications](#)
[User Function Reference](#)

NU_USBF_USER_COMM_DATA_Discon

This function is the Disconnect callback function of data interface class driver. The user driver is expected to reset all its counters and buffers to stop its normal operation after this callback.

Usage

```
STATUS NU_USBF_USER_COMM_DATA_Discon (NU_USB_USER *cb,  
                                       NU_USB_DRV *class_driver,  
                                       VOID *handle)
```

Arguments

- **cb**
Pointer to the user driver control block.
- **class_driver**
Pointer to connection sending class driver.
- **handle**
Pointer to handle of current driver. It normally pointer to communication device structure.

Return Values

- **NU_SUCCESS**
Indicates success.

Related Topics

[Nucleus USB Function Communications](#)
[User Function Reference](#)

NU_USBFS_USER_COMM_Delete

This function is called by specialized communication user driver in its Delete callback function.

Usage

```
STATUS_NU_USBFS_USER_COMM_Delete (VOID *cb)
```

Arguments

- `cb`
Pointer to the user driver control block.

Return Values

- `NU_SUCCESS`
Indicates successful submission.

Related Topics

[Nucleus USB Function Communications
User Function Reference](#)

NU_USBFS_USER_COMM_Wait

This function is used by the application thread to synchronize its operation with the communication device initialization.

Usage

```
STATUS NU_USBFS_USER_COMM_Wait (NU_USBFS_USER_COMM *cb,  
                                UNSIGNED             suspend,  
                                VOID                 **handle_out)
```

Arguments

- **cb**
Pointer to the user driver control block.
- **suspend**
Task suspension option.
- **handle_out**
Pointer to hold the retrieved handler.

Return Values

- **NU_SUCCESS**
Indicates success.

Description

Communication may need to send data and notifications at any time. Before starting any such operation, application will call this function. This function will suspend the calling task unless initialization is complete. This task assumes just one application thread for the communication user. If more tasks are willing to use the user driver, they should do the mutual synchronization at their own.

Related Topics

[Nucleus USB Function Communications](#)
[User Function Reference](#)

Developing a Communication User Driver

This section illustrates various steps involved in developing a new communication user driver. For this purpose, it is assumed that the user driver component being created is ‘NU_USB_F_XYZ’, which is a Nucleus USB Function Communication XYZ user driver.

Step 1: Creating Control Block

The control block for the XYZ user driver must be defined. This control block is a “C” structure that contains all the data elements and resources required for the functioning of XYZ. This structure must include an instance of the parent’s control block as its first field.

The following code snippet shows a sample control block for the NU_USB_F_XYZ.

```
/* XYZ Communication user driver control block. */
typedef struct _nu_usb_f_xyz
{
    NU_USB_F_USER_COMM parent;
    /* XYZ user driver data.*/
} NU_USB_F_XYZ;
```

Step 2: Defining a new Dispatch Table

In general, the services offered by the user drivers depend on the application requirements. The user drivers tend to be application/class driver specific. Any communication user driver being implemented is specialization of NU_USB_F_USER_COMM, so it will include its parents dispatch in its dispatch table. If new user driver specialization is expected by other user drivers, a new function can be added in its dispatch table.

The following code snippet shows a sample dispatch table for NU_USB_F_XYZ.

```
/* XYZ Communication user driver dispatch. */
typedef struct _nu_usb_f_xyz_dispatch
{
    NU_USB_F_USER_COMM_DISPATCH dispatch;
    /* New services by XYZ driver should be defined here. */
} NU_USB_F_XYZ_DISPATCH;
```

Step 3: Implementing the User Driver

Implement all the functions identified in the dispatch table and additional non-extensible services, if any of the new user driver.

Step 4: Populating the Dispatch Table

The dispatch table needs to be filled in with specialized functions implemented by the XYZ user driver. The new user driver may choose to make use of some of the default implementations provided by the parent user driver.

The following code snippet shows a sample dispatch table for the NU_USBF_XYZ.

```
const NU_USBF_XYZ_DISPATCH usbf_xyz_dispatch = {
    /* comm user dispatch */
    {
        /* usbf user dispatch */
        {
            /* usb user dispatch */
            {
                /* usb dispatch */
                {
                    _NU_USBF_XYZ_Delete,
                    _NU_USB_Get_Name,
                    _NU_USB_Get_Object_Id
                },
                _NU_USBF_XYZ_Connect,
                _NU_USBF_XYZ_Disconnect
            },
            _NU_USBF_XYZ_New_Command,
            _NU_USBF_XYZ_New_Transfer,
            _NU_USBF_XYZ_Tx_Done,
            _NU_USBF_XYZ_Notify
        },
        _NU_USBF_XYZ_DATA_Disconnect,
        _NU_USBF_XYZ_DATA_Connect,
        _NU_USBF_USER_COMM_Wait          /* Using default. */
    }
    /* Extension to Driver Services goes here */
};
```

Step 5: Create Function

Each of the user drivers must export a component create function, usually of the name `<component_name>_Create()`. This function must at least have one argument, the pointer to the component control block instance. Optionally, name can be the second argument. After its initialization, this should call the create function of parent with proper subclass code of communication class.

The following code snippet shows the create function for NU_USBF_XYZ.

```
STATUS NU_USBF_XYZ_Create (NU_USBF_ETH *cb, CHAR *name)
{
    /* Initialize the XYZ user driver data. */
    /* Call the create function of parent. */
    return (_NU_USBF_USER_COMM_Create ((NU_USBF_USER_COMM * )cb,
        name, XYZ_SUBCLASS,
        &usbf_xyz_dispatch));
}
```

```
}
```

Step 6: Activating the User Driver

Before the user driver can be used, it needs to be created and must be registered with the relevant class driver.

The following code snippet demonstrates how this can be done.

```
/* Create the communication interface class driver. */
status = NU_USBF_COMM_Create (&function_comm, "demo", &System_Memory);

if (status != NU_SUCCESS)
    my_error_handler();

/* Create Data interface class driver. */
status = NU_USBF_COMM_DATA_Create (&function_data, "demo");

if (status != NU_SUCCESS)
    my_error_handler();

/* Create the XYZ communication user driver. */
status = NU_USBF_XYZ_Create (&function_xyz, "demo");

if (status != NU_SUCCESS)
    my_error_handler();

/* Register the user with communication interface driver. */
status = NU_USB_DRV_Register_User ((NU_USB_DRV *)
                                   &function_comm, (NU_USB_USER *)
                                   &function_xyz);

if (status != NU_SUCCESS)
    my_error_handler();
```

Nucleus USB Communications Class Driver Callback Functions

Communication interface class driver and data interface class driver calls some functions of its register user through the dispatch table. Definitions of these functions are obtained from the dispatch entries of most specialized communication user driver. Usage of these callback functions depends on the communication class drivers (NU_USBF_COMM and NU_USBF_COMM_DATA). The communication user driver must implement the following callback functions.

- [_NU_USBF_XYZ_USER_Connect](#)
- [_NU_USBF_XYZ_USER_Delete](#)
- [_NU_USBF_XYZ_USER_Disconnect](#)
- [_NU_USBF_XYZ_USER_New_Command](#)
- [_NU_USBF_XYZ_USER_New_Transfer](#)
- [_NU_USBF_XYZ_USER_Notify](#)
- [_NU_USBF_XYZ_USER_Tx_Done](#)

_NU_USBIF_XYZ_USER_Connect

This function is defined in NU_USB_USER API. This function is called by the initialization routine of the communication interface class driver.

Usage

```
STATUS _NU_USBIF_XYZ_USER_Connect (NU_USB_USER *cb,
                                   NU_USB_DRV *class_driver,
                                   VOID *handle)
```

Arguments

- **cb**
Pointer to the user driver control block.
- **class_driver**
Pointer to control block of caller class driver (NU_USBIF_COMM).
- **handle**
Pointer to handle of this user driver (USBIF_COMM_DEVICE).

Return Values

- **NU_SUCCESS**
Indicates success.

Description

This function should do its necessary initialization and call the default implementation (_NU_USBIF_USER_COMM_Connect) of NU_USBIF_USER_COMM.

Related Topics

[Nucleus USB Communications Class
Driver Callback Functions](#)

_NU_USB_F_XYZ_USER_Delete

This function is defined in NU_USB_USER API. This function can be called from the application or any task that wants to delete this component.

Usage

```
STATUS _NU_USB_F_XYZ_USER_Delete (VOID *cb)
```

Arguments

- **cb**
Pointer to the user driver control block.

Return Values

- **NU_SUCCESS**
Indicates success.

Related Topics

[Nucleus USB Communications Class
Driver Callback Functions](#)

_NU_USBF_XYZ_USER_Disconnect

This function is defined in the NU_USB_USER API. This function is called by disconnection routine of the communication interface class driver.

Usage

```
STATUS _NU_USBF_XYZ_USER_Disconnect (NU_USB_USER *cb,
                                     NU_USB_DRV *class_driver,
                                     VOID          *handle)
```

Arguments

- **cb**
Pointer to the user driver control block.
- **class_driver**
Pointer to control block of caller class driver (NU_USBF_COMM).
- **handle**
Pointer to handle of this user driver (USBF_COMM_DEVICE).

Return Values

- **NU_SUCCESS**
Indicates success.

Description

This function should do its necessary steps and call the default implementation (_NU_USBF_USER_COMM_Disconnect) of NU_USBF_USER_COMM.

Related Topics

[Nucleus USB Communications Class
Driver Callback Functions](#)

_NU_USBF_XYZ_USER_New_Command

This function is defined in the NU_USBF_USER API. The communication interface class driver calls this function when the host sends class/subclass specific requests to the communication device through control transfer.

Usage

```
STATUS _NU_USBF_XYZ_USER_New_Command (NU_USBF_USER *cb,  
                                       NU_USBF_DRVVR *drvvr,  
                                       VOID *handle,  
                                       UINT8 *command,  
                                       UINT16 cmd_len,  
                                       UINT8 **data_out,  
                                       UINT32 *data_len_out)
```

Arguments

- **cb**
Pointer to the user driver control block.
- **drvvr**
Pointer to control block of caller class driver (NU_USBF_COMM).
- **handle**
Pointer to handle of this user driver (USBF_COMM_DEVICE).
- **command**
Pointer to command structure. For communication user, this command structure is USBF_COMM_USER_CMD.
- **cmd_len**
Length of command.
- **data_out**
Pointer to hold data buffer returned by this function to be transferred to the Host.
- **data_len_out**
Pointer to hold length of data to be transferred to the Host.

Return Values

- **NU_SUCCESS**
Indicates success.
- **NU_USB_NOT_SUPPORTED**
The user driver does not support the current command.

Description

The detail of the command is known from the command structure, which has the following fields:

- **command**
This is a 1 byte command code that identifies an individual command.
- **cmd_index**
This is a 2 byte field which wIndex field of setup packet. This field is specific to this particular command.
- **cmd_value**
This is a 2 byte field which wValue field of setup packet. This field is specific to this command.
- **cmd_data**
Pointer to data buffer for data associated with this command.
- **data_len**
Length of data associated with this command.

If a request has associated data from Host to Device, the class driver requests the data from the Host and calls this function. If the data direction is from Device to Host, the user driver should update the data_out and data_len_out with the data buffer and length before returning.

Related Topics

[Nucleus USB Communications Class
Driver Callback Functions](#)

_NU_USB_F_XYZ_USER_New_Transfer

This function is defined in NU_USB_F_USER API. This function is called by the data interface class driver when it successfully receives one data payload from the Host.

Usage

```
STATUS _NU_USB_F_XYZ_USER_New_Transfer (NU_USB_F_USER *cb,  
                                         NU_USB_F_DRV *drv,  
                                         VOID *handle,  
                                         UINT8 **data_out,  
                                         UINT32 *data_len_out)
```

Arguments

- **cb**
Pointer to the user driver control block.
- **drv**
Pointer to control block of caller class driver (NU_USB_F_COMM).
- **handle**
Pointer to handle of this user driver (USB_F_COMM_DEVICE).
- **data_out**
Pointer to hold data buffer returned by this function to be transferred to the Host.
- **data_len_out**
Pointer to hold length of data to be transferred to the Host.

Return Values

- **NU_SUCCESS**
Indicates success.

Description

The class driver expects the user driver to call its function [NU_USB_F_COMM_DATA_Get_Rcvd](#) to get the received data parameters. The class driver holds the received data buffer parameters in its internal queue unless above mentioned function is not called or the queue is overflowed. As no data transfer is expected in response to current transfer, so updating last two parameters has no effect.

Related Topics

[Nucleus USB Communications Class
Driver Callback Functions](#)

_NU_USB_F_XYZ_USER_Notify

This function is defined in the NU_USB_F_USER API. This function is called by the class driver's Notify function for a bus event like reset, suspend, resume.

Usage

```
STATUS _NU_USB_F_XYZ_USER_Notify (NU_USB_F_USER *cb,
                                NU_USB_F_DRV_R *drv_r,
                                VOID *handle,
                                UINT32 event)
```

Arguments

- **cb**
Pointer to the user driver control block.
- **drv_r**
Pointer to control block of caller class driver (NU_USB_F_COMM).
- **handle**
Pointer to handle of this event (communication device).
- **event**
Value of specific event.

Return Values

- **NU_SUCCESS**
Indicates success.

Related Topics

[Nucleus USB Communications Class
Driver Callback Functions](#)

_NU_USB_F_XYZ_USER_Tx_Done

This function is defined in NU_USB_F_USER API. This function is called by the data Interface class driver/communication interface class driver when any of the following are true:

- Notification is successfully sent to the Host.
- Data associated with New_Command callback is successfully sent to Host.
- Data is successfully sent to the Host through the Send API function of the data interface class driver.

Usage

```
STATUS _NU_USB_F_XYZ_USER_Tx_Done (NU_USB_F_USER *cb,  
                                   NU_USB_F_DRV *drv,  
                                   VOID *handle,  
                                   UINT8 *completed_data,  
                                   UINT32 completed_data_len,  
                                   UINT8 **data_out,  
                                   UINT32 *data_len_out)
```

Arguments

- cb
Pointer to the user driver control block.
- drv
Pointer to control block of caller class driver (NU_USB_F_COMM).
- handle
Pointer to handle of this completed transfer. Values of the handle are as follows:

COMMF_NOTIF_SENT
1
COMMF_GET_CMD_SENT
2
COMMF_DATA_SENT
3
- completed_data
Pointer to data buffer of completed data.
- completed_data_len
Completed data length.
- data_out
Pointer to hold data buffer returned by this function to be transferred to the Host.

- `data_len_out`
Pointer to hold length of data to be transferred to the Host.

Return Values

- `NU_SUCCESS`
Indicates success.

Related Topics

[Nucleus USB Communications Class
Driver Callback Functions](#)

Chapter 8

USB Mass Storage

This chapter describes the Nucleus USB Host Mass Storage Class Driver software module and the requirements for using it.

Caution



To guarantee future support and compatibility for your application, use only documented Nucleus interfaces, structures, macros, and so on.

USB Host Driver

The Nucleus USB Host Mass Storage Class Driver enables the Nucleus USB Host to access mass storage devices, which comply with the *USB Mass Storage Class Specification Overview*. It supports BOT protocol for data transfer. Instead of providing support for all SubClasses by itself monolithically, the Nucleus USB Host Mass Storage Class Driver assigns command set handling to user drivers. In this way it decouples command set interpretation, which is independent of both USB and the mass storage class, to a separate layer. This aids in developing a user driver that handles command set, independently and efficiently by using mass storage driver services. In addition, it helps in reusing existing command set drivers, if any, in the system.

According to the *USB Mass Storage Class Specification Overview*, mass storage class code is defined in the interface descriptor. Consequently, the Nucleus USB Host Mass Storage Driver is an interface driver and owns an interface on a device. The terms SubClass and Protocol codes, used above are defined in the `bInterfaceSubclass` and `bInterfaceProtocol` fields of the interface descriptor of the device.

NU_USBH_MS

Like all other service components in Nucleus USB Host, the Nucleus USB Host Mass Storage Driver is also identified in the system by its control block, `NU_USBH_MS`.

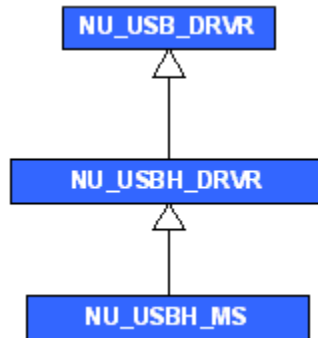
Initialization

`NU_USBH_MS` is initialized in the system using the API call, `NU_USBH_MS_Create`, by passing a reference, a name, and a memory pool to the mass storage control block. The mass storage driver uses this memory pool internally for storing information connected to mass storage devices and other bookkeeping needs.

Component Hierarchy

Figure 8-1 shows the class hierarchy diagram of the Nucleus USB Mass Storage Class Driver. As apparent from the figure, the Nucleus USB Mass Storage Class Driver is a specialization of the base host class driver. It implements common class driver functionality, unique to the USB mass storage class. Additional services required to access mass storage devices are also provided.

Figure 8-1. Host Mass Storage Class Driver Component Hierarchy



Due to this inheritance philosophy, Nucleus USB Host Stack invokes base class driver API services on references to mass storage [extended] class driver control blocks. The stack obtains mass storage [extended] class driver control blocks through registration; the stack uses them as normal references to [base] class driver control blocks.

Mass Storage Host Match Spec

The Nucleus USB Host Mass Storage Class Driver is an interface level driver. As stated earlier, this class driver handles the transport protocols, such as BOT, and user drivers are required to handle SubClasses, for example, command sets such as SCSI. Therefore, it only provides bInterfaceClass in match spec. The class code for mass storage as defined by the USB Forum is 8.

The following is a code example from the initialization routine of the mass storage class driver. The mass storage class driver makes calls to the base class driver created, providing the match spec and dispatch tables are included as parameters to the function.

```
status = _NU_USBH_DRV_Create ((NU_USB_DRV *) cb, "USBH-MS",  
                             USB_MATCH_CLASS, 0, 0, 0, 0, 8  
                             /* Mass Storage Class Code */,  
                             0, 0, &usbh_ms_dispatch );
```


Logical Units

A single mass storage class interface may contain multiple addressable units called Logical Units; each identified by a Logical Unit Number (LUN). NU_USBH_MS supports multiple LUN's and, for each device connected, it finds out the number of LUN's on the device and notifies each user driver for each identified LUN.

Command Block Specification

Storage devices are usually connected to peripheral buses in a system. A command code scheme defines the sequence of control information and data transmitted on those buses that is understood by the media controller on the media. There are many different industry standards for command code schemes owing to the existence of various types of peripheral interfaces.

USB architecture provides conventional storage devices plug-n-play capability by acting as a bridge between peripheral interfaces. Different industry standardized command sets are described by SubClass codes of the mass storage class. Note that the SubClass does not specify the type of storage device. A list of supported SubClass codes can be found in the [Table 8-1](#).

Table 8-1. USB Host Driver Command Block Specification

SubClass Code	Command Block Specification	Comment
01h	Reduced Block Commands (RBC) T10 Project 1240-D	Typically, a FLASH device uses RBC command blocks. However, any Mass Storage device can use RBC command blocks.
02h	SFF-8020i, MMC-2 (ATAPI)	Typically, a CD/DVD device uses SFF-8020i or MMC-2 command blocks for its mass storage interface.
03h	QIC-157	Typically, a tape device uses QIC-157 command blocks.
04h	UFI	Typically, a floppy disk drive (FDD) device.
05h	SFF-8070i	Typically, a floppy disk drive (FDD) device uses SFF-8070i command blocks. However, an FDD device can be in another subclass (for example, RBC) and other types of storage devices can belong to the SFF-8070i subclass.
06h	SCSI transparent command set.	
07h-FFh	Reserved for future use.	

Since the handlers for those command codes are independent of USB, NU_USBH_MS uses the user layer to drive them. When a new mass storage device is connected, NU_USBH_MS issues a connection notification to each registered user whose SubClass matches with the SubClass code of the interface, until it finds a user driver that successfully completes its connection call. Note that NU_USBH_MS makes selections based on the SubClass of the user drivers registered with it. It ignores protocol codes of the registered user drivers

Note

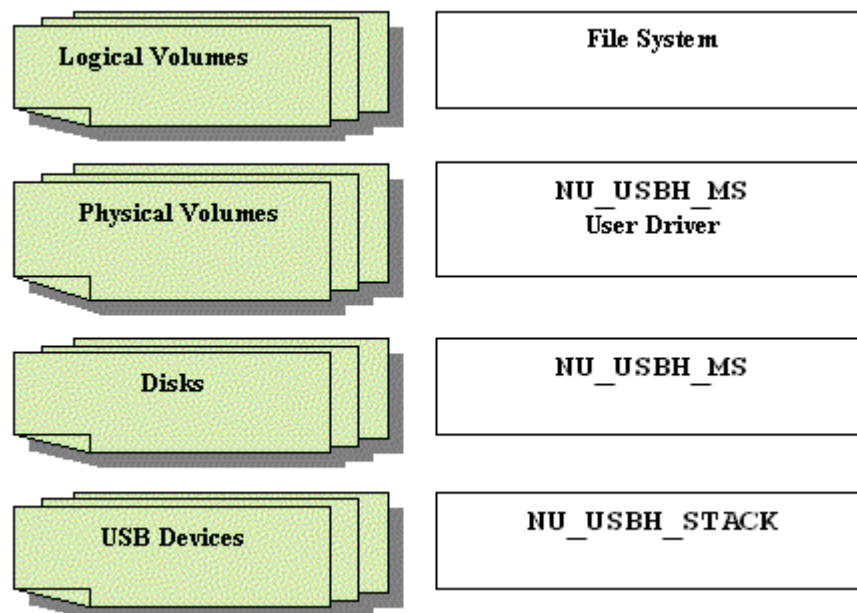


SubClass and Protocol codes are assigned to a user driver during its creation.

A SubClass code of 0 means any SubClass for NU_USBH_MS. If a user driver prefers to handle all SubClasses by itself and wishes to get a connection notification for all types of mass storage devices, it can create the user driver with a subclass code of 0x00.

Figure 8-2 describes the view of each layer for mass storage devices in Nucleus USB Host.

Figure 8-2. USB Host Driver Diagram



The Nucleus USB Host Stack recognizes a mass storage device as any other USB device. Since NU_USBH_MS is a matching driver, it owns the device. NU_USBH_MS, when notified of the device, detects the number of LUNs on the device. Each LUN is viewed by NU_USBH_MS as a disk. A suitable user driver (based on its SubClass code) gains control of each of these disks. The user driver(s) discover the disk geometry and create their view of them as physical volumes. A physical volume is an addressable byte array. Using the mass storage user driver APIs, the file system component recognizes logical volumes on the media.

Writing a User Driver for NU_USBH_MS

Developing a user driver involves the same steps explained in “[Developing a Communication User Driver](#)” on page 345. The user driver must implement all SubClass responsibilities of the mass storage class. It may also require following the file system’s driver interface, if any, since it acts as the glue for the file system. The transport API provided by NU_USBH_MS should be used for transporting all SubClass commands.

The following is an example of a user driver, which registers with NU_USBH_MS using the SubClass code as 01 such as, Reduced Block Commands (RBC).

```
STATUS NU_USBH_MS_USER_RBC_Create (NU_USBH_USER  *cb,
                                   NU_MEMORY_POOL *pool )
{
    STATUS status;

    /* Create base. */
    status = _NU_USBH_USER_Create(cb, "MS-RBC", pool, 01, 0xFF,
                                   /*protocol is don't care */
                                   rbc_user_dispatch);

    if (NU_SUCCESS != status)
        err_hndlr();
}
```

Our example RBC user overrides the connect callback and does not need to override any of the other base functionality of NU_USBH_USER, so it fills in the dispatch table with default values. The dispatch table is given as follows.

```
const NU_USBH_USER_DISPATCH rbc_user_dispatch =
{
    {
        {
            _NU_USB_Delete, _NU_USB_Get_Name,
            _NU_USB_Get_Object_Id
        },
        NU_USBH_USER_RBC_Connect,
        _NU_USBH_USER_Disconnect
    },
    _NU_USBH_USER_Wait,
    _NU_USBH_USER_Open_Device,
    _NU_USBH_USER_Close_Device,
    _NU_USBH_USER_Remove_Device
};
```

The following is a code example for the Overridden connect callback by our user.

```
STATUS _NU_USBH_USER_RBC_Connect (NU_USBH_USER *cb,
                                   NU_USBH_DVR *class_driver,
                                   VOID          *handle)
{
    STATUS status;
    global_ms_cb = (NU_USBH_MS*)class_driver;
    global_device_handle = handle;
}
```

```
/* Call parent behavior */
status = _NU_USBH_USER_Connect (cb, class_driver, handle);
return (status);
}
```

The RBC user provides services to read and write raw sectors (512 bytes) from the mass storage device. It uses the RBC Read10 and Write10 commands (defined by the RBC specification) to do this. The following is example code for how an RBC user makes use of the NU_USBH_MS API to perform data transfer with the device. The services explained are for illustration purposes only and there may be several other services required from an RBC user driver.

```
STATUS NU_USBH_MS_USER_RBC_Read (NU_USBH_USER *rbc_user,
                                  UINT32 sector, VOID *buffer,
                                  UINT16 count)
{
    UINT8 command[10];
    command[0] = 28; /* Set the read command */
    command[1] = 0x00; /* Command Byte */
    command[2] = (UINT8)(sector >> 24); /* Reserved */
    command[3] = (UINT8)(sector >> 16); /* Sector Address */
    command[4] = (UINT8)(sector >> 8);
    command[5] = (UINT8)(sector);
    command[6] = 0x00; /* Reserved */
    command[7] = (UINT8)(count >> 8); /* number of sectors requested */
    command[8] = (UINT8)(count); /* to be transferred */
    command[9] = 0x00; /* Control Byte */

    status = NU_USBH_MS_Transport (global_ms_cb, rbc_user, handle,
                                   command, 10, buffer, 512*count,
                                   USB_DIR_IN);

    return status;
}

STATUS NU_USBH_MS_USER_RBC_Write (NU_USBH_USER *rbc_user,
                                   UINT32 sector, VOID *buffer,
                                   UINT16 count)
{
    UINT8 command[10];
    command[0] = 2A; /* Set the read command */
    command[1] = 0x00; /* Command Byte */
    command[2] = (UINT8)(sector >> 24); /* Reserved */
    command[3] = (UINT8)(sector >> 16); /* Sector Address */
    command[4] = (UINT8)(sector >> 8);
    command[5] = (UINT8)(sector);
    command[6] = 0x00; /* Reserved */
    command[7] = (UINT8)(count >> 8); /* number of sectors requested */
    command[8] = (UINT8)(count); /* to be transferred */
    command[9] = 0x00; /* Control Byte */

    status = NU_USBH_MS_Transport (global_ms_cb, rbc_user,
                                   handle, command, 10, buffer,
                                   512*count, USB_DIR_IN);

    return status;
}
```

```
}
```

Transport Service

The Nucleus USB Host Mass Storage Driver provides a single API called transport interface for data transfers. As stated in the previous chapter, SubClasses define a command sequence to be transmitted to mass storage devices, and they are handled outside of NU_USBH_MS.

NU_USBH_Transport_MS takes a command of any SubClass and transfers it to the device. The mechanism to transfer SubClass specific information is defined by bInterfaceProtocol.

NU_USBH_MS uses appropriate transfer protocol when transporting commands to the device, thus hiding the protocol specific information from mass storage user drivers (the SubClass drivers).

The following is an example, using the transport interface to send an INQUIRY command, which is a SCSI SubClass specific command, to a SCSI disk across USB to print the data returned by the disk in response.

```
/* Send a SCSI Inquiry Command to a SCSI Disk connected across
 * USB and print the Inquiry Data */
CHAR drivename[36]; /* To store the data returned by device */
UINT8      command[6],i;
NU_USBH_MS  ms;
NU_USBH_USER dummy_user;
VOID        *device_handle;

    command[0] = 0x12;      /* Inquiry command          */
    command[1] = 0x00;
    command[2] = 0x00;
    command[3] = 0x00;
    command[4] = 0x24;      /* Length of Inquiry Data */
    command[5] = 0x00;

    status = NU_USBH_Transport_MS(&ms, &dummy_user, device_handle,
                                command, 6, drivename, 36,
                                USB_DIR_IN
                                /*Device to Host*/);

    if (status !=NU_SUCCESS)
        error_hndlr( );
    else
    {
        PrintString("Inquiry Data - ");
        for(i=0;i<0x24;i++)
            PrintChar(drivename[i]);
    }
```

USB Host Driver Functions

This section provides a description of the APIs exported by NU_USBH_MS. The Nucleus USB provides only two API. A create or initialization call is meant for system integrators. A data transfer API is used by user drivers or other clients.

- [NU_USBH_MS_Create](#)
- [NU_USBH_MS_Transport](#)

NU_USBH_MS_Create

This function initializes the Nucleus Host Mass Storage Driver and makes it ready to be registered with a host stack.

Usage

```
STATUS NU_USBH_MS_Create (NU_USBH_MS      *cb,  
                           CHAR            *name,  
                           NU_MEMORY_POOL *pool)
```

Arguments

- **cb**
A Pointer to the mass storage driver control block.
- **name**
The name of the USB object.
- **pool**
Starting in Nucleus 3.2, this argument is no longer required and is present only for backward compatibility. NU_NULL should be passed in for pool.

Return Values

- **NU_SUCCESS**
Successful initialization.
- **NU_INVALID_SEMAPHORE**
Indicates the semaphore pointer is invalid.
- **NU_SEMAPHORE_DELETED**
The semaphore was deleted while the task was suspended.
- **NU_UNAVAILABLE**
Indicates the semaphore is unavailable.
- **NU_INVALID_SUSPEND**
Indicates that this API is called from a non-task thread.
- **NU_USB_INVLD_ARG**
Indicates that the driver value passed is NU_NULL, that the `drv->match_flag` contain invalid values, or that the `drv->initialize_device` function pointer is with the `USB_MATCH_NDR_ID` field set in the `drv->match_flag`.

Description

This function is the mass storage class driver initialization routine. It is designed for use by system integrators for creating mass storage class driver components on the host subsystem.

Related Topics

[USB Host Driver Functions](#)

NU_USBH_MS_Transport

This function transfers data across USB to a mass storage device.

Usage

```
STATUS NU_USBH_MS_Transport (NU_USBH_MS    *cb,
                             NU_USBH_USER  *user,
                             VOID          *session,
                             VOID          *command,
                             UINT8         cmd_length,
                             VOID          *data_buffer,
                             UINT32        data_length,
                             UINT8         direction)
```

Arguments

- **cb**
A pointer to mass storage driver control block.
- **user**
A pointer to user control block calling this API.
- **session**
A pointer to handle of a LUN on device, used to uniquely identify the device. (Provided to the user driver in Connect callback).
- **command**
The command to be transmitted to the device.
- **cmd_length**
The length of the command.
- **data_buffer**
A pointer to send/receive data buffer.
- **data_length**
The length of data to be sent/received.
- **direction**
The direction of data transfer.

Return Values

- **NU_SUCCESS**
Indicates successful completion.
- **NU_USB_MS_TRANSPORT_ERR**
Indicates command could not be executed.

- **NU_USB_MS_TRANSPORT_FAILED**
Indicates command failed by the media.

Description

NU_USBH_MS_Transport is used to transfer data across USB to a mass storage device. It takes a command, which is specific to the SubClass of the device and passes it to the device as per the bInterfaceProtocol of the interface.

Related Topics

[USB Host Driver Functions](#)

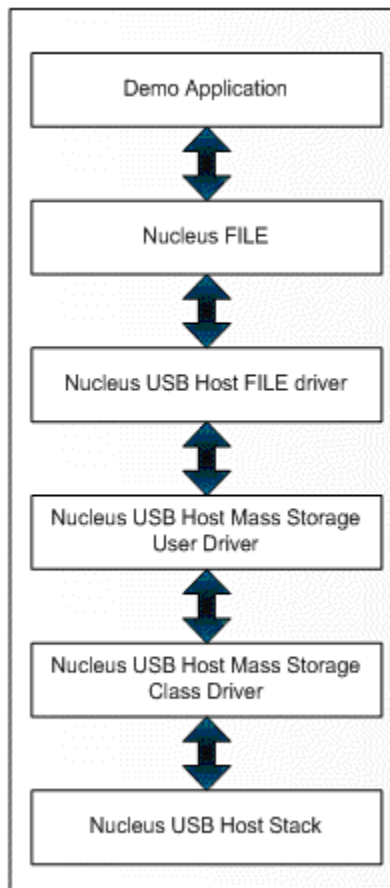
USB Host FILE Driver

USB Host FILE Driver subproduct is developed to allow Nucleus USB Host Mass Storage Class Driver to interact with Nucleus FILE. It is basically a transport layer between Nucleus FILE and Nucleus USB Mass Storage Class Driver.

Architecture

The architectural view for the USB Host FILE Driver subproduct includes, Nucleus USB Host Stack, Nucleus USB Host Mass Storage Class Driver, Nucleus USB Mass Storage User Driver, USB Host FILE Driver subproduct, and the demonstration. On a mass storage device connection, the USB FILE Driver subproduct sends mass storage class-specific commands to initialize the device. It then registers I/O open, close device handlers to Nucleus FILE.

Figure 8-3. Communication Between Host Mass Storage Class Driver and FILE



Nucleus FILE 3.1 Using USB Host FILE Driver Subproduct

Nucleus FILE 3.1 Using FILE USB Driver Subproduct or NU_USBH_NUF_DRVR is a glue layer that provides standard read/write/ioctl/open/close interface for the FILE middleware. It implements all those API's that are defined for a STORAGE_LABEL device.

Nucleus USB Host File driver registers one device per logical unit with the device manager and then its the responsibility of FILE to parse through the partition tables of the logical unit to determine and mount each partition present in that unit. All subclass drivers are derived from this component.

USB Host Driver and the Device Manager

USB Host storage is designed to work with Nucleus FILE, therefore each new device is reported to device manager through the File driver. Whenever a new storage device is plugged into the target's USB bus and is acquired by the USB storage driver and a new device is registered with the device manager with the following labels:

- *USBH_FILE_LABEL*
- *STORAGE_LABEL*
- *POWER_CLASS*

There are no specific IOCTLs defined against the *USBH_FILE_LABEL*.

For the IOCTL definition and usage of STORAGE_LABEL, refer to the [Nucleus Storage Guide](#). For the IOCTL definition and usage of POWER_CLASS, consult the power management services documentation.

The device will be de-registered with the DM and will go away as soon as the user unplugs it over the USB.

Each USB storage device will be registered as a new device with device manager with the above set of labels and will have the default configuration options defined in [Table 8-2](#).

Table 8-2. USB Storage Device Default Options

Name	Default Value	Type	Usage
pt	'A'	CHAR	Drive letter to associate with the device.
fs	"FAT"	String	String describing the file system type
auto_fmt	true	BOOLEAN	Formats the device if the file system is not present

USB Function Driver

The Nucleus USB Function Mass Storage Class Driver enables applications to create storage devices, which comply with the *USB Mass Storage Class Overview*. It supports BOT protocol for data transfers. The Nucleus USB Function Mass Storage Class Driver assigns command set handling to user/media drivers. In this way, it decouples command set interpretation, which is independent of both USB and mass storage class, to a separate layer. This aids in developing a user/media driver for a given command set, independently and efficiently by using mass storage driver services. In addition, it helps in reusing existing command set drivers, if any, in the system.

According to the *USB Mass Storage Class Overview*, mass storage class code is defined in the interface descriptor. Consequently, Nucleus USB Function Mass Storage Driver is an interface driver and owns an interface on a device. The terms SubClass and Protocol codes, used above are defined by bInterfaceSubclass and bInterface Protocol fields of interface descriptor of the device.

NU_USBF_MS

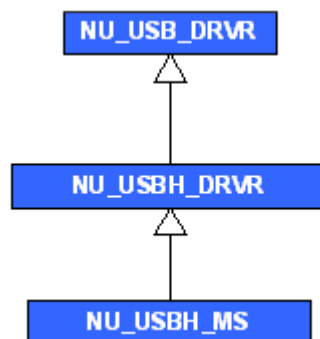
Like all other service components in Nucleus USB, Nucleus USB Function Mass Storage Class Driver is also identified in the system by its control block. This control block is termed as NU_USBF_MS.

NU_USBF_MS is initialized in the system using the API call [NU_USBF_MS_Create](#) by passing a reference and a name to the mass storage control block.

Function Mass Storage Component Hierarchy

[Figure 8-4](#) shows the class hierarchy diagram of the Nucleus USB Function Mass Storage Class Driver. As apparent from the figure, the Nucleus USB Function Mass Storage Class Driver is a specialization of base device class driver. It implements common class driver functionality in a way unique to USB mass storage class.

Figure 8-4. USB Function Mass Storage Class Driver Component Hierarchy



Due to this inheritance philosophy, the Nucleus USB Function Stack invokes base class driver API services on references to mass storage [extended] class driver control blocks. The stack obtains mass storage [extended] class driver control blocks through registration; the stack uses them as normal references to [base] class driver control blocks.

Mass Storage Function Match Spec

The Nucleus USB Function Mass Storage Class Driver is an interface level driver. As stated earlier, this class driver handles the transport protocols, such as BOT, and user drivers are required to handle SubClasses, for example, command sets such as SCSI. Therefore, it only provides bInterfaceClass in match spec. The class code for mass storage as defined by the USB Forum is 8.

The following is a code example from the initialization routine of the mass storage class driver. The mass storage class driver makes calls to the base class driver created, providing the match spec and dispatch tables are included as parameters to the function.

```
/* Invoke the parents create function */
status = _NU_USBH_USER_Create (&cb->parent, name,
                               (USB_MATCH_CLASS), 0, 0, 0, 0,
                               (USBF_MS_CLASS), 0, 0,
                               &usbf_ms_dispatch);
```

USB Function Driver Functions

This section describes the available API that is accessible by the Nucleus USB Function Mass Storage Class Driver.

- [NU_USBF_MS_Create](#)
- [NU_USBF_MS_Create_Task_Mode](#)
- [NU_USBF_MS_Start_Cmd_Processing](#)

NU_USBFS_MS_Create

This function initializes the mass storage class driver when you need to process commands not in task mode. In this case mass storage command processing would be done in HISR mode.

Usage

```
STATUS NU_USBFS_MS_Create (NU_USBFS_MS *pcb_ms_drvr,  
                           CHAR          *p_name)
```

Arguments

- `pcb_ms_drvr`
Pointer to mass storage control block.
- `p_name`
Name of the class driver.

Return Values

- `NU_SUCCESS`
Indicates successful completion.
- `NU_USB_INVLD_ARG`
Indicates incorrect arguments.
- `NU_NOT_PRESENT`
Indicates a configuration problem because of which no more USB objects could be created.

Related Topics

[USB Function Driver Functions](#)

NU_USBF_MS_Create_Task_Mode

This function initializes the mass storage class driver when you need to process commands in task mode. A pointer to the memory pool is taken as input that is used for creating a task initializing command processing task.

Usage

```
STATUS NU_USBF_MS_Create_Task_Mode (NU_USBF_MS      *pcb_ms_drvr,
                                     CHAR             *p_name,
                                     NU_MEMORY_POOL    *p_mem_pool)
```

Arguments

- **pcb_ms_drvr**
 Pointer to mass storage control block.
- **p_name**
 Name of the class driver.
- **p_mem_pool**
 Starting in Nucleus 3.2, this argument is no longer required and is present only for backward compatibility. NU_NULL should be passed in for p_mem_pool.

Return Values

- **NU_SUCCESS**
 Device handle found.
- **NU_USB_INVLD_ARG**
 Indicates incorrect arguments.
- **NU_NOT_PRESENT**
 Indicates a configuration problem because of which no more USB objects could be created.

Related Topics

[USB Function Driver Functions](#)

NU_USBF_MS_Start_Cmd_Processing

This function processes mass storage commands in task mode instead of HISR.

Usage

```
STATUS NU_USBF_MS_Start_Cmd_Processing (NU_USBF_MS_DEVICE *pcb_ms_device)
```

Arguments

- `pcb_ms_device`
Pointer to mass storage device control block.

Return Values

- `NU_SUCCESS`
Indicates successful completion.
- `NU_USB_INVLD_ARG`
Indicates incorrect arguments.

Description

This function should be used only when we need to process mass storage commands in task mode instead of HISR. In that case this function would enable mass storage device to start receiving mass storage class specific commands from USB Host. This function should be in the context of connection callback in demo application.

Related Topics

[USB Function Driver Functions](#)

Function Subclass Drivers

The USB Class Driver supports various SubClasses defined by the *USB Mass Storage Class Overview*. A single mass storage user driver supports these SubClasses. Nucleus USB Function Mass Storage Class Driver software supports multiple mass storage devices at a time. Each type of media may have the same SubClass code, but may differ in how the subclass is supported. For instance, a USB hard drive may support SCSI commands in the hardware where as a USB FLASH drive may require SCSI command support in the software. Each type of media has different media drivers. The user drivers are the containers for the media. The media drivers implement the command set in a media specific way.

In the above-mentioned example, the hard disk media driver is different from the media driver for the FLASH.

NU_USBF_USER_MS

All subclass drivers for the mass storage class driver are direct descendents of the NU_USBF_USER_MS component. The following is its dispatch table.

```
typedef struct _nu_usbf_user_ms_dispatch
{
    NU_USBF_USER_DISPATCH dispatch;
    STATUS (*Reset) (NU_USB_USER * cb, NU_USB_DRV * drvr, VOID *handle);
}
NU_USBF_USER_MS_DISPATCH;
```

The Reset callback cancels all transactions with the host and returns the media to a stable state. The subclass drivers must define the reset callback to achieve the same result. No default implementation is provided by the NU_USBF_USER_MS component.

USB Function User Driver Functions

The following function reference contains all Nucleus USB Function User Mass Storage services.

- [_NU_USBF_USER_MS_Create](#)
- [_NU_USBF_USER_MS_Delete](#)
- [NU_USBF_USER_MS_Get_Max_LUN](#)
- [NU_USBF_USER_MS_Reset](#)

_NU_USBFS_USER_MS_Create

This function initializes the mass storage function user driver. The user driver is characterized by the SubClass of the mass storage class that it supports.

Usage

```
STATUS _NU_USBFS_USER_MS_Create (NU_USBFS_USER_MS *pcb_user_ms,
                                CHAR                *p_name,
                                UINT8               subclass,
                                UINT8               max_lun,
                                const VOID          *p_dispatch)
```

Arguments

- **pcb_user_ms**
Pointer to the function mass storage user control block.
- **p_name**
Pointer to a seven-character name for the mass storage class driver. The name must be null-terminated.
- **subclass**
USB mass storage SubClass supported by this driver.
- **max_lun**
Maximum LUN supported by this driver.
- **p_dispatch**
Dispatch table.

Return Values

- **NU_SUCCESS**
Indicates successful initialization.
- **NU_USB_INVLD_ARG**
Indicates an invalid argument.
- **NU_NOT_PRESENT**
Indicates no more USB objects could be created due to a configuration error.

Related Topics

[USB Function User Driver Functions](#)

_NU_USBF_USER_MS_Delete

This function releases any resources that were allocated by the [_NU_USBF_USER_MS_Create](#) function. This function must be called from the delete routine of the component that derives from NU_USBF_USER_MS.

Usage

```
STATUS _NU_USBF_USER_MS_Delete (NU_USB *pcb_usb)
```

Arguments

- `pcb_usb`
Pointer to the mass storage user driver control block.

Return Values

- `NU_SUCCESS`
Indicates successful uninitialization.

Related Topics

[USB Function User Driver Functions](#)

NU_USB_USER_MS_Get_Max_LUN

This function retrieves the maximum Logical Unit Number supported by the storage device.

Usage

```
STATUS NU_USB_USER_MS_Get_Max_LUN (const NU_USB_USER *pcb_user,
                                   const NU_USB_DRV *p_drvr,
                                   const VOID *p_handle,
                                   UINT8 *p_max_lun_out)
```

Arguments

- **pcb_user**
Pointer to the mass storage user driver control block.
- **p_drvr**
Pointer to associated mass storage class driver control block.
- **p_handle**
Unique mass storage device identification (obtained by the user driver in its Connect callback).
- **p_max_lun_out**
Pointer to a variable to hold the value of the max LUN.

Return Values

- **NU_SUCCESS**
Indicates successful initialization.
- **NU_USB_INVLD_ARG**
Indicated invalid parameter.

Related Topics

[USB Function User Driver Functions](#)

NU_USBFS_USER_MS_Reset

This function resets the user driver, stalls all the ongoing transfers, clears any errors, and brings the device to a stable state.

Usage

```
STATUS NU_USBFS_USER_MS_Reset (NU_USB_USER *pcb_user_ms,  
                               NU_USB_DRIVER *p_drvr,  
                               VOID          *p_handle)
```

Arguments

- **pcb_user_ms**
Pointer to the mass storage user driver control block.
- **p_drvr**
Pointer to the associated mass storage class driver control block.
- **p_handle**
Unique mass storage device identification.

Return Values

- **NU_SUCCESS**
Indicates successful initialization.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVALID_ARG**
Indicates an invalid argument.

Related Topics

[USB Function User Driver Functions](#)

USB Function Device Manager

USB Function storage is a standalone driver and is not opened by any middleware internally during initialization time. Whenever an application opens it, it creates the storage descriptors in the stack and becomes ready for a USB connection with USB host. It is registered with the device manager with the label:

- USBF_STORE_LABEL

An application must first obtain the IOCTL0 base with the USBF_STORE_LABEL that is USB_STORE_IOCTL_BASE. Here is the list of other IOCTLs defined against this label.

Table 8-3. USB_STORE_IOCTL_BASE IOCTLs

IOCTL# OFFSET	USBF_SCSI_MEDIA_SET_CALLBACKS
Description	Registers a set of callback function which should be invoked when the usb storage driver sees a read or write command from the host.
IOCTL Parameters	NU_USBFS_MS_CALLBACKS type of function pointer. Its details are given in the section below.
Include File	<i>os/include/connectivity/nu_usb.h</i>
Parameters	Described under.
Notes	The structure is defined and populated with callback routines all in application.

NU_USBFS_MS_CALLBACKS

This is a structure containing different function pointers of application required and called by the storage function driver to report certain events. An application must implement all of these functions if there is any chance of calling any from the storage driver.

```
typedef struct _nu_usbfs_ms_callbacks
{
    STATUS (*Read) (
        NU_USBFS_SCSI_MEDIA *context,
        UINT32                sector_start_addr,
        UINT8                 *cmd_pointer,
        UINT32                sector_count);

    STATUS (*Write) (
        NU_USBFS_SCSI_MEDIA *context,
        UINT32                sector_start_addr,
        UINT8                 *cmd_pointer,
        UINT32                sector_count);

    STATUS (*conn_disconn) (
        VOID* handle,
        UINT8 event);
}NU_USBFS_MS_CALLBACKS;
```

Table 8-4 shows the NU_USBFS_MS_CALLBACKS descriptions.

Table 8-4. NU_USBFS_MS_CALLBACKS Descriptions

Return Type	Name	Description
Status	Read	Called by the storage driver when it receives a SCSI READ command from the USB host.
Status	Write	Called by the storage driver when it receives a SCSI WRITE command from the USB host.
Status	Conn_disconn	Reporting the connection and disconnection of the device with the USB host.

Table 8-5 shows the USBFS SCSI_MEDIA_SET_LUN_SIZE descriptions.

Table 8-5. USBFS SCSI_MEDIA_SET_LUN_SIZE

IOCTL# OFFSET	USBFS SCSI_MEDIA_SET_LUN_SIZE
Description	Sets the total sector count of the media.
IOCTL Parameters	UINT32* holding the 32-bit total sector count
Include File	<i>os/include/connectivity/nu_usb.h</i>
Parameters	None
Notes	None

Table 8-6 shows the USBFS SCSI_MEDIA_SET_BLOCK_SIZE descriptions.

Table 8-6. USBFS SCSI_MEDIA_SET_BLOCK_SIZE

IOCTL# OFFSET	USBFS SCSI_MEDIA_SET_BLOCK_SIZE
Description	Sets the bytes per sector count of the media.
IOCTL Parameters	UINT32* holding the 32-bit byte/sector value which is 512 bytes in most of cases
Include File	<i>os/include/connectivity/nu_usb.h</i>
Parameters	None
Notes	None

Table 8-7 shows the USBFS SCSI_SET_INQUIRY_DATA descriptions.

Table 8-7. USBFS SCSI_SET_INQUIRY_DATA Descriptions

IOCTL# OFFSET	USBFS SCSI_SET_INQUIRY_DATA
Description	For each storage media application must provide the SCSI Inquiry command data. For details and format of Inquiry command data see <i>SCSI primary commands</i> .

Table 8-7. USBF_SCSI_SET_INQUIRY_DATA Descriptions (cont.)

IOCTL# OFFSET	USBF_SCSI_SET_INQUIRY_DATA
IOCTL Parameters	UINT8*
Include File	<i>os/include/connectivity/nu_usb.h</i>
Parameters	A 8-bit pointer to Inquiry data. Since this IOCTL just sets the pointer value so data should remain valid in the application as long as device is in connected state.
Notes	None

Table 8-8 shows the USBF_SCSI_SET_CAPACITY_DATA descriptions.

Table 8-8. USBF_SCSI_SET_CAPACITY_DATA Descriptions

IOCTL# OFFSET	USBF_SCSI_SET_CAPACITY_DATA
Description	For each storage media application must provide the SCSI Read Capacity return data. For details and format of read capacity command data please see SCSI block commands.
IOCTL Parameters	UINT8*
Include File	<i>os/include/connectivity/nu_usb.h</i>
Parameters	A 8-bit pointer to capacity data. Since this IOCTL just sets the pointer value so data should remain valid in the application as long as device is in connected state.
Notes	None

Table 8-9 shows the USBF_SCSI_INSERT_MEDIA descriptions.

Table 8-9. USBF_SCSI_INSERT_MEDIA Descriptions

IOCTL# OFFSET	USBF_SCSI_INSERT_MEDIA
Description	Nucleus supports runtime insertion and removal of media when modeled as USB storage function. Application should execute this IOCTL when it sees a media ready to be accessed over USB by Host.
IOCTL Parameters	None
Include File	<i>os/include/connectivity/nu_usb.h</i>
Parameters	None
Notes	This IOCTL is only valid when application has set the storage type as removable in Inquiry data. By default a media is always in removed status so application must call this IOCTL for any media to be available over USB.

Table 8-10 shows the USBF_SCSI_REMOVE_MEDIA descriptions.

Table 8-10. USBF_SCSI_REMOVE_MEDIA Descriptions

IOCTL# OFFSET	USBF_SCSI_REMOVE_MEDIA
Description	Nucleus supports runtime insertion and removal of media when modeled as USB storage function. Application should execute this IOCTL when it wants a media to be unavailable for USB Host.
IOCTL Parameters	None
Include File	<i>os/include/connectivity/nu_usb.h</i>
Parameters	None
Notes	None

Table 8-11 shows the specific configuration options defined for these type of devices.

Table 8-11. Device Configuration Options

Name	Default Value	Type	Usage
configstring	"MS Configuration String"	String	This parameter is used to pass optional configuration string which will appear in the configuration descriptor of the device.
interfacestring	"MS Interface String"	String	This parameter is used to pass optional interface string which will appear in the interface descriptor of the device.
max_lun	1	UINT8	This parameter is used to pass maximum LUN information

Nucleus USB Function User SCSI Driver

NU_USBF_USER_SCSI

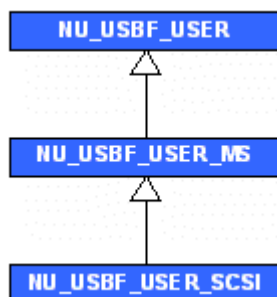
The NU_USBF_USER_SCSI component is a derivative of the NU_USBF_USER_MS component that acts as a container for holding SCSI media.

NU_USBF_USER_SCSI is initialized in the system using the API call [NU_USBF_USER_SCSI_Create](#) by passing a reference to the control block, name and the maximum number of media to be supported by this container.

Function SCSI Component Hierarchy

The figure below shows the class hierarchy diagram for NU_USBF_USER_SCSI. As apparent from the figure, NU_USBF_USER_SCSI is a specialization of NU_USBF_USER_MS. It implements common mass storage user driver functionality in a way unique to SCSI devices. Additional services required to access the container are also provided.

Figure 8-5. USB Function SCSI User Driver



USB Function User SCSI Driver Functions

The following function reference contains all Nucleus USB Function User SCSI services.

- [NU_USBF_USER_SCSI_Create](#)
- [NU_USBF_USER_SCSI_Delete](#)
- [NU_USBF_USER_SCSI_Dereg_Media](#)
- [NU_USBF_USER_SCSI_Get_Class_Handl](#)
- [NU_USBF_USER_SCSI_Reg_Media](#)

NU_USBFS_USER_SCSI_Create

This function initializes the SCSI media container to hold a given number of media.

Usage

```
STATUS NU_USBFS_USER_SCSI_Create (NU_USBFS_USER_SCSI *pcb_user_scsi,  
                                CHAR *p_name,  
                                UINT8 max_lun)
```

Arguments

- **pcb_user_scsi**
Pointer to the SCSI media container control block.
- **p_name**
Pointer to a seven-character name for the SCSI media container. The name must be null-terminated.
- **max_lun**
Maximum SCSI LUN to be supported by the container.

Return Values

- **NU_SUCCESS**
Indicates successful initialization.
- **NU_USB_INVLD_ARG**
Indicates invalid arguments.
- **NU_NOT_PRESENT**
Indicates no more USB objects could be created due to a configuration error.

Related Topics

[Nucleus USB Function User SCSI Driver](#)

NU_USBF_USER_SCSI_Delete

This function releases resources, if any, allocated by the [_NU_USBF_USER_MS_Create](#) function. This function must be called from the delete routine of the component that derives from NU_USBF_USER_MS.

Usage

```
STATUS NU_USBF_USER_SCSI_Delete (VOID *pcb)
```

Arguments

- `pcb`
Container to be un-initialized.

Return Values

- `NU_SUCCESS`
Indicates successful initialization.
- `NU_USB_INVLD_ARG`
Invalid argument.

Related Topics

[Nucleus USB Function User SCSI Driver](#)

NU_USBFS_USER_SCSI_Dereg_Media

This function deregisters SCSI media with this container.

Usage

```
STATUS NU_USBFS_USER_SCSI_Dereg_Media (NU_USBFS_USER_SCSI  *pcb_user_scsi,  
                                       NU_USBFS_SCSI_MEDIA *p_media)
```

Arguments

- **pcb_user_scsi**
Pointer to SCSI container control block.
- **p_media**
Pointer to Media to be deregistered with the container.

Return Values

- **NU_SUCCESS**
Indicates successful deregistration.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVLD_ARG**
Indicates an invalid argument.

Related Topics

[Nucleus USB Function User SCSI Driver](#)

NU_USBFS_USER_SCSI_Get_Class_Handl

This function returns a pointer to the device control block structure. This is helpful while calling class driver API.

Usage

```
STATUS NU_USBFS_USER_SCSI_Get_Class_Handl (  
    const NU_USBFS_USER_SCSI *pcb_user_scsi,  
    const NU_USBFS_SCSI_MEDIA *pcb_scsi_media,  
    VOID **pcb_ms_device)
```

Arguments

- `pcb_user_scsi`
Pointer to the SCSI user driver control block.
- `pcb_scsi_media`
Pointer to the SCSI media control block.
- `pcb_ms_device`
Return pointer for device control block structure.

Return Values

- `NU_SUCCESS`
Device handle found.
- `NU_NOT_PRESENT`
Device handle not found.
- `NU_USB_INVLD_ARG`
Indicates invalid parameter.

Related Topics

[Nucleus USB Function User SCSI Driver](#)

NU_USBF_USER_SCSI_Reg_Media

This function registers SCSI media with this container. The media exports functions that process individual SCSI commands.

Usage

```
STATUS NU_USBF_USER_SCSI_Reg_Media (NU_USBF_USER_SCSI  *pcb_user_scsi,  
                                     NU_USBF_SCSI_MEDIA *p_media)
```

Arguments

- `pcb_user_scsi`
SCSI container control block.
- `p_media`
Pointer to the control block of the SCSI media that's being registered.

Return Values

- `NU_SUCCESS`
Indicates successful registration.
- `NU_USB_NOT_SUPPORTED`
Indicates that the media does not support this event.
- `NU_USB_INVLD_ARG`
Indicates an invalid argument.

Related Topics

[Nucleus USB Function User SCSI Driver](#)

Media Drivers and NU_USBF_USER_SCSI

As mentioned earlier, the NU_USBF_USER_SCSI component is a container for holding SCSI media. The media drivers are registered with the SCSI container through the NU_USBF_USER_SCSI_Register_Media() and NU_USBF_USER_SCSI_Deregister_Media() APIs provided. These APIs are explained in the next chapter, along with examples.

SCSI Container Configuration Parameters

The following are the configuration macros for the Nucleus USB Function SCSI Container SubClass Driver. These macros can be found in the header file *nu_usbf_user_scsi_cfg.h*. The default values set for these macros are suitable for most of the applications. However, they can be changed to suit the application needs as required.

Table 8-12. SCSI Container Configuration Parameters

Macro	Suitable Range of Values	Description
NU_USBF_USER_SCSI_NUM_DISKS	1 - 15	Number of USB SCSI Storage disks intended to be created using the mass storage class driver.
NU_USBF_MS_TASK_STACK_SIZE	4096*100	Mass Storage command processing task stack size.
NU_USBF_MS_TASK_PRIORITY	0x01	Priority of command processing task.
NU_USBF_MS_TASK_MODE	Not Applicable	If the macro is defined, mass storage would work in task mode otherwise in HISR mode.
NU_USBF_MS_TASK_PREEMPTION	NU_PREEMPT OR NO_PREMPT	This is also configurable, you can change the value of NU_USBF_MS_TASK_PREEMPTION to NU_PREEMPT OR NO_PREMPT.

Function Media Drivers

Various USB Storage media may differ in how the command set is supported by the media. For instance, a USB hard drive may support SCSI commands in the hardware where as a USB FLASH drive may require SCSI command support in the software. Each type of media has different media drivers. The media drivers implement the command set in a media-specific way. In this example, the hard disk media driver is different from the media driver for the FLASH.

The media drivers are specific to the SubClass driver container. For example, the media drivers that exist for the NU_USBF_USER_SCSI do not work with other subclass drivers.

Currently, media drivers exist only for the SCSI container.

NU_USBFS_SCSI_MEDIA

NU_USBFS_SCSI_MEDIA is the base component for the media drivers for the SCSI SubClass driver container. This component provides default implementations for majority of the mandatory SCSI commands for the block devices. The default implementations always provide an error free response to the initiating host. For example, when the host sends a SCSI request sense command, the default implementation returns data that indicates that no error exists at the device. The children of this component can override any of the default implementations to provide a behavior of their own. In addition, this component provides a way in which newer commands can be implemented.

The following is the dispatch table for the NU_USBFS_SCSI_MEDIA component.

```
typedef struct _nu_usbfs_scsi_media_dispatch
{
    NU_USB_DISPATCH usb_dispatch;

    STATUS (*Connect) (
        const NU_USBFS_SCSI_MEDIA *pcb_scsi_media,
        const NU_USBFS_USER_SCSI *pcb_user_scsi);

    STATUS (*Disconnect) (
        const NU_USBFS_SCSI_MEDIA *pcb_scsi_media,
        const NU_USBFS_USER_SCSI *pcb_user_scsi);

    STATUS (*New_Transfer) (
        NU_USBFS_SCSI_MEDIA *pcb_scsi_media,
        const NU_USBFS_USER_SCSI *pcb_user_scsi,
        UINT8 **pp_buf_out,
        UINT32 *p_data_length_out);

    STATUS (*Tx_Done) (
        NU_USBFS_SCSI_MEDIA *pcb_scsi_media,
        const NU_USBFS_USER_SCSI *pcb_user_scsi,
        const UINT8 *p_completed_data,
        const UINT32 completed_data_length,
        UINT8 **pp_buf_out,
        UINT32 *p_data_length_out);

    STATUS (*Ready) (
        NU_USBFS_SCSI_MEDIA *pcb_scsi_media,
        const NU_USBFS_USER_SCSI *pcb_user_scsi,
        const UINT8 *p_cmd,
        const UINT16 cmd_len,
        UINT8 **pp_buf_out,
        UINT32 *p_data_length_out);

    STATUS (*Sense) (
        NU_USBFS_SCSI_MEDIA *pcb_scsi_media,
        const NU_USBFS_USER_SCSI *pcb_user_scsi,
        const UINT8 *p_cmd,
        const UINT16 cmd_len,
        UINT8 **pp_buf_out,
        UINT32 *p_data_length_out);
}
```

```

STATUS (*Inquiry) (
    NU_USBFS_SCSI_MEDIA          *pcb_scsi_media,
    const NU_USBFS_USER_SCSI     *pcb_user_scsi,
    const UINT8                  *p_cmd,
    const UINT16                  cmd_len,
    UINT8                         **pp_buf_out,
    UINT32                        *p_data_length_out);

STATUS (*Mode_Sense_6) (
    NU_USBFS_SCSI_MEDIA          *pcb_scsi_media,
    const NU_USBFS_USER_SCSI     *pcb_user_scsi,
    const UINT8                  *p_cmd,
    const UINT16                  cmd_len,
    UINT8                         **pp_buf_out,
    UINT32                        *p_data_length_out);

STATUS (*Mode_Sel6) (
    NU_USBFS_SCSI_MEDIA          *pcb_scsi_media,
    const NU_USBFS_USER_SCSI     *pcb_user_scsi,
    const UINT8                  *p_cmd,
    const UINT16                  cmd_len,
    UINT8                         **pp_buf_out,
    UINT32                        *p_data_length_out);

STATUS (*Snd_Diag) (
    NU_USBFS_SCSI_MEDIA          *pcb_scsi_media,
    const NU_USBFS_USER_SCSI     *pcb_user_scsi,
    const UINT8                  *p_cmd,
    const UINT16                  cmd_len,
    UINT8                         **pp_buf_out,
    UINT32                        *p_data_length_out);

STATUS (*Format) (
    NU_USBFS_SCSI_MEDIA          *pcb_scsi_media,
    const NU_USBFS_USER_SCSI     *pcb_user_scsi,
    const UINT8                  *p_cmd,
    const UINT16                  cmd_len,
    UINT8                         **pp_buf_out,
    UINT32                        *p_data_length_out);

STATUS (*Reserve_Unit) (
    NU_USBFS_SCSI_MEDIA          *pcb_scsi_media,
    const NU_USBFS_USER_SCSI     *pcb_user_scsi,
    const UINT8                  *p_cmd,
    const UINT16                  cmd_len,
    UINT8                         **pp_buf_out,
    UINT32                        *p_data_length_out);

STATUS (*Release_Unit) (
    NU_USBFS_SCSI_MEDIA          *pcb_scsi_media,
    const NU_USBFS_USER_SCSI     *pcb_user_scsi,
    const UINT8                  *p_cmd,
    const UINT16                  cmd_len,
    UINT8                         **pp_buf_out,
    UINT32                        *p_data_length_out);

STATUS (*Capacity) (
    NU_USBFS_SCSI_MEDIA          *pcb_scsi_media,

```

```

    const NU_USBF_USER_SCSI    *pcb_user_scsi,
    const UINT8                *p_cmd,
    const UINT16               cmd_len,
    UINT8                      **pp_buf_out,
    UINT32                     *p_data_length_out);

STATUS (*Verify) (
    NU_USBF_SCSI_MEDIA         *pcb_scsi_media,
    const NU_USBF_USER_SCSI    *pcb_user_scsi,
    const UINT8                *p_cmd,
    const UINT16               cmd_len,
    UINT8                      **pp_buf_out,
    UINT32                     *p_data_length_out);

STATUS (*Unknown_SCSI_Command) (
    NU_USBF_SCSI_MEDIA         *pcb_scsi_media,
    const NU_USBF_USER_SCSI    *pcb_user_scsi,
    const UINT8                *p_cmd,
    const UINT16               cmd_len,
    UINT8                      **pp_buf_out,
    UINT32                     *p_data_length_out);

STATUS (*Reset_Device) (
    NU_USBF_SCSI_MEDIA         *pcb_scsi_media,
    NU_USBF_USER_SCSI          *pcb_user_scsi);

STATUS (*Read) (
    NU_USBF_SCSI_MEDIA         *pcb_scsi_media,
    UINT32                     block_num,
    UINT8                      *buffer,
    UINT32                     length);

STATUS (*Write) (
    NU_USBF_SCSI_MEDIA         *pcb_scsi_media,
    UINT32                     block_num,
    UINT8                      *buffer,
    UINT32                     length);

STATUS (*Mode_Sense_10) (
    NU_USBF_SCSI_MEDIA         *pcb_scsi_media,
    const NU_USBF_USER_SCSI    *pcb_user_scsi,
    const UINT8                *p_cmd,
    const UINT16               cmd_len,
    UINT8                      **pp_buf_out,
    UINT32                     *p_data_length_out);

STATUS (*Mode_Sel_10) (
    NU_USBF_SCSI_MEDIA         *pcb_scsi_media,
    const NU_USBF_USER_SCSI    *pcb_user_scsi,
    const UINT8                *p_cmd,
    const UINT16               cmd_len,
    UINT8                      **pp_buf_out,
    UINT32                     *p_data_length_out);

STATUS (*Prevent-Allow) (
    NU_USBF_SCSI_MEDIA         *pcb_scsi_media,
    const NU_USBF_USER_SCSI    *pcb_user_scsi,
    const UINT8                *p_cmd,

```

```

const UINT16      cmd_len,
UINT8             **pp_buf_out,
UINT32            *p_data_length_out);

STATUS (*Start_Stop) (
    NU_USBFS_SCSI_MEDIA      *pcb_scsi_media,
    const NU_USBFS_USER_SCSI *pcb_user_scsi,
    const UINT8              *p_cmd,
    const UINT16             cmd_len,
    UINT8                    **pp_buf_out,
    UINT32                   *p_data_length_out);

}NU_USBFS_SCSI_MEDIA_DISPATCH;

```

The Connect and Disconnect callbacks are invoked when the device is connected to the host and disconnected from the bus respectively. The Reset_Device callback must bring the media to a stable state, cancel all outstanding transfers, and clear all error conditions. Read and Write callbacks must accomplish reading and writing blocks of data into the media. The New_Transfer callback is invoked when host initiates a data transfer in response to a previously sent SCSI command. The Tx_Done callback indicates completion of a data transfer. All other callbacks in the dispatch table have a one-to-one correspondence to a SCSI command. These callbacks are invoked when the host sends the corresponding command. The mapping is as follows:

Table 8-13. SCSI Command Mapping

Callback	SCSI Command
Sense	Request_Sense
Ready	Test_Unit_Ready
Mode_Sense_6	Mode_Sense (6 Byte)
Mode_Sense_10	Mode_Sense (10 Byte)
Mode_Sel6	Mode_Select (6 Byte)
Mode_Sel10	Mode_Select (10 Byte)
Inquiry	Inquiry
Snd_Diag	Send_Diagnostic
Format	Format_Unit
Reserve_Unit	Reserve_Unit
Release_Unit	Release_Unit
Capacity	Read_Capacity
Verify	Verify

READ (6), READ (10), READ (12), WRITE (6), WRITE (10), and WRITE (12) commands are internally supported by this component. If the Host sends any command that is not in the list, then the Unknown_Cmd callback is invoked.

Default implementations provided by this component for the callbacks are listed in [Table 8-14](#). As noted, these implementations offer limited functionality, always-indicating success to the host.

Table 8-14. SCSI Command Implementation

Callback	Default Implementation
Sense	_NU_USBF_SCSI_MEDIA_Sense
Ready	_NU_USBF_SCSI_MEDIA_Ready
Mode_Sense_6	_NU_USBF_SCSI_MEDIA_Mode_Sense (6 Byte)
Mode_Sense_10	_NU_USBF_SCSI_MEDIA_Mode_Sense (10 Byte)
Mode_Sel6	_NU_USBF_SCSI_MEDIA_Mode_Select (6 Byte)
Mode_Sel10	_NU_USBF_SCSI_MEDIA_Mode_Select (10 Byte)
Inquiry	_NU_USBF_SCSI_MEDIA_Inquiry
Snd_Diag	NU_USBF_SCSI_MEDIA_Snd_Diag
Format	_NU_USBF_SCSI_MEDIA_Format_Unit
Reserve	_Unit_NU_USBF_SCSI_MEDIA_Reserve_Unit
Release	_Unit_NU_USBF_SCSI_MEDIA_Release_Unit
Capacity	_NU_USBF_SCSI_MEDIA_Capacity
Verify	_NU_USBF_SCSI_MEDIA_Verify
Connect	_NU_USBF_SCSI_MEDIA_Connect
Disconnect	_NU_USBF_SCSI_MEDIA_Disconnect
New_Transfer	_NU_USBF_SCSI_MEDIA_New_Transfer
Tx_Done	_NU_USBF_SCSI_MEDIA_Tx_Done
Reset_Device	_NU_USBF_SCSI_MEDIA_Reset_Device

For the other callbacks, no default implementation is provided by this component.

_NU_USBF_SCSI_MEDIA_Sense always returns sense data indicating no error.

_NU_USBF_SCSI_MEDIA_Mode Select stores the data that is returned during a subsequent _NU_USBF_SCSI_MEDIA_ModeSense. Otherwise NU_USBF_SCSI_MEDIA_Mode Sense returns data indicating that no mode pages are supported.

_NU_USBF_SCSI_MEDIA_Inquiry always returns data indicating a block device with specified size.

_NU_USBF_SCSI_MEDIA_Snd_Diag always returns data indicating that the device is ready without any errors.

_NU_USBF_SCSI_MEDIA_Capacity always returns data indicating the size of the disk given during the component creation.

All other functions return success of execution of the specified command.

Developing SCSI Media Drivers

This section describes how new SCSI media drivers are created.

Creating Components

As noted above, all the SCSI media drivers are derived from the base SCSI media driver component `NU_USBFS SCSI_MEDIA`. Therefore, whenever a media driver is created, the component's create function must invoke the base create function `_NU_USBFS SCSI_MEDIA_Create`. This function accepts, as parameters, a pointer to the media driver control block, its name, pointers to the inquiry and capacity data, a Boolean variable indicating if the device is pre-formatted, and finally the over-riding dispatch table. The following code example demonstrates how this is achieved, in a sample RAM Disk driver.

```
size = 2048; /* Kilo Bytes */
memset (cb->inquiry, 0, sizeof (cb->inquiry));
memset (cb->capacity, 0, sizeof (cb->capacity));

/* Fill capacity data */
... ..

/* Fill Inquiry Data */
... ..

/* Call parent's create function */
return _NU_USBFS SCSI_MEDIA_Create (&cb->parent, "DISK", cb->inquiry,
                                     cb->capacity, NU_FALSE,
                                     &scsi_ram_dispatch);
```

Registering Media with SCSI Container

All the media drivers must be registered with the SCSI container SubClass driver. Once registered, the media will be eligible for communicating with the host. If a USB device contains multiple mass storage devices, these devices are enabled in the order in which they are registered with the container SubClass driver. The following code example demonstrates how a RAM Disk media is registered with the SCSI container.

```
/* Create the SCSI Container subclass driver */
NU_USBFS_USER SCSI_Create (&function_scsi, "demo", 0);

/* Create the RamDisk */
...

/* Register the RamDisk with the Container */
NU_USBFS_USER SCSI_Reg_Media (&function_scsi,
                              (NU_USBFS SCSI_MEDIA *) scsi_disk);
```

Providing Custom Response to Commands

The base SCSI media driver component provides the default implementations for SCSI command processing. It ensures that the communication between the host and the storage device never breaks due to an error returned by the device. In other words, an error is never returned to the host by these functions. Such behavior is acceptable in majority of situations where the devices are capable of recovering from the error automatically, and/or the media is not SCSI specific. (for example, a FLASH drive) However, this may not always be the desired response. Default implementations can be overridden to provide custom behavior in order to manage such requirements. The media drivers must provide specific behavior in the dispatch table registered. The following code example illustrates how this is achieved.

```
const NU_USBF_SCSI_MEDIA_DISPATCH scsi_ram_dispatch = {
{
    /* USB dispatch. */
    _NU_USBF_SCSI_MEDIA_Delete,
    _NU_USB_Get_Name,
    _NU_USB_Get_Object_Id,
},

    /* SCSI media dispatch. */
    scsi_ram_connect,
    scsi_ram_disconnect,
    _NU_USBF_SCSI_MEDIA_Transfer,
    _NU_USBF_SCSI_MEDIA_Tx_Done,
    _NU_USBF_SCSI_MEDIA_Ready,
    _NU_USBF_SCSI_MEDIA_Sense,
    _NU_USBF_SCSI_MEDIA_Inquiry,
    _NU_USBF_SCSI_MEDIA_Mode_Sense6,
    _NU_USBF_SCSI_MEDIA_Mode_Sel6,
    _NU_USBF_SCSI_MEDIA_Snd_Diag,
    _NU_USBF_SCSI_MEDIA_Format,
    _NU_USBF_SCSI_MEDIA_ReserveUnit,
    _NU_USBF_SCSI_MEDIA_ReleaseUnit,
    _NU_USBF_SCSI_MEDIA_Capacity,
    _NU_USBF_SCSI_MEDIA_Verify,
    _NU_USBF_SCSI_MEDIA_Command_23,
    _NU_USBF_SCSI_MEDIA_Reset_Device,
    scsi_ram_read,
    scsi_ram_write,
    _NU_USBF_SCSI_MEDIA_Mode_Sense_10,
    _NU_USBF_SCSI_MEDIA_Mode_Sel_10,
    _NU_USBF_SCSI_MEDIA_Prevent_Allow,
    _NU_USBF_SCSI_MEDIA_Start_Stop
    .....
    .....

    /* Later, during the component create function */
    status = _NU_USBF_SCSI_MEDIA_Create (&cb->parent, "DISK", cb->inquiry,
                                         cb->capacity, NU_FALSE,
                                         &scsi_ram_dispatch);
```

In this example, the RAM Disk driver overrides the default implementations for the Request Sense, Snd Diag, Verify, read, and the write callbacks.

Each of these command callbacks takes the following arguments:

- `pcb_scsi_media`
Media to which the host sent the command.
- `pcb_user_scsi`
The media container.
- `p_cmd`
Byte buffer containing the command block.
- `cmd_len`
Length, in bytes, of the command block.
- `pp_buf_out`
Memory location pointed by this variable must contain the data location, from where the data must be transferred.
- `p_data_len_out`
Location where the number of bytes of data to be transferred must be initialized.

Reading and Writing to Media

The base media driver does not provide any default implementations for the read and write callbacks. The read and write mechanisms are media specific and are accomplished in a media specific way. The base media driver uses these callbacks and implements the responses to the READ (6), READ (10), READ (12), WRITE (6), WRITE (10), and WRITE (12) commands internally.

USB Function SCSI Media Functions

The following function reference contains all Nucleus USB Function SCSI Media services. All APIs in this section should be called from the command received callback.

- [NU_USBF_SCSI_MEDIA_Capacity](#)
- [NU_USBF_SCSI_MEDIA_Connect](#)
- [NU_USBF_SCSI_MEDIA_Disconnect](#)
- [NU_USBF_SCSI_MEDIA_Format](#)
- [NU_USBF_SCSI_MEDIA_Inquiry](#)
- [NU_USBF_SCSI_MEDIA_Mode_Sel6](#)
- [NU_USBF_SCSI_MEDIA_Mode_Sel_10](#)
- [NU_USBF_SCSI_MEDIA_Mode_Sense_6](#)
- [NU_USBF_SCSI_MEDIA_Mode_Sense_10](#)
- [NU_USBF_SCSI_MEDIA_New_Transfer](#)
- [NU_USBF_SCSI_MEDIA_Prevent_Allow](#)
- [NU_USBF_SCSI_MEDIA_Read](#)
- [NU_USBF_SCSI_MEDIA_Read_6](#)
- [NU_USBF_SCSI_MEDIA_Read_10](#)
- [NU_USBF_SCSI_MEDIA_Read_12](#)
- [NU_USBF_SCSI_MEDIA_Ready](#)
- [NU_USBF_SCSI_MEDIA_Release_Unit](#)
- [NU_USBF_SCSI_MEDIA_Reserve_Unit](#)
- [NU_USBF_SCSI_MEDIA_Reset_Device](#)
- [NU_USBF_SCSI_MEDIA_Sense](#)
- [NU_USBF_SCSI_MEDIA_Snd_Diag](#)
- [NU_USBF_SCSI_MEDIA_Start_Stop](#)
- [NU_USBF_SCSI_MEDIA_Tx_Done](#)
- [NU_USBF_SCSI_MEDIA_Unknown_Cmd](#)
- [NU_USBF_SCSI_MEDIA_Verify](#)
- [NU_USBF_SCSI_MEDIA_Write](#)

- [NU_USBF_SCSI_MEDIA_Write_6](#)
- [NU_USBF_SCSI_MEDIA_Write_10](#)
- [NU_USBF_SCSI_MEDIA_Write_12](#)
- [_NU_USBF_SCSI_MEDIA_Create](#)
- [_NU_USBF_SCSI_MEDIA_Delete](#)
- [_NU_USBF_SCSI_MEDIA_Insert](#)
- [_NU_USBF_SCSI_MEDIA_Remove](#)

NU_USBF_SCSI_MEDIA_Capacity

This function processes the SCSI READ CAPACITY command. Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBF_SCSI_MEDIA_Capacity (
    NU_USBF_SCSI_MEDIA      *pcb_scsi_media,
    const NU_USBF_USER_SCSI *pcb_user_scsi,
    const UINT8              *p_cmd,
    const UINT16             cmd_len,
    UINT8                    **pp_buf_out,
    UINT32                   *p_data_len_out)
```

Arguments

- **pcb_scsi_media**
Media to which the host sent the command.
- **pcb_user_scsi**
The media container.
- **p_cmd**
Byte buffer containing the command block.
- **cmd_len**
Length, in bytes, of the command block.
- **pp_buf_out**
Memory location pointed by this variable must contain the data location, from where the data must be transferred.
- **p_data_len_out**
Location where the number of bytes of data to be transferred must be initialized.

Return Values

- **NU_SUCCESS**
Indicates successful processing of the command.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVLD_ARG**
Indicates incorrect command block.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBFS SCSI_MEDIA_Connect

Any hardware specific initializations that are required are done here. This event prepares the hardware for further transactions with the host. The implementation is media specific.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBFS SCSI_MEDIA_Connect (  
    const NU_USBFS SCSI_MEDIA *pcb_scsi_media,  
    const NU_USBFS_USER SCSI *pcb_user_scsi)
```

Arguments

- `pcb_scsi_media`
Media to be prepared for transactions.
- `pcb_user_scsi`
The media container.

Return Values

- `NU_SUCCESS`
Indicates successful processing of this event.
- `NU_USB_INVLD_ARG`
Indicates an invalid argument.
- `NU_USB_NOT_SUPPORTED`
Indicates that the media does not support this event.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBFS SCSI_MEDIA_Disconnect

This function processes the disconnection event. Any hardware specific de-initializations that are required are done here. The implementation is media specific.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBFS SCSI_MEDIA_Disconnect (  
    const NU_USBFS SCSI_MEDIA *pcb_scsi_media,  
    const NU_USBFS_USER SCSI *pcb_user_scsi)
```

Arguments

- `pcb_scsi_media`
Media to be prepared to put in quiescent state.
- `pcb_user_scsi`
The media container.

Return Values

- `NU_SUCCESS`
Indicates successful processing of this event.
- `NU_USB_INVLD_ARG`
Indicates an invalid argument.
- `NU_USB_NOT_SUPPORTED`
Indicates that the media does not support this event.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBF_SCSI_MEDIA_Format

This function returns the capacity list to the host.

Usage

```
STATUS NU_USBF_SCSI_MEDIA_Format (
    NU_USBF_SCSI_MEDIA      *pcb_scsi_media,
    const NU_USBF_USER_SCSI *pcb_user_scsi,
    const UINT8              *p_cmd,
    UINT16                   cmd_len,
    UINT8                    **pp_buf_out,
    UINT32                   *p_data_len_out)
```

Arguments

- **pcb_scsi_media**
Media to which the host sent the command.
- **pcb_user_scsi**
The media container.
- **p_cmd**
Byte buffer containing the command block.
- **cmd_len**
Length, in bytes, of the command block.
- **pp_buf_out**
Memory location pointed by this variable must contain the data location, from where the data must be transferred.
- **p_data_len_out**
Location where the number of bytes of data to be transferred must be initialized.

Return Values

- **NU_SUCCESS**
Indicates successful processing of the command.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVLD_ARG**
Indicates incorrect command block.

Description

This is the callback function for the mass storage Format capacities command. This is called when the host issues the Format Capacity command. You may extend this command according to the requirement.

Do not call this function directly unless you are extending this type.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBFS SCSI_MEDIA_Inquiry

This function processes the SCSI INQUIRY command.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBFS SCSI_MEDIA_Inquiry (  
    NU_USBFS SCSI_MEDIA          *pcb_scsi_media,  
    const NU_USBFS_USER SCSI    *pcb_user_scsi,  
    const UINT8                  *p_cmd,  
    const UINT16                 cmd_len,  
    UINT8                        **pp_buf_out,  
    UINT32                       *p_data_len_out)
```

Arguments

- **pcb_scsi_media**
Media to which the host sent the command.
- **pcb_user_scsi**
The media container.
- **p_cmd**
Byte buffer containing the command block.
- **cmd_len**
Length, in bytes, of the command block.
- **pp_buf_out**
Memory location pointed by this variable must contain the data location, from where the data must be transferred.
- **p_data_len_out**
Location where the number of bytes of data to be transferred must be initialized.

Return Values

- **NU_SUCCESS**
Indicates successful processing of the command.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVLD_ARG**
Indicates incorrect command block.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBFS SCSI_MEDIA_Mode_Sel6

This function processes the SCSI MODE SELECT (6) command.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBFS SCSI_MEDIA_Mode_Sel6 (
    NU_USBFS SCSI_MEDIA          *pcb_scsi_media,
    const NU_USBFS_USER SCSI    *pcb_user_scsi,
    const UINT8                  *p_cmd,
    const UINT16                 cmd_len,
    UINT8                        **pp_buf_out,
    UINT32                       *p_data_len_out)
```

Arguments

- **pcb_scsi_media**
Media to which the host sent the command.
- **pcb_user_scsi**
The media container.
- **p_cmd**
Byte buffer containing the command block.
- **cmd_len**
Length, in bytes, of the command block.
- **pp_buf_out**
Memory location pointed by this variable must contain the data location, from where the data must be transferred.
- **p_data_len_out**
Location where the number of bytes of data to be transferred must be initialized.

Return Values

- **NU_SUCCESS**
Indicates successful processing of the command.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVLD_ARG**
Indicates incorrect command block.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBFS SCSI_MEDIA_Mode_Sel_10

This function processes the SCSI MODE SELECT (10) command.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBFS SCSI_MEDIA_Mode_Sel_10 (  
    NU_USBFS SCSI_MEDIA          *pcb_scsi_media,  
    const NU_USBFS_USER SCSI    *pcb_user_scsi,  
    const UINT8                  *p_cmd,  
    const UINT16                 cmd_len,  
    UINT8                        **pp_buf_out,  
    UINT32                       *p_data_len_out)
```

Arguments

- **pcb_scsi_media**
Media to which the host sent the command.
- **pcb_user_scsi**
The media container.
- **p_cmd**
Byte buffer containing the command block.
- **cmd_len**
Length, in bytes, of the command block.
- **pp_buf_out**
Memory location pointed by this variable must contain the data location, from where the data must be transferred.
- **p_data_len_out**
Location where the number of bytes of data to be transferred must be initialized.

Return Values

- **NU_SUCCESS**
Indicates successful processing of the command.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVLD_ARG**
Indicates incorrect command block.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBF_SCSI_MEDIA_Mode_Sense_6

This function processes the SCSI MODE SENSE (6) command.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBF_SCSI_MEDIA_Mode_Sense_6 (
    NU_USBF_SCSI_MEDIA          *pcb_scsi_media,
    const NU_USBF_USER_SCSI     *pcb_user_scsi,
    const UINT8                  *p_cmd,
    const UINT16                 cmd_len,
    UINT8                         **pp_buf_out,
    UINT32                       *p_data_len_out)
```

Arguments

- **pcb_scsi_media**
Media to which the host sent the command.
- **pcb_user_scsi**
The media container.
- **p_cmd**
Byte buffer containing the command block.
- **cmd_len**
Length, in bytes, of the command block.
- **pp_buf_out**
Memory location pointed by this variable must contain the data location, from where the data must be transferred.
- **p_data_len_out**
Location where the number of bytes of data to be transferred must be initialized.

Return Values

- **NU_SUCCESS**
Indicates successful processing of the command.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVLD_ARG**
Indicates incorrect command block.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBFS SCSI_MEDIA_Mode_Sense_10

This function processes the SCSI MODE SENSE (10) command.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBFS SCSI_MEDIA_Mode_Sense_10 (  
    NU_USBFS SCSI_MEDIA          *pcb_scsi_media,  
    const NU_USBFS_USER SCSI    *pcb_user_scsi,  
    const UINT8                  *p_cmd,  
    const UINT16                 cmd_len,  
    UINT8                        **pp_buf_out,  
    UINT32                       *p_data_len_out)
```

Arguments

- **pcb_scsi_media**
Media to which the host sent the command.
- **pcb_user_scsi**
The media container.
- **p_cmd**
Byte buffer containing the command block.
- **cmd_len**
Length, in bytes, of the command block.
- **pp_buf_out**
Memory location pointed by this variable must contain the data location, from where the data must be transferred.
- **p_data_len_out**
Location where the number of bytes of data to be transferred must be initialized.

Return Values

- **NU_SUCCESS**
Indicates successful processing of the command.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVLD_ARG**
Indicates incorrect command block.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBFS SCSI_MEDIA_New_Transfer

This function processes a new transfer request from the host.

Usage

```
STATUS NU_USBFS SCSI_MEDIA_New_Transfer (
    NU_USBFS SCSI_MEDIA      *pcb_scsi_media,
    const NU_USBFS_USER SCSI *pcb_user_scsi,
    UINT8                    **pp_buf_out,
    UINT32                    *p_data_len_out)
```

Arguments

- **pcb_scsi_media**
Media to which the host requested a transfer
- **pcb_user_scsi**
The media container.
- **pp_buf_out**
Memory location pointed by this variable must contain the data location, from where the data must be transferred.
- **p_data_len_out**
Location where the number of bytes of data to be transferred must be initialized.

Return Values

- **NU_SUCCESS**
Indicates successful processing of this event.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVLD_ARG**
Indicates an invalid parameter.

Description

If any of the data is to be transferred to/from the host, the corresponding data pointer and the length will be set in the parameters. If no data transfer is required then the memory location pointed to by the buf_out parameter must be filled with NULL.

Do not call this function directly unless you are extending this type.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBFS SCSI_MEDIA_Prevent_Allow

This function processes the SCSI PREVENT ALLOW command.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBFS SCSI_MEDIA_Prevent_Allow (  
    NU_USBFS SCSI_MEDIA          *pcb_scsi_media,  
    const NU_USBFS_USER SCSI    *pcb_user_scsi,  
    const UINT8                  *p_cmd,  
    const UINT16                 cmd_len,  
    UINT8                        **pp_buf_out,  
    UINT32                       *p_data_len_out)
```

Arguments

- **pcb_scsi_media**
Media to which the host sent the command.
- **pcb_user_scsi**
The media container.
- **p_cmd**
Byte buffer containing the command block.
- **cmd_len**
Length, in bytes, of the command block.
- **pp_buf_out**
Memory location pointed by this variable must contain the data location, from where the data must be transferred.
- **p_data_len_out**
Location where the number of bytes of data to be transferred must be initialized.

Return Values

- **NU_SUCCESS**
Indicates successful processing of the command.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBFS SCSI_MEDIA_Read

This function reads length number of bytes from the media into the buffer.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBFS SCSI_MEDIA_Read (NU_USBFS SCSI_MEDIA *pcb_scsi_media,
                                UINT32                block,
                                UINT8                  *p_buffer,
                                UINT32                length)
```

Arguments

- **pcb_scsi_media**
The media to be read from.
- **block**
Block number to start from.
- **p_buffer**
Buffer to place the read data.
- **length**
Number of bytes to read.

Return Values

- **NU_SUCCESS**
Indicates a successful read operation.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVLD_ARG**
Indicates an invalid parameter.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBFS SCSI_MEDIA_Read_6

This function processes the SCSI READ (6) command.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBFS SCSI_MEDIA_Read_6 (
    NU_USBFS SCSI_MEDIA          *pcb_scsi_media,
    const NU_USBFS_USER SCSI     *pcb_user_scsi,
    const UINT8                  *p_cmd,
    const UINT16                  cmd_len,
    UINT8                         **pp_buf_out,
    UINT32                        *p_data_len_out)
```

Arguments

- **pcb_scsi_media**
Media to which the host sent the command.
- **pcb_user_scsi**
The media container.
- **p_cmd**
Byte buffer containing the command block.
- **cmd_len**
Length, in bytes, of the command block.
- **pp_buf_out**
Memory location pointed by this variable must contain the data location, from where the data must be transferred.
- **p_data_len_out**
Location where the number of bytes of data to be transferred must be initialized.

Return Values

- **NU_SUCCESS**
Indicates successful processing of the command.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVLD_ARG**
Indicates incorrect command block.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBF_SCSI_MEDIA_Read_10

This function processes the SCSI READ (10) command.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBF_SCSI_MEDIA_Read_10 (
    NU_USBF_SCSI_MEDIA      *pcb_scsi_media,
    const NU_USBF_USER_SCSI *pcb_user_scsi,
    const UINT8              *p_cmd,
    const UINT16             cmd_len,
    UINT8                    **pp_buf_out,
    UINT32                   *p_data_len_out)
```

Arguments

- **pcb_scsi_media**
Media to which the host sent the command.
- **pcb_user_scsi**
The media container.
- **p_cmd**
Byte buffer containing the command block.
- **cmd_len**
Length, in bytes, of the command block.
- **pp_buf_out**
Memory location pointed by this variable must contain the data location, from where the data must be transferred.
- **p_data_len_out**
Location where the number of bytes of data to be transferred must be initialized.

Return Values

- **NU_SUCCESS**
Indicates a successful read operation.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVLD_ARG**
Indicates an invalid parameter.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBF_SCSI_MEDIA_Read_12

This function processes the SCSI READ (12) command.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBF_SCSI_MEDIA_Read_12 (
    NU_USBF_SCSI_MEDIA      *pcb_scsi_media,
    const NU_USBF_USER_SCSI *pcb_user_scsi,
    const UINT8              *p_cmd,
    const UINT16             cmd_len,
    UINT8                    **pp_buf_out,
    UINT32                   *p_data_len_out)
```

Arguments

- **pcb_scsi_media**
Media to which the host sent the command.
- **pcb_user_scsi**
The media container.
- **p_cmd**
Byte buffer containing the command block.
- **cmd_len**
Length, in bytes, of the command block.
- **pp_buf_out**
Memory location pointed by this variable must contain the data location, from where the data must be transferred.
- **p_data_len_out**
Location where the number of bytes of data to be transferred must be initialized.

Return Values

- **NU_SUCCESS**
Indicates a successful read operation.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVLD_ARG**
Indicates an invalid parameter.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBFS SCSI_MEDIA_Ready

This function processes the SCSI TEST UNIT READY command.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBFS SCSI_MEDIA_Ready (NU_USBFS SCSI_MEDIA      *pcb_scsi_media,
                                   const NU_USBFS_USER SCSI *pcb_user_scsi,
                                   const UINT8                *p_cmd,
                                   const UINT16                cmd_len,
                                   UINT8                      **pp_buf_out,
                                   UINT32                     *p_data_len_out)
```

Arguments

- **pcb_scsi_media**
Media to which the host sent the command.
- **pcb_user_scsi**
The media container.
- **p_cmd**
Byte buffer containing the command block.
- **cmd_len**
Length, in bytes, of the command block.
- **pp_buf_out**
Memory location pointed by this variable must contain the data location, from where the data must be transferred.
- **p_data_len_out**
Location where the number of bytes of data to be transferred must be initialized.

Return Values

- **NU_SUCCESS**
Indicates successful processing of the command.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVLD_ARG**
Indicates incorrect command block.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBF_SCSI_MEDIA_Release_Unit

This function processes the SCSI RELEASE UNIT command.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBF_SCSI_MEDIA_Release_Unit (  
    NU_USBF_SCSI_MEDIA          *pcb_scsi_media,  
    const NU_USBF_USER_SCSI     *pcb_user_scsi,  
    const UINT8                 *p_cmd,  
    const UINT16                 cmd_len,  
    UINT8                        **pp_buf_out,  
    UINT32                       *p_data_len_out)
```

Arguments

- **pcb_scsi_media**
Media to which the host sent the command.
- **pcb_user_scsi**
The media container.
- **p_cmd**
Byte buffer containing the command block.
- **cmd_len**
Length, in bytes, of the command block.
- **pp_buf_out**
Memory location pointed by this variable must contain the data location, from where the data must be transferred.
- **p_data_len_out**
Location where the number of bytes of data to be transferred must be initialized.

Return Values

- **NU_SUCCESS**
Indicates successful processing of the command.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVLD_ARG**
Indicates incorrect command block.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBFS SCSI_MEDIA_Reserve_Unit

This function processes the SCSI RESERVE UNIT command.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBFS SCSI_MEDIA_Reserve_Unit (
    NU_USBFS SCSI_MEDIA          *pcb_scsi_media,
    const NU_USBFS_USER SCSI    *pcb_user_scsi,
    const UINT8                  *p_cmd,
    UINT16                       cmd_len,
    UINT8                        **pp_buf_out,
    UINT32                       *p_data_len_out)
```

Arguments

- **pcb_scsi_media**
Media to which the host sent the command.
- **pcb_user_scsi**
The media container.
- **p_cmd**
Byte buffer containing the command block.
- **cmd_len**
Length, in bytes, of the command block.
- **pp_buf_out**
Memory location pointed by this variable must contain the data location, from where the data must be transferred.
- **p_data_len_out**
Location where the number of bytes of data to be transferred must be initialized.

Return Values

- **NU_SUCCESS**
Indicates successful processing of the command.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVLD_ARG**
Indicates incorrect command block.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBFS SCSI_MEDIA_Reset_Device

This function resets the media and brings the media to a stable state.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBFS SCSI_MEDIA_Reset_Device (  
    NU_USBFS SCSI_MEDIA *pcb_scsi_media,  
    NU_USBFS_USER SCSI *pcb_user_scsi)
```

Arguments

- `pcb_scsi_media`
Media to which the host sent the command.
- `pcb_user_scsi`
The media container.

Return Values

- `NU_USB_NOT_SUPPORTED`
Indicates that the media does not support this event.
- `NU_USB_INVLD_ARG`
Indicates an invalid argument.
- `NU_SUCCESS`
Indicates successful completion.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBFS SCSI_MEDIA_Sense

This function processes the SCSI REQUEST SENSE command.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBFS SCSI_MEDIA_Sense (NU_USBFS SCSI_MEDIA      *pcb_scsi_media,
                                   const NU_USBFS_USER SCSI *pcb_user_scsi,
                                   const UINT8                *p_cmd,
                                   const UINT16               cmd_len,
                                   UINT8                      **pp_buf_out,
                                   UINT32                     *p_data_len_out)
```

Arguments

- **pcb_scsi_media**
Media to which the host sent the command.
- **pcb_user_scsi**
The media container.
- **p_cmd**
Byte buffer containing the command block.
- **cmd_len**
Length, in bytes, of the command block.
- **pp_buf_out**
Memory location pointed by this variable must contain the data location, from where the data must be transferred.
- **p_data_len_out**
Location where the number of bytes of data to be transferred must be initialized.

Return Values

- **NU_SUCCESS**
Indicates successful processing of the command.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVLD_ARG**
Indicates incorrect command block.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBFS SCSI_MEDIA_Snd_Diag

This function processes the SCSI SEND DIAGNOSTIC command.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBFS SCSI_MEDIA_Snd_Diag (  
    NU_USBFS SCSI_MEDIA          *pcb_scsi_media,  
    const NU_USBFS_USER SCSI     *pcb_user_scsi,  
    const UINT8                  *p_cmd,  
    const UINT16                  cmd_len,  
    UINT8                         **pp_buf_out,  
    UINT32                        *p_data_len_out)
```

Arguments

- **pcb_scsi_media**
Media to which the host sent the command.
- **pcb_user_scsi**
The media container.
- **p_cmd**
Byte buffer containing the command block.
- **cmd_len**
Length, in bytes, of the command block.
- **pp_buf_out**
Memory location pointed by this variable must contain the data location, from where the data must be transferred.
- **p_data_len_out**
Location where the number of bytes of data to be transferred must be initialized.

Return Values

- **NU_SUCCESS**
Indicates successful processing of the command.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVLD_ARG**
Indicates incorrect command block.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBF_SCSI_MEDIA_Start_Stop

This function processes the SCSI Start-Stop command.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBF_SCSI_MEDIA_Start_Stop (
    NU_USBF_SCSI_MEDIA      *pcb_scsi_media,
    const NU_USBF_USER_SCSI *pcb_user_scsi,
    const UINT8              *p_cmd,
    const UINT16             cmd_len,
    UINT8                    **pp_buf_out,
    UINT32                   *p_data_len_out)
```

Arguments

- **pcb_scsi_media**
Media to which the host sent the command.
- **pcb_user_scsi**
The media container.
- **p_cmd**
Byte buffer containing the command block.
- **cmd_len**
Length, in bytes, of the command block.
- **pp_buf_out**
Memory location pointed by this variable must contain the data location, from where the data must be transferred.
- **p_data_len_out**
Location where the number of bytes of data to be transferred must be initialized.

Return Values

- **NU_SUCCESS**
Indicates successful processing of the command.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBFS SCSI_MEDIA_Tx_Done

This function processes a new transfer request from the host.

Usage

```
STATUS NU_USBFS SCSI_MEDIA_Tx_Done (  
    NU_USBFS SCSI_MEDIA          *pcb_scsi_media,  
    const NU_USBFS_USER SCSI    *pcb_user_scsi,  
    const UINT8                  *p_cmd,  
    const UINT16                 cmd_len,  
    UINT8                        **pp_buf_out,  
    UINT32                       *p_data_len_out)
```

Arguments

- **pcb_scsi_media**
Media to which the host sent the command.
- **pcb_user_scsi**
The media container.
- **p_cmd**
Byte buffer containing the command block.
- **cmd_len**
Length, in bytes, of the command block.
- **pp_buf_out**
Memory location pointed by this variable must contain the data location, from where the data must be transferred.
- **p_data_len_out**
Location where the number of bytes of data to be transferred must be initialized.

Return Values

- **NU_SUCCESS**
Indicates successful processing of this event.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVLD_ARG**
Indicates incorrect command block.

Description

This function processes a transfer completion notification. It gets the data pointer and the length associated with the previous transfer as parameters.

It processes a new transfer request from the host. If any of the data is to be transferred to/from the Host, the corresponding data pointer and the length will be set in the parameters. If no data transfer is required then the memory location pointed to by the buf_out parameter must be filled with NULL.

Do not call this function directly unless you are extending this type.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBFS SCSI_MEDIA_Unknown_Cmd

This function processes the any of the SCSI command that is not mandatory as per the SCSI specification.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBFS SCSI_MEDIA_Unknown_Cmd (
    NU_USBFS SCSI_MEDIA          *pcb_scsi_media,
    const NU_USBFS_USER SCSI    *pcb_user_scsi,
    const UINT8                  *p_cmd,
    const UINT16                 cmd_len,
    UINT8                        **pp_buf_out,
    UINT32                       *p_data_len_out)
```

Arguments

- **pcb_scsi_media**
Media to which the host sent the command.
- **pcb_user_scsi**
The media container.
- **p_cmd**
Byte buffer containing the command block.
- **cmd_len**
Length, in bytes, of the command block.
- **pp_buf_out**
Memory location pointed by this variable must contain the data location, from where the data must be transferred.
- **p_data_len_out**
Location where the number of bytes of data to be transferred must be initialized.

Return Values

- **NU_SUCCESS**
Indicates successful processing of the command.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVLD_ARG**
Indicates incorrect command block.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBF_SCSI_MEDIA_Verify

This function processes the SCSI VERIFY command.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBF_SCSI_MEDIA_Verify (
    NU_USBF_SCSI_MEDIA          *pcb_scsi_media,
    const NU_USBF_USER_SCSI     *pcb_user_scsi,
    const UINT8                  *p_cmd,
    const UINT16                 cmd_len,
    UINT8                         **pp_buf_out,
    UINT32                       *p_data_len_out)
```

Arguments

- **pcb_scsi_media**
Media to which the host sent the command.
- **pcb_user_scsi**
The media container.
- **p_cmd**
Byte buffer containing the command block.
- **cmd_len**
Length, in bytes, of the command block.
- **pp_buf_out**
Memory location pointed by this variable must contain the data location, from where the data must be transferred.
- **p_data_len_out**
Location where the number of bytes of data to be transferred must be initialized.

Return Values

- **NU_SUCCESS**
Indicates successful processing of the command.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVLD_ARG**
Indicates incorrect command block.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBFS SCSI_MEDIA_Write

This function reads length number of bytes from the media into the buffer.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBFS SCSI_MEDIA_Write (NU_USBFS SCSI_MEDIA *pcb_scsi_media,  
                                UINT32                block,  
                                UINT8                 *p_buffer,  
                                UINT32                length)
```

Arguments

- **pcb_scsi_media**
The media to be read from.
- **block**
Block number to start from.
- **p_buffer**
Buffer to place the read data.
- **length**
Number of bytes to read.

Return Values

- **NU_SUCCESS**
Indicates a successful read operation.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVLD_ARG**
Indicates an invalid parameter.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBF_SCSI_MEDIA_Write_6

This function processes the SCSI WRITE (6) command.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBF_SCSI_MEDIA_Write_6 (
    NU_USBF_SCSI_MEDIA      *pcb_scsi_media,
    const NU_USBF_USER_SCSI *pcb_user_scsi,
    const UINT8              *p_cmd,
    const UINT16             cmd_len,
    UINT8                    **pp_buf_out,
    UINT32                   *p_data_len_out)
```

Arguments

- **pcb_scsi_media**
Media to which the host sent the command.
- **pcb_user_scsi**
The media container.
- **p_cmd**
Byte buffer containing the command block.
- **cmd_len**
Length, in bytes, of the command block.
- **pp_buf_out**
Memory location pointed by this variable must contain the data location, from where the data must be transferred.
- **p_data_len_out**
Location where the number of bytes of data to be transferred must be initialized.

Return Values

- **NU_SUCCESS**
Indicates successful processing of the command.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVLD_ARG**
Indicates incorrect command block.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBF_SCSI_MEDIA_Write_10

This function processes the SCSI WRITE (10) command.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBF_SCSI_MEDIA_Write_10 (
    NU_USBF_SCSI_MEDIA      *pcb_scsi_media,
    const NU_USBF_USER_SCSI *pcb_user_scsi,
    const UINT8              *p_cmd,
    const UINT16             cmd_len,
    UINT8                    **pp_buf_out,
    UINT32                   *p_data_len_out)
```

Arguments

- **pcb_scsi_media**
Media to which the host sent the command.
- **pcb_user_scsi**
The media container.
- **p_cmd**
Byte buffer containing the command block.
- **cmd_len**
Length, in bytes, of the command block.
- **pp_buf_out**
Memory location pointed by this variable must contain the data location, from where the data must be transferred.
- **p_data_len_out**
Location where the number of bytes of data to be transferred must be initialized.

Return Values

- **NU_SUCCESS**
Indicates successful processing of the command.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVLD_ARG**
Indicates incorrect command block.

Related Topics

[USB Function SCSI Media Functions](#)

NU_USBF_SCSI_MEDIA_Write_12

This function processes the SCSI WRITE (12) command.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS NU_USBF_SCSI_MEDIA_Write_12 (
    NU_USBF_SCSI_MEDIA      *pcb_scsi_media,
    const NU_USBF_USER_SCSI *pcb_user_scsi,
    const UINT8              *p_cmd,
    const UINT16             cmd_len,
    UINT8                    **pp_buf_out,
    UINT32                   *p_data_len_out)
```

Arguments

- **pcb_scsi_media**
Media to which the host sent the command.
- **pcb_user_scsi**
The media container.
- **p_cmd**
Byte buffer containing the command block.
- **cmd_len**
Length, in bytes, of the command block.
- **pp_buf_out**
Memory location pointed by this variable must contain the data location, from where the data must be transferred.
- **p_data_len_out**
Location where the number of bytes of data to be transferred must be initialized.

Return Values

- **NU_SUCCESS**
Indicates successful processing of the command.
- **NU_USB_NOT_SUPPORTED**
Indicates that the media does not support this event.
- **NU_USB_INVLD_ARG**
Indicates incorrect command block.

Related Topics

[USB Function SCSI Media Functions](#)

_NU_USBFS SCSI_MEDIA_Create

This function initializes the media from the SCSI perspective. The component extending NU_USBFS SCSI_MEDIA must call this function from its create routine.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS _NU_USBFS SCSI_MEDIA_Create (NU_USBFS SCSI_MEDIA *pcb_scsi_media,  
                                     CHAR *p_name,  
                                     UINT8 *p_inquiry,  
                                     UINT8 *p_capacity,  
                                     BOOLEAN is_formatted,  
                                     const VOID *p_dispatch)
```

Arguments

- **pcb_scsi_media**
Media (control block) to be initialized.
- **p_name**
Pointer to a seven-character name for the SCSI media. The name must be null-terminated.
- **p_inquiry**
Data bytes to be sent in response to an inquiry command.
- **p_capacity**
Data bytes to be sent in response to a read capacity command
- **is_formatted**
If TRUE indicates that the media has been pre formatted.
- **p_dispatch**
Dispatch table.

Return Values

- **NU_SUCCESS**
Indicates that the initialization has been successful.
- **NU_USB_NOT_PRESENT**
Indicates no more USB objects could be created due to a configuration error.
- **NU_USB_INVLD_ARG**
Indicates an invalid parameter.

Related Topics

[USB Function SCSI Media Functions](#)

_NU_USBFS SCSI_MEDIA_Delete

This function uninitializes the media. This delete function must be called from the delete routine of the component that derives from NU_USBFS SCSI_MEDIA.

Do not call this function directly unless you are extending this type.

Usage

```
STATUS _NU_USBFS SCSI_MEDIA_Delete (VOID *pcb_media)
```

Arguments

- **pcb_media**
Pointer to the control block of the SCSI media to be uninitialized.

Return Values

- **NU_SUCCESS**
Indicates that the uninitialization has been successful.

Related Topics

[USB Function SCSI Media Functions](#)

_NU_USBFS SCSI_MEDIA_Insert

This function responds to the USB host for media access calls when SCSI media is reinserted.

Usage

```
STATUS NU_USBFS SCSI_MEDIA_Insert (NU_USBFS SCSI_MEDIA *pcb_scsi_media)
```

Arguments

- `pcb_scsi_media`
Pointer to Media control block for the inserted media.

Return Values

- `NU_SUCCESS`
Indicates that the operation has been successful.

Description

Storage media such as card readers can be inserted and removed from a USB Storage device as required by the user. When SCSI media that has been removed through `NU_USBFS SCSI_MEDIA_Remove()` is reinserted, then this API must be called by the application to return an appropriate response to the USB host for media access calls.

Related Topics

[USB Function SCSI Media Functions](#)

_NU_USBFS SCSI_MEDIA_Remove

This function returns the status of removed SCSI media to the USB host.

Usage

```
STATUS _NU_USBFS SCSI_MEDIA_Remove (_NU_USBFS SCSI_MEDIA *pcb_scsi_media)
```

Arguments

- `pcb_scsi_media`
Pointer to Media control block for media to be removed.

Return Values

- `NU_SUCCESS`
Indicates that the operation has been successful.

Description

Storage media, such as card readers, can be inserted and removed from a USB Storage device on as required by the user. When SCSI media is created through a [_NU_USBFS SCSI_MEDIA_Create](#) call, then it is treated as inserted by default.

If you remove the storage media from the mass storage device, this API must be called by the application to return a media removed status to the USB host for media access calls.

Related Topics

[USB Function SCSI Media Functions](#)

Chapter 9

USB Human Interface Design (HID)

This chapter describes the Nucleus USB HID software module and the requirements for using it.

Caution



To guarantee future support and compatibility for your application, use only documented Nucleus interfaces, structures, macros, and so on.

HID Overview

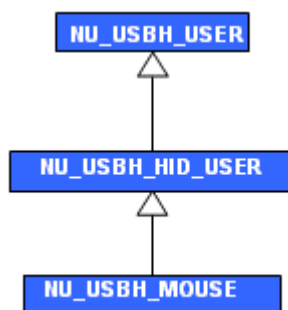
The Nucleus USB HID software module consists of the Nucleus USB Host, Nucleus USB Function, keyboard and mouse user drivers. Both mouse and keyboard are HID class devices. Each device supports a control and interrupt in endpoint. Control endpoint is used for getting information about the descriptors present in the device and for getting, setting any features supported by the device. Interrupt IN endpoint reports any data generated by the device at regular intervals. Both of these drivers are secondary drivers using services provided by the Nucleus USB and Nucleus USB HID Base Class Driver.

USB Host Mouse Driver

Nucleus USB Host HID mouse driver is an extension of NU_USBH_HID_USER component. It supports usage id X (0x30), usage id Y (0x31), usage id Wheel (0x38) of Generic Desktop (1) usage page and the Buttons usage page (0x9).

Figure 9-1 shows the component hierarchy.

Figure 9-1. USB Host Mouse User Driver Component Hierarchy



USB Host Mouse Driver and Device Manager

Whenever a new USB mouse is plugged into the target USB bus it is acquired by the USB HID mouse driver and a new device is registered with the device manager with the following label:

- USBH_MSE_LABEL

USB host HID mouse driver is a standalone component. Devices with the label USBH_MSE_LABEL are not opened by any middleware internally.

An application must open it for further use. After opening the USB mouse device the application must first obtain the IOCTL0 base with the USBH_MSE_LABEL that is USB_MSE_IOCTL_BASE.

HID_MSE_IOCTL_DATA

HID_MSE_IOCTL_DATA is a generic structure which has all the fields required by different IOCTLs of the USB HID mouse user driver. You must fill all related fields within this structure before calling any specific IOCTL. This structure has the following members:

Table 9-1. Details of the HID_MSE_IOCTL_DATA Structure

Structure Element	Description
call_back	This element is of type VOID * and used to hold the function pointer of any event handler that the application wants to register with the user driver.
context	This element is of type VOID * and is used to hold any context that will be passed through callback.
data_buffer	This element is of type VOID * and is used to hold the pointer to any data which the application wants to share with the user driver.
usages	This element is of type UINT8 and is used to hold the value of the max number of usages that can be filled by the user driver.
num_buttons	This element is of type UINT8 and is used to hold the value of the number of buttons supported by the mouse device.
wheel_present	This element is of type BOOLEAN and is used to hold binary information whether a wheel is present on the mouse device or not.

[Table 9-2](#) displays the details of other IOCTLs that are defined against USBH_MSE_LABEL and are implemented in USB Host HID Mouse user driver.

Table 9-2. Details of the USBH_MSE_Get_USAGES IOCTL

IOCTL# OFFSET	USBH_MSE_Get_USAGES
Description	Reports the usage pages supported by the mouse driver.
IOCTL Parameters	HID_MSE_IOCTL_DATA – details provided in Table 9-1 .

Table 9-2. Details of the USBH_MSE_Get_USAGES IOCTL (cont.)

IOCTL# OFFSET	USBH_MSE_Get_USAGES
Include File	<i>os/include/connectivity/ nu_connectivity.h</i>
Parameters	<p>The following parameters are elements of the HID_MSE_IOCTL_DATA structure and must be filled before calling this IOCTL:</p> <ul style="list-style-type: none"> • data_buffer – Pointer to the NU_USBH_HID_USAGE structure where the supported usages must be filled. • usages – Max number of usages that can be filled by the data buffer.

The details of USBH_MSE_REG_EVENT_HANDLER are as follows:

Table 9-3. Details of USBH_MSE_REG_EVENT_HANDLER

IOCTL# OFFSET	USBH_MSE_REG_EVENT_HANDLER
Description	Registers the function pointer that has to be invoked upon detecting a mouse event.
IOCTL Parameters	HID_MSE_IOCTL_DATA – details provided in Table 9-1 .
Include File	<i>os/include/connectivity/ nu_connectivity.h</i>
Parameters	<p>The following parameters are elements of the HID_MSE_IOCTL_DATA structure and must be filled before calling this IOCTL:</p> <ul style="list-style-type: none"> • call_back – Pointer to the event handler. The event handler function pointer must be of NU_USBH_MOUSE_Event_Handler type.

NU_USBH_MOUSE_Event_Handler

The NU_USBH_MOUSE_Event_Handler is defined as:

```
typedef STATUS (*NU_USBH_MOUSE_Event_Handler)
(NU_USBH_MOUSE* mse, VOID *device, const NU_USBH_MOUSE_EVENT* event);
```

[Table 9-4](#) summarizes the properties of event handler.

Table 9-4. Mouse Event Handler Properties

Return Type	Parameters
STATUS	<ul style="list-style-type: none"> • mse – Pointer to mouse user driver handle. • device – Pointer to the mouse device for which this event is being received. • event – Pointer to the NU_USBH_MOUSE_EVENT structure that describes the mouse event.

NU_USBH_MOUSE_EVENT

NU_USBH_MOUSE_EVENT is defined as:

```
typedef struct _nu_usbh_mouse_event
{
    UINT8 buttons;
    #define NU_USBH_LEFT_BUTTON_STATE(a) (a) & 0x1
    #define NU_USBH_RIGHT_BUTTON_STATE(a) (a) & 0x2
    #define NU_USBH_MIDDLE_BUTTON_STATE(a) (a) & 0x4
    #define NU_USBH_SIDE_BUTTON_STATE(a) (a) & 0x8

    INT8 x,y;
    INT8 wheel;
}
NU_USBH_MOUSE_EVENT;
```

The details of its elements are as follows.

Table 9-5. NU_USBH_MOUSE_EVENT Elements

Element Name	Description
buttons	Bit map that indicates the state of each of the buttons. The following macros return a non-zero value if the corresponding button is pressed and a zero if not. Call these macros by passing the buttons parameter. NU_USBH_LEFT_BUTTON_STATE(a) NU_USBH_RIGHT_BUTTON_STATE(a) NU_USBH_MIDDLE_BUTTON_STATE(a) NU_USBH_SIDE_BUTTON_STATE(a) Invoke USBH_MSE_GET_INFO IOCTL to know the number of buttons the mouse has and accordingly apply the relevant macros from the above list.
x	X-axis position relative to its previous value.
y	Y-axis position relative to its previous value.
wheel	Position of the wheel relative to its previous value. Only applicable if the wheel is present on the mouse. Invoke USBH_MSE_GET_INFO IOCTL to know about its presence.

Table 9-6. Details of USBH_MSE_REG_STATE_HANDLER

IOCTL# OFFSET	USBH_MSE_REG_STATE_HANDLER
Description	Registers the function pointers for the connection and disconnection callback routines implemented in the application.
IOCTL Parameters	HID_MSE_IOCTL_DATA – details provided in Table 9-1
Include File	<i>os/include/connectivity/nu_connectivity.h</i>
Parameters	The following parameters are elements of the HID_MSE_IOCTL_DATA structure and must be filled before calling this IOCTL: <ul style="list-style-type: none"> • context – Pointer which will be passed as context in the callbacks. • call_back – Pointer to the NU_USBH_MOUSE_CALLBACK structure containing the pointers to callback routines.

NU_USBH_MOUSE_CALLBACK

This is a structure containing function pointers of application connection and disconnection callbacks. This structure is defined as follows:

```
typedef struct _nu_usbh_mouse_callback
{
```

```

STATUS (*Connection)      (VOID* context, VOID* device);
STATUS (*Disconnection)   (VOID* context, VOID* device);
}NU_USBH_MOUSE_CALLBACK;

```

Table 9-7. Callback Functions Properties

Return Type	Name	Parameters
STATUS	Connection	<ul style="list-style-type: none"> context – pointer to any context required with this callback device – pointer to the mouse device control block which is connected to USB host
STATUS	Disconnection	<ul style="list-style-type: none"> context – pointer to any context required with this callback device – pointer to the mouse device control block which is disconnected from USB host

Table 9-8. Details of USBH_MSE_DELETE

IOCTL# OFFSET	USBH_MSE_DELETE
Description	Deletes the instance of the mouse driver and releases all the resources associated with the mouse devices managed by this driver.
Include File	<i>os/include/connectivity/ nu_connectivity.h</i>

Table 9-9. Details of USBH_MSE_GET_HANDLE

IOCTL# OFFSET	USBH_MSE_GET_HANDLE
Description	Returns the USB host HID mouse user driver's address.
IOCTL Parameters	HID_MSE_IOCTL_DATA
Include File	<i>os/include/connectivity/ nu_connectivity.h</i>
Parameters	<p>The following parameter is an element of the HID_MSE_IOCTL_DATA structure and must be filled before calling this IOCTL:</p> <ul style="list-style-type: none"> data_buffer – Double pointer which will store the address of the user driver.

Table 9-10. Details of USBH_MSE_GET_INFO

IOCTL# OFFSET	USBH_MSE_GET_INFO
Description	Returns information about a mouse-like number of buttons and whether the wheel is present or not.

Table 9-10. Details of USBH_MSE_GET_INFO (cont.)

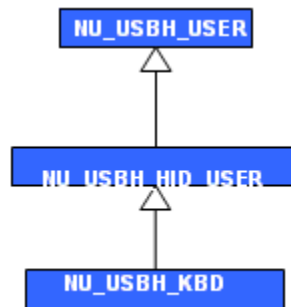
IOCTL# OFFSET	USBH_MSE_GET_INFO
IOCTL Parameters	HID_MSE_IOCTL_DATA – details provided in Table 9-1
Include File	<i>os/include/connectivity/nu_connectivity.h</i>
Parameters	<p>The following parameters are elements of the HID_MSE_IOCTL_DATA structure and must be filled before calling this IOCTL:</p> <ul style="list-style-type: none"> • num_buttons – Pointer to the location in which the number of buttons is returned. • wheel_present – Pointer to the location which displays if the wheel is present or not.

USB Host Keyboard Driver

Nucleus USB Host HID keyboard driver is an extension of NU_USBH_HID_USER component. It supports keyboard usage page (7) and all its associated usage ids and LED usage page (8) and its entire associated usage page. The LED usage page is supported by this driver to glow the NUM LOCK, CAPS LOCK, and SCROLL LOCK LEDS as per the state of their respective keys.

[Figure 9-2](#) shows the component hierarchy.

Figure 9-2. USB Host Keyboard User Driver Component Hierarchy



USB Host Keyboard Driver and Device Manager

Whenever a new USB keyboard is plugged into the target's USB bus it is acquired by the USB HID keyboard driver and a new device is registered with the device manager with the following label:

- USBH_KBD_LABEL

The USB host HID keyboard driver is a standalone component. Devices with this label are not opened by any middleware internally.

An application must open it for further use. After opening the USB keyboard device the application must first obtain the IOCTL0 base with the USBH_KBD_LABEL that is USB_KBD_IOCTL_BASE.

HID_KBD_IOCTL_DATA

HID_KBD_IOCTL_DATA is a generic structure which has all fields required by different IOCTLs of the USB HID keyboard user driver. You must fill all related fields within this structure before calling any specific IOCTL. This structure has the following members:

Table 9-11. Details of the HID_KBD_IOCTL_DATA Structure

Structure Element	Description
call_back	This element is of type VOID * and is used to hold a function pointer of any event handler that the application wants to register with the user driver.
context	This element is of type VOID * and is used to hold any context which will be passed through callback.
data_buffer	This element is of type VOID * and is used to hold the pointer to any data that the application wants to share with the user driver.
usages	This element is of type UINT8 and is used to hold the value of the max number of usages that may be filled by the user driver.

The following is the detail of other IOCTLs defined against USBH_MSE_LABEL and is implemented in the USB host HID keyboard user driver.

Table 9-12. Details of USBH USBH_KBD_Get_USAGES

IOCTL# OFFSET	USBH_KBD_Get_USAGES
Description	Reports the usage pages supported by the keyboard driver.
IOCTL Parameters	HID_KBD_IOCTL_DATA – details provided in Table 9-11
Include File	<i>os/include/connectivity/ nu_connectivity.h</i>
Parameters	The following parameters are elements of the HID_KBD_IOCTL_DATA structure and must be filled before calling this IOCTL. <ul style="list-style-type: none">• data_buffer – Pointer to the NU_USBH_HID_USAGE structure where the supported usages must be filled.• usages – Max number of usages that can be filled by the data buffer.

NU_USBH_HID_USAGE

NU_USBH_HID_USAGE is defined as:

```
typedef struct _nu_usbh_hid_usage
{
    UINT16 usage_page;
    UINT16 usage_id;
}NU_USBH_HID_USAGE;
```

Details of its element are as follows:

Table 9-13. NU_USBH_HID_USAGE Elements

Element Name	Description
usage_page	Page usage value for the given HID user driver.
usage_id	Values of usage ID.

Table 9-14. Details of USBH_KBD_REG_EVENT_HANDLER

IOCTL# OFFSET	USBH_KBD_Get_USAGES
Description	Reports the usage pages supported by the keyboard driver.
IOCTL Parameters	HID_KBD_IOCTL_DATA – details provided in Table 9-11
Include File	<i>os/include/connectivity/nu_connectivity.h</i>
Parameters	<p>The following parameters are elements of the HID_KBD_IOCTL_DATA structure and must be filled before calling this IOCTL:</p> <ul style="list-style-type: none"> call_back – Pointer to the event handler. The event handler function pointer must be of NU_USBH_KBD_Event_Handler type.

NU_USBH_KBD_Event_Handler

NU_USBH_KBD_Event_Handler is defined as:

```
typedef STATUS (*NU_USBH_KBD_Event_Handler)
(NU_USBH_KBD* kbd, VOID *device, const NU_USBH_KBD_EVENT* event);
```

Table 9-15 summarizes the properties of the event handler:

Table 9-15. Keyboard Event Handler Properties

Return Type	Parameters
STATUS	<ul style="list-style-type: none"> kbd – Pointer to keyboard user driver handle. device – Pointer to the keyboard device for which this event is being received. event – Pointer to the NU_USBH_KBD_EVENT structure that describes the keyboard event.

NU_USBH_KBD_EVENT

NU_USBH_KBD_EVENT is defined as:

```
typedef struct _nu_usbh_kbd_event
{
    /* as per bCountryCode values defined in HID spec. */
    UINT8 country_code;

    /* USB_KEY_UP or USB_KEY_DOWN. */
    UINT8 type;
    UINT8 modifier_keys; /* bit 0 - L ctrl
                           bit 1 - L Alt
                           bit 2 - L shift
                           bit 3 - L GUI
                           bit 4 - R ctrl
                           bit 5 - R Alt
                           bit 6 - R shift
                           bit 7 - R GUI */

    /* as per Table 9-12 of USB HID usage tables spec. */
    UINT8 scan_code;
}
NU_USBH_KBD_EVENT;
```

Details of its elements are as follows:

Table 9-16. NU_USBH_KBD_EVENT Elements

Element Name	Description
country_code	The bCountryCode value as defined by HID specification that identifies the hardware localized.
type	Identifies the type of keyboard event: <ul style="list-style-type: none"> USB_KEY_UP – a key has been released which is identified by the field scan_code. USB_KEY_UP – a key has been pressed which is identified by the field scan_code. USBH_KEY_MODIFIER – modifier keys have been pressed whose states are identified by the bitmap field: modifier_keys. The field scan_code contains an undefined value for this event type.

Table 9-16. NU_USBH_KBD_EVENT Elements (cont.)

Element Name	Description
scan_code	Identifies the key pressed or released. The codes are as per Table 9-12 of USB HID usage tables spec.

Table 9-17. Details of Modifier Keys

Bit #	Key
0	Left Ctrl key
1	Left Alt key
2	Left Shift key
3	Left GUI key
4	Right Ctrl key
5	Right Alt key
6	Right Shift key
7	Right GUI key

Table 9-18. Details of USBH_KBD_REG_STATE_HANDLER

IOCTL# OFFSET	USBH_KBD_REG_STATE_HANDLER
Description	Registers the function pointers for the connection and disconnection callback routines implemented in the application.
IOCTL Parameters	HID_KBD_IOCTL_DATA – details provided in Table 9-11
Include File	<i>os/include/connectivity/ nu_connectivity.h</i>
Parameters	<p>The following parameters are elements of the HID_KBD_IOCTL_DATA structure and must be filled before calling this IOCTL:</p> <ul style="list-style-type: none"> • context – Pointer which will be passed as context in the callbacks. • call_back – Pointer to NU_USBH_KBD_CALLBACK structure containing the pointers to callback routines.

NU_USBH_KBD_CALLBACK

This is a structure containing function pointers of application connection and disconnection callbacks. This structure is defined as follows:

```
typedef struct _nu_usbh_kbd_callback
{
    STATUS (*Connection)      (VOID* context, VOID* device);
    STATUS (*Disconnection) (VOID* context, VOID* device);
}NU_USBH_KBD_CALLBACK;
```

[Table 9-19](#) summarizes the properties of these callback functions.

Table 9-19. Callback Function Properties

Return Type	Name	Parameters
STATUS	Connection	<ul style="list-style-type: none"> context – pointer to any context required with this callback device – pointer to the keyboard device control block which is connected to USB host
STATUS	Disconnection	<ul style="list-style-type: none"> context – pointer to any context required with this callback device – pointer to the keyboard device control block which is disconnected from USB host

Table 9-20. Details of USBH_KBD_DELETE

IOCTL# OFFSET	USBH_KBD_DELETE
Description	Deletes the instance of the keyboard driver and releases all the resources associated with the keyboard devices managed by this driver.
Include File	<i>os/include/connectivity/ nu_connectivity.h</i>

Table 9-21. Details of USBH_KBD_GET_HANDLE

IOCTL# OFFSET	USBH_KBD_GET_HANDLE
Description	Returns the USB host HID keyboard user driver's address.
IOCTL Parameters	HID_KBD_IOCTL_DATA – details provided in Table 9-11
Include File	<i>os/include/connectivity/ nu_connectivity.h</i>
Parameters	The following parameters are elements of the HID_KBD_IOCTL_DATA structure and must be filled before calling this IOCTL.
data_buffer	Double pointer which will store the address of the user driver.

USB Function Mouse Driver

This user driver provides mouse functionality to the HID class driver. Format of data for mouse devices exchanged between the device and host is defined in the report descriptor outlined in the

class specification for HID devices. This user driver is based on the function user driver template provided with the Nucleus USB Function Stack.

Function Mouse Driver Configuration Macros

The following are the configuration macros for the Nucleus USB Function User Driver. These macros can be found in the header file *nu_usbhf_mse_imp.h*. The default values set for these macros are suitable for most of the applications. However, they can be changed to suit the application needs as required.

Table 9-22. Configuration Macros

Macro	Suitable Range of Values	Description
MSE_REPORT_LEN	3	Maximum length of buffer containing the report data
MUS_REPORT_DESCRIPTOR_SIZE	50	Report descriptor size

Function Mouse Data Structures

The [NU_USBHF_MSE_REPORT](#) function defines the data structure for data exchanged between the device and the host. Data must follow this format, which is defined by the report descriptor for the device.

Values of Characters and LED States in the Application

A callback function registered with the USB Host Keyboard User Driver can be used to address the events generated by keyboard. This callback event handler is registered with the Keyboard class driver block and is invoked when the keyboard generates an event. The parameter event received in the callback function holds information about the generated event and the parameter device tells which of the attached keyboard devices generated the event.

The event->type can be used to check what event was generated (USB_KEY_UP, USB_KEY_DOWN, or USB_KEY_MODIFIER). In case of USB_KEY_UP or USB_KEY_DOWN events, the event->scan_code tells which key generated the event. For USB_KEY_MODIFIER event, the bitmap event->modifier_keys holds information about the modifier keys generating the event. The array Keys defined in the application maintains information about keys indexed with their scan codes. Hence, Keys[event->scan_code] will hold the key whose scan code is event->scan_code.

For example, when key “A” is pressed, the event->type will be USBH_KEY_DOWN, event->scan_code will be 0x4 and Keys[event->scan_code] will hold character “A”.

The device parameter of event handler is of type NU_USBH_KBD_DEVICE * and points to the keyboard device that generated the event. The bitmap device->led_keys holds the state of

LED keys on the keyboard. The USBH_KEY_NUM_LOCK, USBH_KEY_CAPS_LOCK, and USBH_KEY_SCROLL_LOCK are used with device->led_keys to check the state of respective LED on keyboard.

The keyboard event handler callback function is declared as follows:

```
/* Function prototype of keyboard event handler callback */
STATUS Keyboard_Event_Handler (NU_USBH_KBD* cb, VOID *device,
                               const NU_USBH_KBD_EVENT *event);
```

The event handler is registered with the keyboard driver control block as follows:

```
NU_USBH_KBD kbd;
.....
/* Create the keyboard driver control block kbd */
.....
STATUS status;
.....
.....
/* Register the callback function with keyboard driver control block */
status = NU_USBH_KBD_Reg_Event_Handler (&kbd, Keyboard_Event_Handler);
```

Define the keyboard event handler to retrieve the events:

```
STATUS Keyboard_Event_Handler (NU_USBH_KBD * cb, VOID *device,
                               const NU_USBH_KBD_EVENT *event)
{
    /* The parameter device points to the keyboard device that generated
       the event, and is passed as NU_USBH_KBD_DEVICE *device. */

    /* Check if the event is raised due to release of a pressed key */
    if (event->type == USBH_KEY_UP)
    {
        /* Check which key was released */
        Printf(Keys[event->scan_code]);
        Printf(" Key Up\r\n");
    }

    /* Check if the event is raised due to pressing of a key */
    if (event->type == USBH_KEY_DOWN)
    {
        /* Check which key was pressed */
        Printf(Keys[event->scan_code]);
        Printf(" Key Down\r\n");
    }

    /* Check if the event is raised due to a modifier key */
    if (event->type == USBH_KEY_MODIFIER)
    {
        /* Check which modifier key was pressed */
        if (event->modifier_keys & 0)
        {
            Printf("Left CTRL Key\r\n");
        }
        if (event->modifier_keys & 6)
        {

```



```

        Printf("Right Shift Key\r\n");
    }

    ...
    ...
    ...
}

/* Check the state of NUM Lock LED */
if (device->led_keys & USBH_KEY_NUM_LOCK)
{
    Printf("NUM LOCK is ON\r\n");
}
else
{
    Printf("NUM LOCK is OFF\r\n");
}
...
...
...
return NU_SUCCESS;
}

```

Function Mouse Driver and Device Manager

USB Function HID Mouse is a standalone driver that is not opened by any middleware internally during initialization time. Whenever an application opens it, it creates the HID mouse descriptors in the function stack and becomes ready for a USB connection with an USB host. It is registered with the device manager with the label:

- USBF_MOUSE_LABEL

After opening the USB mouse device the application must first obtain the IOCTL0 base with the USBF_MOUSE_LABEL that is USB_MSE_IOCTL_BASE.

The following are the details of other IOCTLs defined against USBF_MOUSE_LABEL and implemented in the USB function HID mouse user driver.

Table 9-23. Details of NU_USBF_MSE_IOCTL_WAIT

IOCTL# OFFSET	NU_USBF_MSE_IOCTL_WAIT
Description	Provides services for user level threads to wait for a particular device to be connected. The thread goes into a state specified by the suspension option if the device is not yet connected. This IOCTL returns a device handle as output when the call is successful, which can be used by applications to use other services of USER layer.
IOCTL Parameters	USBF_MSE_WAIT_DATA
Include File	<i>os/include/connectivity/ nu_connectivity.h</i>

Table 9-23. Details of NU_USB_MSE_IOCTL_WAIT (cont.)

IOCTL# OFFSET	NU_USB_MSE_IOCTL_WAIT
Parameters	The following parameters are elements of the USBF_MSE_WAIT_DATA structure and must be filled before calling this IOCTL: <ul style="list-style-type: none">• suspend – Suspension option.• handle_out – Pointer to a memory location to hold the pointer to the device handle.

USBF_MSE_WAIT_DATA

USBF_MSE_WAIT_DATA is defined as:

```
typedef struct _usb_mse_wait_data_  
{  
    UNSIGNED suspend;  
    VOID **handle_out;  
} USBF_MSE_WAIT_DATA;
```

Details of its elements are as follows:

Table 9-24. Details of USBF_MSE_WAIT_DATA Elements

Element Name	Description
suspend	Task suspension option.
handle_out	It is an output parameter and will return the handle of the function HID class driver control block.

Table 9-25. Details of NU_USB_MSE_IOCTL_SEND_LFT_BTN_CLICK

IOCTL# OFFSET	NU_USB_MSE_IOCTL_SEND_LFT_BTN_CLICK
Description	This IOCTL is called to tell the host that the application has clicked the left mouse button. It fills up the report data and passes it to the host on the interrupt in the pipe.
Include File	<i>os/include/connectivity/nu_connectivity.h</i>

Table 9-26. Details of NU_USBF_MSE_IOCTL_SEND_RHT_BTN_CLICK

IOCTL# OFFSET	NU_USBF_MSE_IOCTL_SEND_RHT_BTN_CLICK
Description	This IOCTL is called to tell the host that the application has clicked the right mouse button. It fills up the report data and passes it to the host on the interrupt in the pipe.
Include File	<i>os/include/connectivity/nu_connectivity.h</i>

Table 9-27. Details of NU_USBF_MSE_IOCTL_SEND_MDL_BTN_CLICK

IOCTL# OFFSET	NU_USBF_MSE_IOCTL_SEND_MDL_BTN_CLICK
Description	This IOCTL is called to tell the host that the application has clicked the middle mouse button. It fills up the report data and passes it to the host on the interrupt in the pipe.
Include File	<i>os/include/connectivity/nu_connectivity.h</i>

Table 9-28. Details of NU_USBF_MSE_IOCTL_MOVE_POINTER

IOCTL# OFFSET	NU_USBF_MSE_IOCTL_MOVE_POINTER
Description	This IOCTL is called to tell the host that the application has moved the mouse wheel. It fills up the report data and passes it to the host on the interrupt in the pipe.
IOCTL Parameters	NU_USBF_MSE_REPORT
Include File	<i>os/include/connectivity/nu_connectivity.h</i>
Parameters	<p>The following parameters are elements of the NU_USBF_MSE_REPORT structure and must be filled before calling this IOCTL.</p> <ul style="list-style-type: none"> • MouseX – Movement in X direction. • MouseY – Movement in Y direction.

NU_USBF_MSE_REPORT

NU_USBF_MSE_REPORT is defined as:

```
typedef struct _nu_usbfs_mse_report
{
    UINT8 Button;
    UINT8 MouseX;
    UINT8 MouseY;
}
```

```
    /* To properly align this structure on 32-bit boundary */  
    UINT8 pad[1];  
}  
NU_USBFS_MSE_REPORT;
```

The details of its elements are as follows:

Table 9-29. NU_USBFS_MSE_REPORT Element Details

Element Name	Description
Button	The active button of the mouse. Unused for this IOCTL.
MouseX	Movement in X direction.
MouseY	Movement in Y direction.

Table 9-30. Details of NU_USBFS_MSE_IOCTL_GET_HID_CB

IOCTL# OFFSET	NU_USBFS_MSE_IOCTL_GET_HID_CB
Description	This IOCTL retrieves the function HID class driver's address.
IOCTL Parameters	VOID **
Include File	<i>os/include/connectivity/ nu_connectivity.h</i>
Parameters	Double pointer used to retrieve the class driver's address.

Table 9-31. Details of NU_USBFS_MSE_IOCTL_IS_HID_DEV_CONNECTED

IOCTL# OFFSET	NU_USBFS_MSE_IOCTL_IS_HID_DEV_CONNECTED
Description	This IOCTL tells the caller whether the function HID device is connected to the host or not.
IOCTL Parameters	BOOLEAN
Include File	<i>os/include/connectivity/ nu_connectivity.h</i>
Parameters	Boolean variable to hold the status of the function HID device.

USB Function Keyboard Driver

This user driver provides keyboard functionality to the HID class driver. Format of data for keyboard devices exchanged between the device and host is defined in the report descriptor outlined in the class specification for HID devices. This user driver is based on the function user driver template provided with the Nucleus USB Function Stack.

Function Keyboard Configuration Macros

The following are the configuration macros for the Nucleus USB Function User Driver. These macros can be found in the header file *nu_usbf_kb_imp.h*. The default values set for these macros are suitable for most of the applications. However, they can be changed to suit the application needs as required.

Table 9-32. Nucleus USB Function User Driver Configuration Macros

Macro	Suitable Range of Values	Description
USBF_KB_REPORT_LEN	8	Maximum length of buffer in bytes containing the report data
USBF_KB_MAX_KEYS	6	Maximum number that can be pressed at a time
USBF_KB_BUFFER_SIZE	20	Buffer size for the key pressed data
USBF_KB_ASCII_TABLE_SIZE	100	Number of ASCII keys
USBF_KB_REPORT_DESCRIPTOR_LEN	65	Report descriptor size

Mouse Driver Data Structures

The following are the important data structures defined in the mouse user driver.

NU_USBF_KB_IN_REPORT

This function defines the data structure for the data to be exchanged from device to host. Data must follow this format, which is defined by the report descriptor for the device.

Table 9-33. NU_USBF_KB_IN_REPORT

Field	Description
modifierbyte	Field that represents the modifier key data
reserved	Reserved byte passed as zero

Table 9-33. NU_USBF_KB_IN_REPORT (cont.)

Field	Description
keyCode	Usage code for the keys pressed are stored in this array

NU_USBF_KB_OUT_REPORT

This function defines the data structure for the data to be exchanged from the host to the device. Data must follow this format, which is defined by the report descriptor for the device.

Table 9-34. NU_USBF_KB_OUT_REPORT

Field	Description
ledstates	Field that represents the keyboard LED states.

Function Keyboard Driver and Device Manager

USB Function HID Keyboard is a standalone driver and it is not opened by any middleware internally during initialization time. Whenever an application opens it, USB Function HID Keyboard creates the HID keyboard descriptors in the function stack and becomes ready for a USB connection with USB host. It is registered with the device manager with the label:

- USBF_KEYBOARD_LABEL

After opening the USB keyboard device the application must first obtain the IOCTL0 base with the USBF_KEYBOARD_LABEL that is USB_KBD_IOCTL_BASE.

[Table 9-35](#) describes the details of other IOCTLs defined against USBF_KEYBOARD_LABEL and implemented in the USB function HID keyboard user driver.

Table 9-35. Details of NU_USBF_KBD_IOCTL_WAIT

IOCTL# OFFSET	NU_USBF_KBD_IOCTL_WAIT
Description	Provides services for user level threads to wait for a particular device to be connected. The thread goes into a state specified by the suspension option if the device is not yet connected. This IOCTL returns a device handle as output when the call is successful, which can be used by applications to use other services of the USER layer.
IOCTL Parameters	USBF_KBD_WAIT_DATA
Include File	<i>os/include/connectivity/nu_connectivity.h</i>

Table 9-35. Details of NU_USBKBD_IOCTL_WAIT (cont.)

IOCTL# OFFSET	NU_USBKBD_IOCTL_WAIT
Parameters	<p>The following parameters are elements of the USBKBD_WAIT_DATA structure and must be filled before calling this IOCTL.</p> <ul style="list-style-type: none"> • suspend – Suspension option. • handle_out – Pointer to the memory location to hold the pointer to the device handle.

USBKBD_WAIT_DATA

USBKBD_WAIT_DATA is defined as:

```
typedef struct _usbkbd_wait_data_
{
    UNSIGNED suspend;
    VOID **handle_out;
} USBKBD_WAIT_DATA;
```

Details of its elements are as follows:

Table 9-36. USBKBD_WAIT_DATA Element Details

Element Name	Description
suspend	Task suspension option.
handle_out	It is an output parameter and will return the handle of the function HID class driver control block.

Table 9-37. Details of NU_USBKBD_IOCTL_REG_CALLBACK

IOCTL# OFFSET	NU_USBKBD_IOCTL_REG_CALLBACK
Description	This IOCTL is used to register the receive callback with the keyboard user driver.
IOCTL Parameters	KEYBOARD_RX_CALLBACK
Include File	<i>os/include/connectivity/ nu_connectivity.h</i>
Parameters	Function pointer of type KEYBOARD_RX_CALLBACK

KEYBOARD_RX_CALLBACK

KEYBOARD_RX_CALLBACK is defined as:

```
typedef STATUS (*KEYBOARD_RX_CALLBACK) (UINT32 size,UINT8 *data_buf);
```

Table 9-38 summarizes the properties of this callback function.

Table 9-38. KEYBOARD_RX_CALLBACK Properties

Return Type	Parameters
STATUS	<ul style="list-style-type: none"> size – Size of received data. data_buf – Pointer to data buffer to hold the input data from host side.

Table 9-39. Details of NU_USBKBD_IOCTL_SEND_KEY_EVENT

IOCTL# OFFSET	NU_USBKBD_IOCTL_SEND_KEY_EVENT
Description	This IOCTL is used to pass data from the application to the class driver. The user driver can send a maximum of six keys in the key buffer, otherwise the keyboard reports a phantom state indexing usage (ErrorRollOver). The class driver sends the keyboard report data on the interrupt IN pipe.
IOCTL Parameters	USBKBD_SEND_KEY_DATA
Include File	<i>os/include/connectivity/nu_connectivity.h</i>
Parameters	<p>The following parameters are elements of the USBKBD_SEND_KEY_DATA structure and must be filled before calling this IOCTL.</p> <ul style="list-style-type: none"> key – Pointer to the array of character pointers containing the Keys pressed such as ‘a’, ‘f’, and so on. buffer_size – Number of keys pressed. modifierkeybuffer – Pointer to the array of character pointers containing the Modifier keys pressed such as LEFT CTRL, LEFT ALT, and so on. modifierkeybuffer_size – Number of modifier keys pressed.

USBKBD_SEND_KEY_DATA

USBKBD_SEND_KEY_DATA is defined as:

```
typedef struct _usbkkbd_send_key_data_
{
    CHAR **key;
    UINT8 buffer_size;
    UINT8 *modifierkeybuffer;
    UINT8 modifierkeybuffer_size;
} USBKBD_SEND_KEY_DATA;
```


The details of its elements are as follows:

Table 9-40. USBF_KBD_SEND_KEY_DATA Elements

Element Name	Description
key	Pointer to the array of character pointers containing the Keys pressed such as 'a', 'f' etc.
buffer size	Number of keys pressed.
modifierkeybuffer	Pointer to the array of character pointers containing the Modifier keys pressed such as LEFT CTRL, LEFT ALT, and so on.
modifierkeybuffer size	Number of modifier keys pressed.

Table 9-41. Details of NU_USBF_KBD_IOCTL_GET_HID_CB

IOCTL# OFFSET	NU_USBF_KBD_IOCTL_GET_HID_CB
Description	This IOCTL retrieves the function HID class driver's address.
IOCTL Parameters	VOID **
Include File	<i>os/include/connectivity/ nu_connectivity.h</i>
Parameters	Double pointer used to retrieve the class driver's address.

Table 9-42. Details of NU_USBF_KBD_IOCTL_IS_HID_DEV_CONNECTED

IOCTL# OFFSET	NU_USBF_KBD_IOCTL_IS_HID_DEV_CONNECTED
Description	This IOCTL tells the caller whether the function HID device is connected to the host or not.
IOCTL Parameters	BOOLEAN
Include File	<i>os/include/connectivity/ nu_connectivity.h</i>
Parameters	Boolean variable to hold the status of the function HID device

Chapter 10

USB Audio Driver

Nucleus ReadyStart contains a host-side USB audio driver. This chapter describes support for the USB host audio driver in Nucleus ReadyStart in detail.

Caution



To guarantee future support and compatibility for your application, use only documented Nucleus interfaces, structures, macros, and so on.

USB Host Audio Driver Overview

Audio devices are classified by USB specifications as audio class devices. The audio class definitions are defined by the Universal Serial Bus Class Definitions for Audio Devices. This specification also defines the command sets and bus usage adopted for communicating with USB audio devices. The USB Implementers Forum, Inc. (USB-IF) specification for Audio Devices can be downloaded at the following location:

http://www.usb.org/developers/devclass_docs/audio10.pdf

Audio functions are addressed through their audio interfaces. Each audio function has a single AudioControl (AC) interface and can have several AudioStreaming and MIDIStreaming interfaces. The AC interface is used to access the audio Controls of the function. The AudioStreaming (AS) interfaces are used to transport audio streams into and out of the function. The MIDIStreaming (MS) interfaces are used to transport MIDI data streams into and out of the audio function. MS interfaces are not supported by the Nucleus USB Host Audio driver. The collection of the single AC interface, and the AS and MS interfaces that belong to the same audio function, is called the Audio Interface Collection (AIC). A device can have multiple AICs active at the same time. These Collections are used to control multiple independent audio functions located in the same composite device

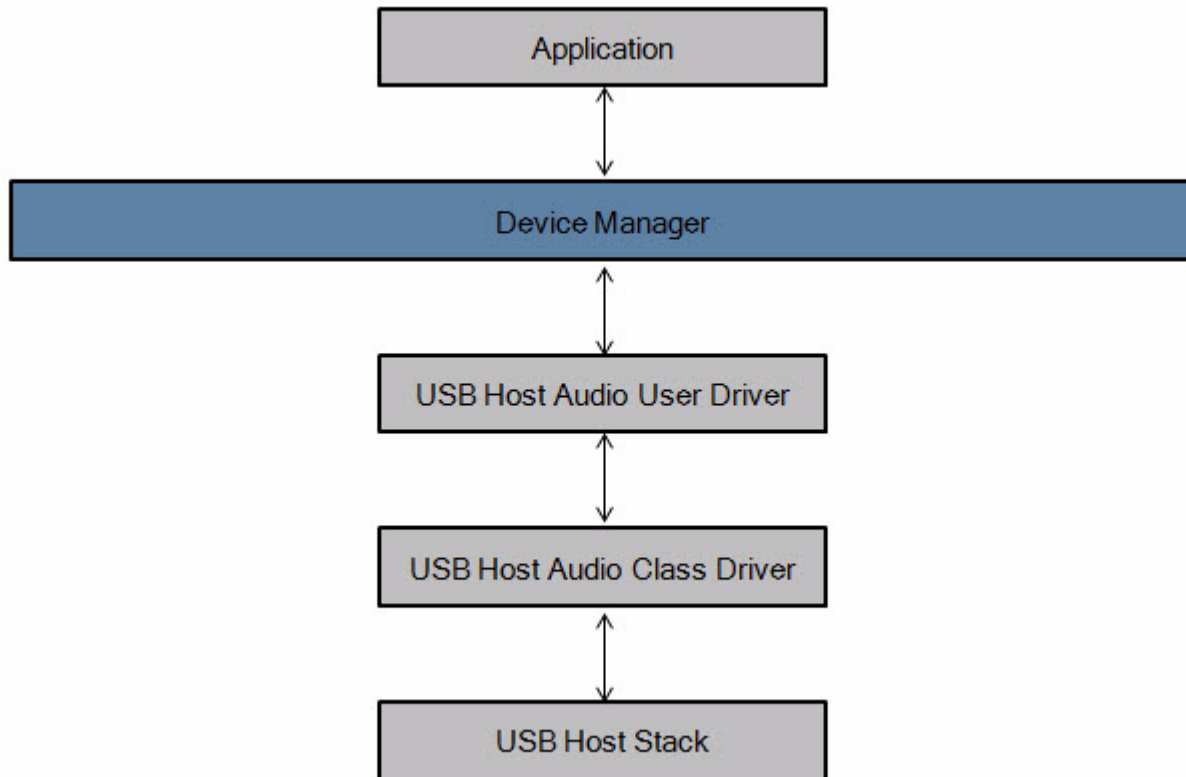
According to the Universal Serial Bus Class Definitions for Audio Devices, the audio class code is defined in the interface descriptor. Consequently, the Nucleus USB Host Audio Driver is an interface driver and owns an interface on a device. The SubClass and Protocol codes are defined in the bInterfaceSubclass and bInterfaceProtocol fields of the interface descriptor for the device. All audio functions are part of a certain Subclass. The following three Subclasses are currently defined in the Universal Serial Bus Class Definitions for Audio Devices:

- AudioControl Interface Subclass
- AudioStreaming Interface Subclass

- MIDISTreaming Interface Subclass

The USB host audio class driver is registered with the USB host stack and communicates with the USB audio device using USB Audio Data Formats. An application interacts with the USB host audio driver through the standard Device Manager interface (Open, Close, Read, Write and Ioctl). [Figure 10-1](#) provides a block diagram of the USB Host Audio driver's layered architecture.

Figure 10-1. USB Host Audio Driver Block Diagram



Nucleus USB Host Audio Driver Operation

This section provides details on various aspects of operation for the USB host audio class driver. USB device enumeration is handled at the USB host stack level; therefore, is considered out of the scope of this document.

Configuration

In order to use the USB host audio driver, it should be enabled in the build configuration of Nucleus ReadyStart. The following keys should be defined to enable USB host audio class and user driver.

```
nu.os.conn.usb.host.audio.class.enable = true
nu.os.conn.usb.host.audio.user.enable = true
```

Please refer to the Nucleus ReadyStart Guide for further information about build configurations.

Note



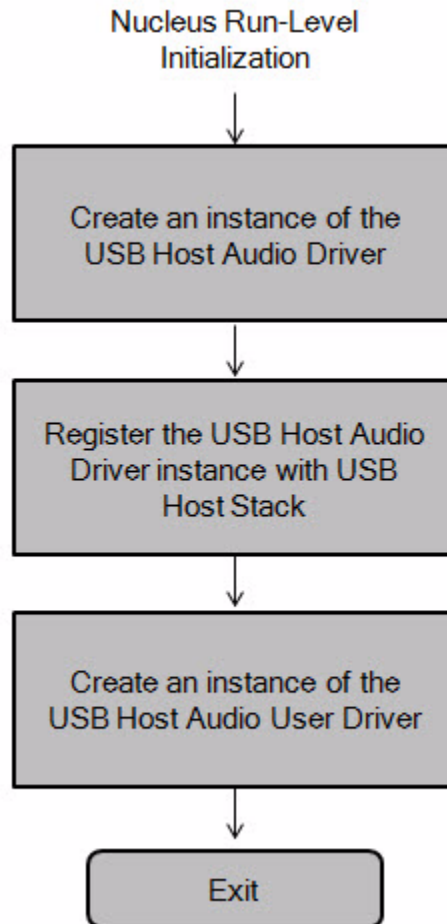
In order to use the Nucleus USB host audio driver, the USB host and common stack components should also be enabled.

Initialization

The Nucleus USB Host Audio Driver initialization is completed at the requested run level during Nucleus ReadyStart initialization. During the initialization process, an instance of the USB host audio class driver is created and registered with the USB host stack. After successful registration, the USB host audio user driver is created and initialized.

Figure 10-2 describes the initialization sequence for the Nucleus USB Host Audio Driver.

Figure 10-2. Nucleus USB Host Audio Driver Initialization Sequence



Device Connection and Disconnection Handling

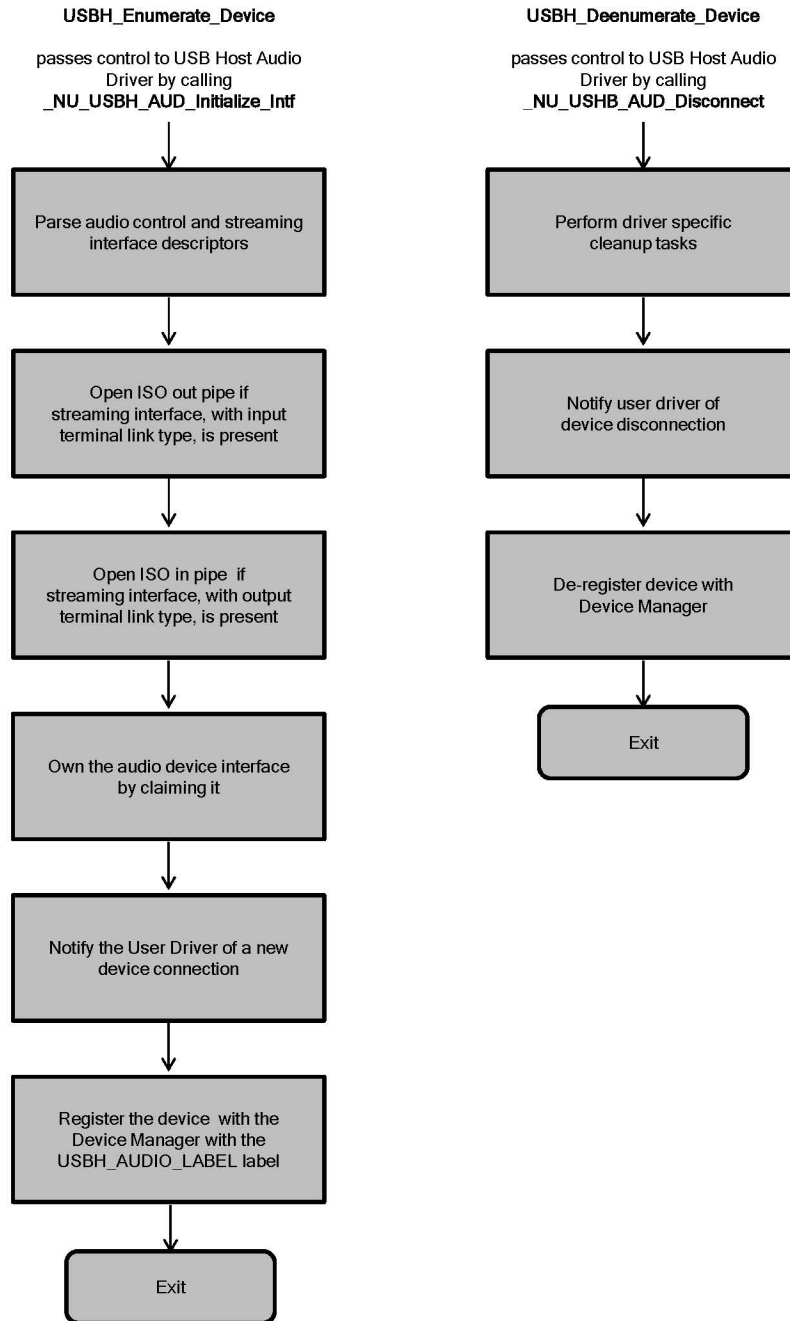
Initially the USB host stack controls the device after connection. After analyzing the configuration descriptor for the device, it searches for the best driver (match specifications) for each interface of the connected device. The USB host audio class driver is matched on the basis of class code (defined as 0x01 by USB-IF). After matching the driver, the USB host stack passes control of the USB host audio interface to the audio class driver.

The USB host audio class driver performs a number of management tasks before declaring the ownership of respective interface. At this point, a new device with an `USBH_AUDIO_LABEL` label is registered with the Device Manager so that an application interested in a USB audio device may open and communicate with it.

Device disconnection processing involves local cleanup and sending disconnection notification to the user driver. Upon disconnection, the device is de-registered from the Device Manager so that further attempts of I/O operation on this device are declined.

[Figure 10-3](#) describes detailed flow for USB audio device connection and disconnection processing.

Figure 10-3. Nucleus USB Audio Device Connect/Disconnect Process Flow



Nucleus USB Host Audio Driver Application Interface

An application interacts with the Nucleus USB Host Audio Driver using a standard Device Manager interface. The application waits for a device with USBH_AUDIO_LABEL to be registered with the Device Manager. Once device is available, it opens it and communicates with it using Device Manager's read, write and IOCTL functions.

Device Open

An application looks for a device having the label USBH_AUDIO_LABEL registered with the Device Manager. Once a device is available, it opens it by calling DVC_Dev_Open or DVC_Dev_ID_Open.

Read/Write Operation

An application performs read and write operation on a device by calling the DVC_Dev_Read and DVC_Dev_Write functions.

IOCTL Execution

In order to execute an IOCTL on a device, an application calls the DVC_Dev_Ioctl function. Additional information for IOCTL execution is provided with [Nucleus USB Host Audio Driver IOCTLs](#).

Device Close

After completing I/O operations, a device is closed by calling DVC_Dev_Close function.

Nucleus USB Host Audio Driver IOCTL Data Structures

This section covers data structures used by the [Nucleus USB Host Audio Driver IOCTLs](#).

NU_AUDIO_DEVICE_SETTINGS

NU_AUDIO_DEVICE_SETTINGS is a data structure of type _nu_audio_device_settings. NU_AUDIO_DEVICE_SETTINGS contains information about different settings of the audio device, such as sampling rate, channels, and supported frequencies.

```
typedef struct _nu_audio_device_settings
{
    NU_USBH_AUD_DEV    *aud_dev;
    UINT32             sampling_rate;
    UINT16             lock_delay;
    UINT8              channels;
    UINT8              sample_size;
    UINT8              reserved_bits;
    BOOLEAN            double_buffering;
    UINT8              function;
    UINT32             index;
    UINT32             func_count;
```



```

    UINT32      freq_count;
    BOOLEAN     freq_type;
    UINT32      *freq_list;
} NU_AUDIO_DEVICE_SETTINGS;

```

The members of the NU_AUDIO_DEVICE_SETTINGS data structure are defined in [Table 10-1](#).

Table 10-1. NU_AUDIO_DEVICE_SETTINGS

Member	Description
aud_dev	Pointer to the NU_USBH_AUD_DEV audio device control block.
sampling_rate	The number of samples of audio carried per second.
lock_delay	The time it takes the audio device to reliably lock its internal clock recovery circuitry to reliably produce or consume the audio data stream.
channels	Channel type(MONO or STEREO) in an Audio Clip. For Mono, it is 1. For stereo, it is 2.
sample_size	Sample size of the audio.
reserved_bits	Reserved bits for future use.
double_buffering	Flag showing whether single buffering or double buffering is used for recording.
function	Microphone or Speaker function.
index	Zero based index of the audio function.
func_count	The number of functions available in the device.
freq_count	The count of supported frequencies.
freq_type	The type of supported sampling frequency (1 for Discrete Frequency, 0 for Continuous Frequency).
freq_list	Pointer to the list of supported frequencies.

Related Topics

[USBH_AUDIO_OPEN_PLAY_SESSION](#) [USBH_AUDIO_CLOSE_PLAY_SESSION](#)
[USBH_AUDIO_OPEN_RECORD_SESSION](#) [USBH_AUDIO_CLOSE_RECORD_SESSION](#)
[USBH_AUDIO_GET_FUNC_COUNT](#) [USBH_AUDIO_GET_FREQ_COUNT](#)
[USBH_AUDIO_GET_FUNC_SETTINGS](#)

NU_USBH_AUD_Data_Callback

The NU_USBH_AUD_Data_Callback data structure is used to notify an application of completion of a playing or recording session.

```
typedef VOID (*NU_USBH_AUD_Data_Callback) (  
    VOID    *device,  
    VOID    *buffer,  
    UINT32  status,  
    UINT32  transfer_length);
```

The members of the NU_USBH_AUD_Data_Callback data structure are defined in [Table 10-2](#).

Table 10-2. NU_USBH_AUD_Data_Callback

Member	Description
device	Pointer to the audio device control block.
buffer	Pointer to application buffer to hold or send data.
status	Status of completion for the playing or recording of sound.
transfer_length	The length of the application buffer in terms of audio samples.

Related Topics

[USBH_AUDIO_REGISTER_RECORD_CALLBACK](#)
[USBH_AUDIO_REGISTER_PLAY_CALLBACK](#)

NU_AUDIO_OP_PARAM

NU_AUDIO_OP_PARAM is a data structure of type _nu_audio_op_param. NU_AUDIO_OP_PARAM contains information about the operational parameters of an audio device, such as the pointer to the start of an audio clip, length of the audio clip and sampling rate of the audio device.

```
typedef struct _nu_audio_op_param  
{  
    NU_USBH_AUD_DEV    *aud_dev;
```

```

VOID          *buffer;
VOID          *loopback_buffer;
UINT32        samples;
UINT32        max_samples;
UINT32        sampling_rate;
UINT32        event;
UINT8         sample_size;
UINT8         channels;
UINT8         handle;
DATA_ELEMENT  padding[1];
} NU_AUDIO_OP_PARAM;

```

The members of the NU_AUDIO_OP_PARAM data structure are defined in [Table 10-3](#).

Table 10-3. NU_AUDIO_OP_PARAM

Member	Description
aud_dev	Pointer to NU_USBH_AUD_DEV audio device control block.
buffer	Pointer to the start of audio clip to be played or recorded.
loopback_buffer	Pointer to the buffer used to hold or send data in loopback mode.
samples	Length of an audio clip in terms of audio samples.
max_samples	Maximum length of an audio clip in terms of audio samples.
sampling_rate	Sampling rate of the audio device.
event	Used by the application to contain notification codes, such as the notification used for playback completion.
sample_size	Sample size, in bytes.
channels	Channel type (MONO or STEREO) in an Audio Clip (1 for Mono, 2 for Stereo).
handle	Zero based index of audio adapters connected with the target platform.
padding	Padding for 4-byte alignment.

Related Topics

[USBH_AUDIO_PLAY_SOUND](#)

[USBH_AUDIO_RECORD_SOUND](#)

NU_USBH_AUDIO_NOTIFY_CALLBACKS

NU_USBH_AUDIO_NOTIFY_CALLBACKS is a data structure of type _nu_usbh_audio_notify_callbacks. NU_USBH_AUDIO_NOTIFY_CALLBACKS is used to provide an application with the notification callbacks for connection and disconnection of an audio device.

```
typedef struct _nu_usbh_audio_notify_callbacks
{
    NU_AUDH_USER_DISCON disconnect_callback;
    NU_AUDH_USER_CONN   connect_callback;

}NU_USBH_AUDIO_NOTIFY_CALLBACKS;
```

The members of the NU_USBH_AUDIO_NOTIFY_CALLBACKS data structure are defined in [Table 10-4](#).

Table 10-4. NU_USBH_AUDIO_NOTIFY_CALLBACKS

Member	Description
disconnect_callback	The callback invoked when an audio device is disconnected.
connect_callback	The callback invoked when an audio device is connected.

Related Topics

[USBH_AUDIO_REGISTER_NOTIFICATION_CALLBACKS](#)

NU_AUDIO_USER_DEVICE_SETTINGS

NU_AUDIO_USER_DEVICE_SETTINGS is a data structure of type `_nu_audio_user_device_settings`. NU_AUDIO_USER_DEVICE_SETTINGS contains information about settings of an audio user device, such as volume information and supported functions in the audio user device.

```
typedef struct _nu_audio_user_device_settings
{
    NU_USBH_AUD_USER_DEV      *aud_user_dev;
    UINT8                     function;
    UINT8                     ch_type;
    BOOLEAN                   mute;
    NU_USBH_AUD_USR_FEATURE_INFO *vol_info;
    UINT16                    vol_val;
    NU_USBH_AUD_USR_FUNCTIONS *functions;
    UINT16                    *bitmap;
    DATA_ELEMENT             padding[1];
} NU_AUDIO_USER_DEVICE_SETTINGS;
```

The members of the NU_AUDIO_USER_DEVICE_SETTINGS data structure are defined in [Table 10-5](#).

Table 10-5. NU_AUDIO_USER_DEVICE_SETTINGS

Member	Description
aud_user_dev	Pointer to NU_USBH_AUD_USER_DEV data structure containing the audio user device information.
function	The target functionality of microphone or speaker.

Table 10-5. NU_AUDIO_USER_DEVICE_SETTINGS (cont.)

Member	Description
ch_type	Channel number of the audio user device.
mute	Flag to enable or disable mute control.
vol_info	Pointer to NU_USBH_AUD_USR_FEATURE_INFO data structure containing complete volume information for one particular channel, such as minimum, maximum, current, and resolution for the volume.
vol_value	Volume level of the audio user device.
functions	Pointer to NU_USBH_AUD_USR_FUNCTIONS data structure containing information for the supported functions.
bitmap	Pointer to the bitmap representing which controls are available in the audio user device
padding	Padding for 4-byte alignment.

Related Topics

[USBH_AUDIO_USER_GET_VOL_STATS](#) [USBH_AUDIO_USER_ADJUST_VOL](#)
[USBH_AUDIO_USER_GET_AUD_FUNCS](#) [USBH_AUDIO_USER_GET_SUPPORTED_CTRL](#)

NU_USBH_AUD_USER_DEV

NU_USBH_AUD_USER_DEV is a data structure of type _nu_usb_aud_usr_dev. NU_USBH_AUD_USER_DEV contains information about an audio user device.

```
typedef struct _nu_usb_aud_usr_dev
{
    CS_NODE                node;
    NU_USB_USER            *user_drvr;
    NU_USB_DVR             *class_drvr;
    NU_USBH_AUD_DEV        *audio_dev;
    NU_USBH_AUD_FUNC_INFO  mphone;
    NU_USBH_AUD_FUNC_INFO  speaker;
    NU_USBH_AUD_USR_FUNCTIONS supported_fnc;
} NU_USBH_AUD_USER_DEV;
```

The members of the NU_USBH_AUD_USER_DEV data structure are defined in [Table 10-6](#).

Table 10-6. NU_USBH_AUD_USER_DEV

Member	Description
node	The data structure used for managing the list of connected audio user devices.

Table 10-6. NU_USBH_AUD_USER_DEV (cont.)

Member	Description
user_drvr	Pointer to USB user driver control block.
class_drvr	Pointer to the NU_USBH_DRV data structure containing information about the USB class driver control block.
audio_dev	Pointer to the NU_USBH_AUD_DEV data structure containing the audio user device control block information.
mphone	Pointer to the NU_USBH_AUD_FUNC_INFO microphone information structure.
speaker	Pointer to the NU_USBH_AUD_FUNC_INFO speaker information structure.
supported_fnc	Pointer to the NU_USBH_AUD_USR_FUNCTIONS data structure containing the supported functions.

Related Topics

[USBH_AUDIO_USER_GET_AUD_FUNCS](#)

NU_USBH_AUD_USR_FUNCTIONS

NU_USBH_AUD_USR_FUNCTIONS is a data structure of type `_nu_usbh_aud_usr_functions`. NU_USBH_AUD_USR_FUNCTIONS is used to store the audio user-supported functions.

```
typedef struct _nu_usbh_aud_usr_functions
{
    BOOLEAN speaker;
    BOOLEAN microphone;
}NU_USBH_AUD_USR_FUNCTIONS;
```

The members of the NU_USBH_AUD_USR_FUNCTIONS data structure are defined in [Table 10-7](#).

Table 10-7. NU_USBH_AUD_USR_FUNCTIONS

Member	Description
speaker	Flag representing whether, or not, the speaker function is present in the audio device.
microphone	Flag representing whether, or not, the microphone function is present in the audio device.

Related Topics

[USBH_AUDIO_USER_GET_AUD_FUNCS](#)

Nucleus USB Host Audio Driver IOCTLs

This section provides details for IOCTLs that are defined against the `USBH_AUDIO_LABEL` and are implemented in the USB Host Audio user driver. Once a USB Audio device is open and an application has its IOCTL0 value, the application should and can exercise I/O control operations (IOCTLs) as required.

Note



The IOCTL definitions are present in file `os/include/connectivity/nu_connectivity.h` which must be included in the application source to perform these IOCTL operations.

The IOCTLs are exercised using the `DVC_Dev_Ioctl` function, as defined below:

```
STATUS DVC_Dev_Ioctl (DV_DEV_HANDLE dev_handle,
                     INT          ioctl_num,
                     VOID          *ioctl_data,
                     INT          ioctl_data_len);
```

Where the IOCTL parameters listed for each IOCTL and passed to the function call are:

- `ioctl_data` - points to a data structure containing the desired data.
- `ioctl_data_len` - contains the length of the `ioctl_data` data structure.

And example of an IOCTL call is:

```
DVC_Dev_Ioctl(device_handle,
              (usb_audio_ioctl0_base + USBH_AUDIO_OPEN_PLAY_SESSION),
              &(NU_AUDIO_DEVICE_SETTINGS),
              sizeof(NU_AUDIO_DEVICE_SETTINGS));
```

The USB Host Audio Driver IOCTLs are listed below:

- [USBH_AUDIO_OPEN_PLAY_SESSION](#)
- [USBH_AUDIO_CLOSE_PLAY_SESSION](#)
- [USBH_AUDIO_OPEN_RECORD_SESSION](#)
- [USBH_AUDIO_CLOSE_RECORD_SESSION](#)
- [USBH_AUDIO_REGISTER_RECORD_CALLBACK](#)
- [USBH_AUDIO_REGISTER_PLAY_CALLBACK](#)

- [USBH_AUDIO_PLAY_SOUND](#)
- [USBH_AUDIO_RECORD_SOUND](#)
- [USBH_AUDIO_GET_FUNC_COUNT](#)
- [USBH_AUDIO_GET_FREQ_COUNT](#)
- [USBH_AUDIO_GET_FUNC_SETTINGS](#)
- [USBH_AUDIO_REGISTER_NOTIFICATION_CALLBACKS](#)
- [USBH_AUDIO_USER_GET_VOL_STATS](#)
- [USBH_AUDIO_USER_ADJUST_VOL](#)
- [USBH_AUDIO_USER_GET_AUD_FUNCS](#)
- [USBH_AUDIO_USER_GET_SUPPORTED_CTRLs](#)
- [USBH_AUDIO_USER_GET_DEV_CB](#)

USBH_AUDIO_OPEN_PLAY_SESSION

This IOCTL creates a session to schedule the out transfers to an audio device. It creates the necessary environment for playing the sound on the audio device.

IOCTL Parameter

- Pointer to the [NU_AUDIO_DEVICE_SETTINGS](#) structure containing information about channels, sample size and sampling rate of the audio device.

Include File

os/include/connectivity/nu_connectivity.h

Parameter

[IN] Pointer to NU_AUDIO_DEVICE_SETTINGS

[OUT] Don't care

Related Topics

[NU_AUDIO_DEVICE_SETTINGS](#)

[Nucleus USB Host Audio Driver IOCTLs](#)

USBH_AUDIO_CLOSE_PLAY_SESSION

This IOCTL deletes a session created earlier to schedule the out transfers to an audio device.

IOCTL Parameter

- Pointer to the [NU_AUDIO_DEVICE_SETTINGS](#) structure containing pointer to the audio device.

Include File

os/include/connectivity/nu_connectivity.h

Parameter

[IN] Pointer to NU_AUDIO_DEVICE_SETTINGS

[OUT] Don't care

Related Topics

[NU_AUDIO_DEVICE_SETTINGS](#)

[Nucleus USB Host Audio Driver IOCTLs](#)

USBH_AUDIO_OPEN_RECORD_SESSION

This IOCTL creates a session to schedule the IN transfers to an audio device.

IOCTL Parameter

- Pointer to the [NU_AUDIO_DEVICE_SETTINGS](#) structure containing information about channels, sample size, sampling rate and double buffering enabling of the audio device.

Include File

os/include/connectivity/nu_connectivity.h

Parameter

[IN] Pointer to NU_AUDIO_DEVICE_SETTINGS

[OUT] Don't care

Related Topics

[NU_AUDIO_DEVICE_SETTINGS](#)

[Nucleus USB Host Audio Driver IOCTLs](#)

USBH_AUDIO_CLOSE_RECORD_SESSION

This IOCTL deletes a session created earlier to schedule the in transfers to an audio device.

IOCTL Parameter

- Pointer to the [NU_AUDIO_DEVICE_SETTINGS](#) structure containing pointer to the audio device.

Include File

os/include/connectivity/nu_connectivity.h

Parameter

[IN] Pointer to NU_AUDIO_DEVICE_SETTINGS

[OUT] Don't care

Related Topics

[NU_AUDIO_DEVICE_SETTINGS](#)

[Nucleus USB Host Audio Driver IOCTLs](#)

USBH_AUDIO_REGISTER_RECORD_CALLBACK

An application may execute this IOCTL to register a callback which will be invoked when recording is completed.

IOCTL Parameter

- Pointer to the [NU_USBH_AUD_Data_Callback](#) data structure.

Include File

os/include/connectivity/nu_connectivity.h

Parameter

[IN] Pointer to to NU_USBH_AUD_Data_Callback.

[OUT] Don't care

Related Topics

[NU_USBH_AUD_Data_Callback](#)

[Nucleus USB Host Audio Driver IOCTLs](#)

USBH_AUDIO_REGISTER_PLAY_CALLBACK

An application may execute this IOCTL to register a callback which will be invoked when playing sound is completed.

IOCTL Parameter

- Pointer to the [NU_USBH_AUD_Data_Callback](#) data structure.

Include File

os/include/connectivity/nu_connectivity.h

Parameter

[IN] Pointer to to NU_USBH_AUD_Data_Callback.

[OUT] Don't care

Related Topics

[NU_USBH_AUD_Data_Callback](#)

[Nucleus USB Host Audio Driver IOCTLs](#)

USBH_AUDIO_PLAY_SOUND

This IOCTL starts playing sound on an audio device with the data rate specified when opening the play session.

IOCTL Parameter

- Pointer to [NU_AUDIO_OP_PARAM](#) structure containing pointer to audio clip buffer and audio clip length in terms of samples of the audio device.

Include File

os/include/connectivity/nu_connectivity.h

Parameter

[IN] Pointer to to [NU_AUDIO_OP_PARAM](#).

[OUT] Don't care

Related Topics

[NU_AUDIO_OP_PARAM](#)

[Nucleus USB Host Audio Driver IOCTLs](#)

USBH_AUDIO_RECORD_SOUND

This IOCTL starts recording sound on an audio device with data rate specified earlier while opening the recording session.

IOCTL Parameter

- Pointer to [NU_AUDIO_OP_PARAM](#) structure containing pointer to audio clip buffer and audio clip length in terms of samples of the audio device.

Include File

os/include/connectivity/nu_connectivity.h

Parameter

[IN] Pointer to to NU_AUDIO_OP_PARAM.

[OUT] Don't care

Related Topics

[NU_AUDIO_OP_PARAM](#)

[Nucleus USB Host Audio Driver IOCTLs](#)

USBH_AUDIO_GET_FUNC_COUNT

This IOCTL counts the number of playback or recording functions available in the device and return it to the caller.

IOCTL Parameter

- Pointer to [NU_AUDIO_DEVICE_SETTINGS](#) structure containing the microphone or speaker function and func_count in which number of functions available in the device are returned.

Include File

os/include/connectivity/nu_connectivity.h

Parameter

[IN] Pointer to to [NU_AUDIO_DEVICE_SETTINGS](#).

[OUT] func_count, a member of [NU_AUDIO_DEVICE_SETTINGS](#) structure containing the number of functions available in the device.

Related Topics

[NU_AUDIO_DEVICE_SETTINGS](#)

[Nucleus USB Host Audio Driver IOCTLs](#)

USBH_AUDIO_GET_FREQ_COUNT

This ioctl counts the number of sampling frequencies supported by passed playback or recording function and return it to the caller.

IOCTL Parameter

- Pointer to [NU_AUDIO_DEVICE_SETTINGS](#) structure containing the microphone or speaker function , function index and freq_count in which number of frequencies supported by the device are returned.

Include File

os/include/connectivity/nu_connectivity.h

Parameter

[IN] Pointer to to [NU_AUDIO_DEVICE_SETTINGS](#).

[OUT] freq_count containing number of frequencies supported by the device.

Related Topics

[NU_AUDIO_DEVICE_SETTINGS](#)

[Nucleus USB Host Audio Driver IOCTLs](#)

USBH_AUDIO_GET_FUNC_SETTINGS

This IOCTL returns the playback or recording settings supported by the playback or recording function and return it to the caller.

IOCTL Parameter

- Pointer to [NU_AUDIO_DEVICE_SETTINGS](#) structure containing information about the function of the audio device. Type of supported sampling frequencies, list of frequencies, number of frequencies , number of channels, sample size and lock delay values are returned to the caller.

Include File

os/include/connectivity/nu_connectivity.h

Parameter

[IN] Pointer to to NU_AUDIO_DEVICE_SETTINGS.

[OUT] freq_type, freq_list, freq_count, channels, sample size and lock delay values supported by the audio device.

Related Topics

[NU_AUDIO_DEVICE_SETTINGS](#)

[Nucleus USB Host Audio Driver IOCTLs](#)

USBH_AUDIO_REGISTER_NOTIFICATION_CALLBACKS

This IOCTL is called to register connection and disconnection callback functions with audio host user driver.

IOCTL Parameter

- Pointer to [NU_USBH_AUDIO_NOTIFY_CALLBACKS](#).

Include File

os/include/connectivity/nu_connectivity.h

Parameter

[IN] Pointer to [NU_USBH_AUDIO_NOTIFY_CALLBACKS](#).

[OUT] Don't care.

Related Topics

[NU_USBH_AUDIO_NOTIFY_CALLBACKS](#) [Nucleus USB Host Audio Driver IOCTLs](#)

USBH_AUDIO_USER_GET_VOL_STATS

This IOCTL returns the volume attributes related to a particular feature unit. It returns minimum, maximum, current and resolution of volume in `vol_info` structure.

IOCTL Parameter

- Pointer to [NU_AUDIO_USER_DEVICE_SETTINGS](#) containing microphone or speaker function of the audio device and channel number (`ch_type`) for which volume is required. Complete volume information (`vol_info`) for one particular channel.

Include File

os/include/connectivity/nu_connectivity.h

Parameter

[IN] Pointer to `NU_AUDIO_USER_DEVICE_SETTINGS` containing function and `ch_type` of audio device.

[OUT] `vol_info` of the audio device.

Related Topics

[NU_AUDIO_USER_DEVICE_SETTINGS](#) [Nucleus USB Host Audio Driver IOCTLs](#)

USBH_AUDIO_USER_ADJUST_VOL

This IOCTL sets the current volume for the desired Feature Unit in an Audio device.

IOCTL Parameter

- Pointer to [NU_AUDIO_USER_DEVICE_SETTINGS](#) containing microphone or speaker functions of the audio device and channel number (ch_type) for the volume to be adjusted. Volume level that is set is placed in vol_val.

Include File

os/include/connectivity/nu_connectivity.h

Parameter

[IN] Pointer to NU_AUDIO_USER_DEVICE_SETTINGS containing the function, ch_type and vol_value to be set for the audio device.

[OUT] Don't care.

Related Topics

[NU_AUDIO_USER_DEVICE_SETTINGS](#) [Nucleus USB Host Audio Driver IOCTLs](#)

USBH_AUDIO_USER_GET_AUD_FUNCS

This IOCTL informs the user of functions present in the Audio device. The device can support speaker and microphone.

IOCTL Parameter

- Pointer to [NU_AUDIO_USER_DEVICE_SETTINGS](#) containing a pointer to the [NU_USBH_AUD_USER_DEV](#) structure. A pointer to [NU_USBH_AUD_USR_FUNCTIONS](#) is returned containing information about the supported functions.

Include File

os/include/connectivity/nu_connectivity.h

Parameter

[IN] Pointer to [NU_AUDIO_USER_DEVICE_SETTINGS](#) containing the function, ch_type and vol_value to be set for the audio device.

[OUT] Don't care.

Related Topics

NU_AUDIO_USER_DEVICE_SETTINGS	NU_USBH_AUD_USER_DEV
NU_USBH_AUD_USR_FUNCTIONS	Nucleus USB Host Audio Driver IOCTLs

USBH_AUDIO_USER_GET_SUPPORTED_CTRLs

This IOCTL returns supported features against a particular channel index in the bitmap.

IOCTL Parameter

- Pointer to the [NU_AUDIO_USER_DEVICE_SETTINGS](#) data structure containing the microphone or speaker functions, channel number (ch_type) of the audio device, and the bitmap representing which controls are available.

Include File

os/include/connectivity/nu_connectivity.h

Parameter

[IN] Pointer to NU_AUDIO_USER_DEVICE_SETTINGS containing function and channel number ch_type.

[OUT] Bitmap representing which controls are available against a particular channel.

Related Topics

[NU_AUDIO_USER_DEVICE_SETTINGS](#) [Nucleus USB Host Audio Driver IOCTLs](#)

USBH_AUDIO_USER_GET_DEV_CB

This IOCTL returns a pointer to NU_USBH_AUD_USER_DEV structure for the application.

IOCTL Parameter

- Pointer to VOID containing the [NU_USBH_AUD_USER_DEV](#) pointer is returned.

Include File

os/include/connectivity/nu_connectivity.h

Parameter

[IN] Don't care.

[OUT] Pointer to VOID that will contain pointer to NU_USBH_AUD_USER_DEV.

Related Topics

[NU_USBH_AUD_USER_DEV](#)

[Nucleus USB Host Audio Driver IOCTLs](#)

Chapter 11

USB Device Firmware Upgrade Driver

Nucleus ReadyStart contains a USB Device Firmware Upgrade (DFU) driver for the USB function side. This chapter describes the support of the Nucleus USB Function DFU Driver in detail.

Caution



To guarantee future support and compatibility for your application, use only documented Nucleus interfaces, structures, macros, and so on.

USB Function DFU Driver Overview

The USB Function DFU Class Driver provides runtime firmware download and upload functionality for USB devices. This functionality can be enabled in USB devices such as cameras, mass storage devices, printers, and scanners. This driver component is meant only for firmware upgrades; the primary USB device functionality is provided by other class drivers. Therefore, the device firmware upgrade is under the control of other class drivers in almost all cases.

Specifications defining which USB device drivers work are defined by their respective device class specification standards. The DFU class is defined by USB Device class specification for device firmware upgrade. The USB Implementers Forum, Inc. (USB-IF) specification for Device Firmware Upgrade can be downloaded at the following location:

http://www.usb.org/developers/devclass_docs/DFU_1.1.pdf

This specification defines which device can communicate with the host, the method for uploading and downloading firmware on a device, and cases in which a device should respond with an error.

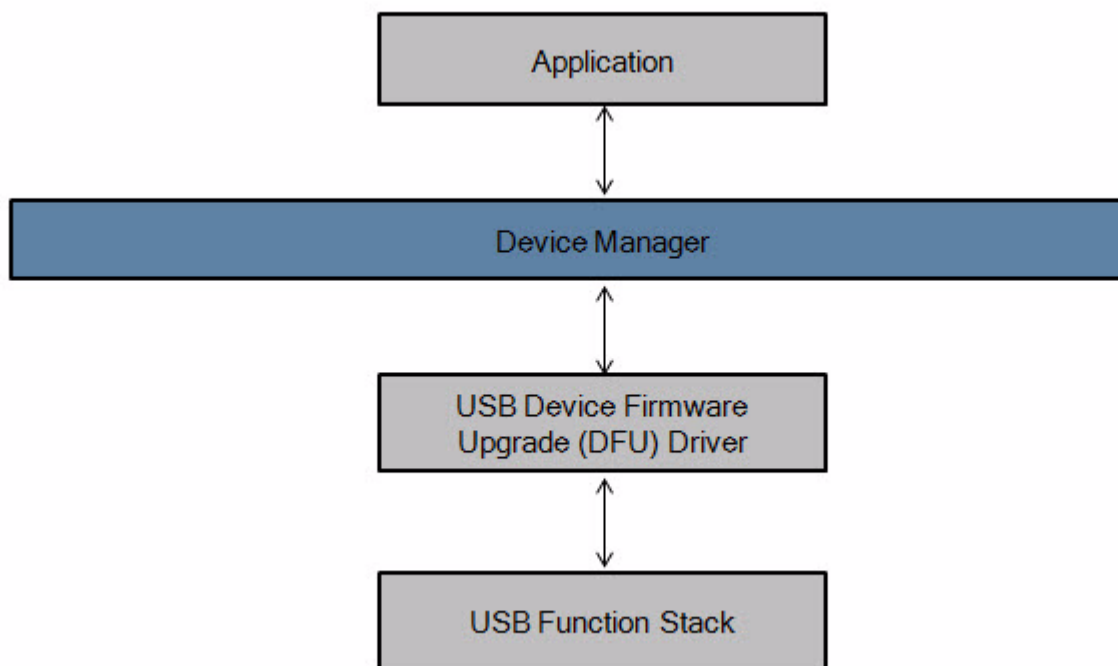
There are two available modes in each DFU-capable device, normal mode and DFU mode. During a firmware upgrade, it is not possible for the device to continue its normal operation; therefore, the mode of the device is required to be changed to the DFU mode. This mode change done using external human intervention. According to USB Device Class Specification for DFU, a device has to go through four distinct phases in order to complete a successful firmware upgrade.

1. Enumeration: Once the device is connected with the host in normal mode, it informs the host of its capabilities. In normal runtime mode, the device has one extra interface descriptor. This interface descriptor tells host about its DFU capabilities.

2. **Reconfiguration:** When the host and device agree to upgrade the firmware, the host sends a class type request to the device. After receiving the request, the device waits for a USB reset from the host for a prespecified time. If the USB reset event occurs within the prespecified time, the device deactivates its normal runtime capabilities and switches to DFU mode.
3. **Transfer:** Firmware can be downloaded or uploaded in this phase. Correct block sizes used to transfer firmware are mentioned in the device descriptor. Transfer is acknowledged, when completed, to maintain synchronization between devices.
4. **Manifestation:** After the firmware upgrade, the device notifies the host that it has reprogrammed its memory and now it is ready. A USB reset from the host converts the operating mode of device back to normal. Once the operating mode is switched, the device starts executing the upgraded firmware.

The hierarchy for the DFU driver implementation in Nucleus is shown in [Figure 11-1](#).

Figure 11-1. USB Function DFU Driver Block Diagram



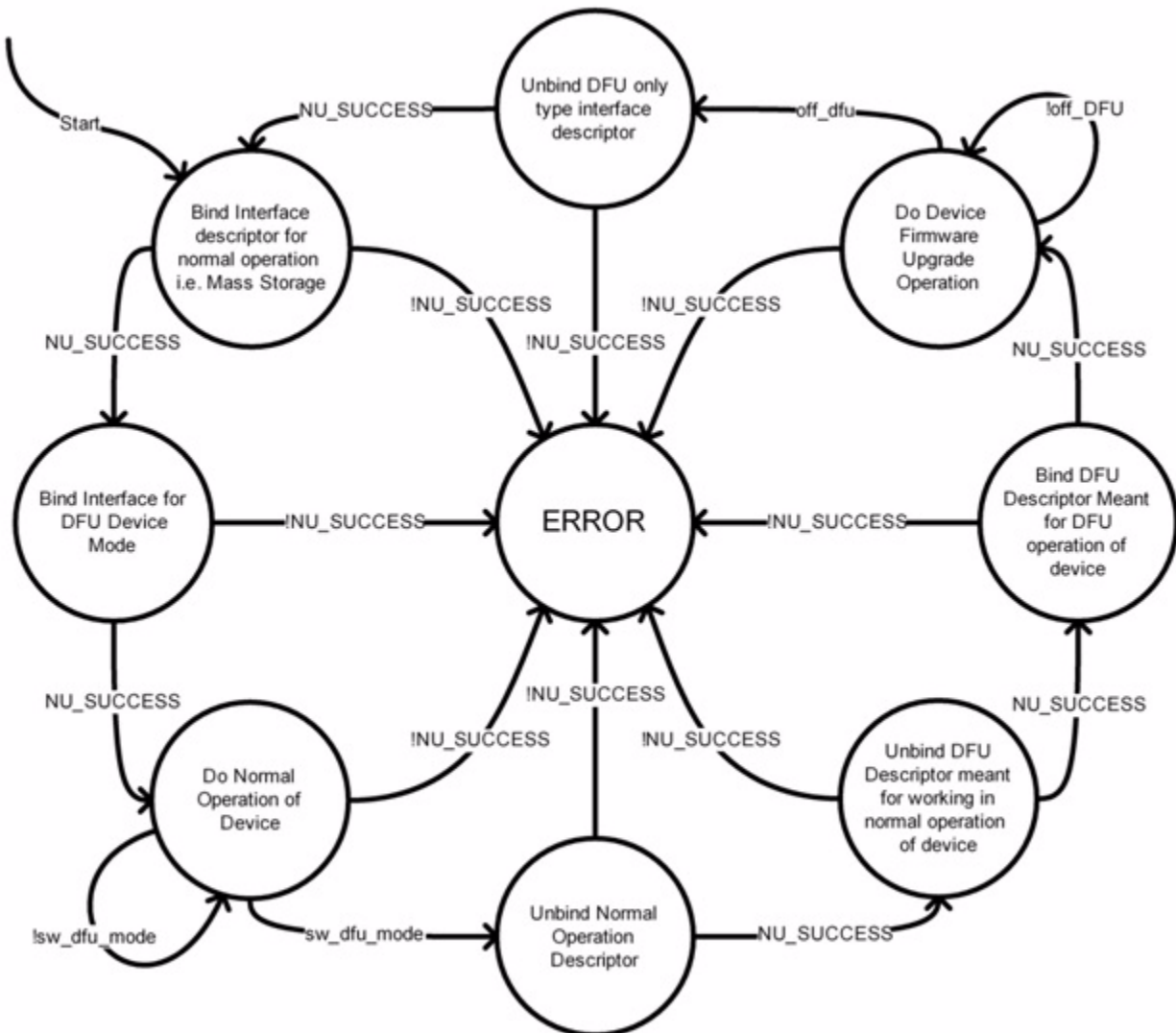
USB Function DFU Driver Operation

This section provides details on various aspects of operation of USB function device firmware upgrade driver. It includes run level initialization, mode switching and user configurable options.

USB DFU Mode Switching Sequence

If device firmware is required to be upgraded, both of the normal operating descriptors are unbound from the configuration descriptor. This unbinding operation can be performed by the application by using open and close system calls to the Device Manager. Once these descriptors are unbound, a DFU type descriptor is then registered. This mode of operation will be used to upload or download the firmware on the device. After the desired operation is performed, the device mode is converted back to normal. An illustration of switching descriptor sequence is provided in [Figure 11-2](#).

Figure 11-2. USB Function DFU Driver Descriptor Switching



Descriptor Switching User-Configurable Parameters

The user-configurable parameters for the USB Function DFU Descriptor switching are described below. These options are defined in metadata file of USB function DFU driver which can be found at *os/connectivity/usb/function/dfu/metadata*. The user can set these values through the build configuration.

- detach_timeout

The amount of time, in milliseconds, for which the function device has to wait after it receives DFU_DETACH from the host. If this time elapses without a USB reset, the function device terminates the reconfiguration phase and goes back to the normal mode. Otherwise, the device switches to DFU mode. Default value is 10.

- poll_time

The amount of time, in milliseconds, for which host must wait before sending the GETSTATUS request after a firmware download. Default Value is 30.

- transfer_size

The maximum number of bytes that the device can accept per control write transaction. Default value is 256.

- timer_ticks

The total number of timer ticks that make up one millisecond. Default value is 10.

- detach_timeout_ticks

This value is used for creating the detach timer and represents the initial number of timer ticks. Default Value is 1000.

- bitwill_detach

This value controls detachment; 0 for false (default value), 1 for true. The value set in this option affects the bit3:bitWillDetach of the bmAttributes field of DFU functional descriptor.

- bitmanifest_tolerant

Enables the host to communicate during download and upload. Default value is true. The value set in this option affects bit2:bitManifestationTolerant of the bmAttributes field of the DFU functional descriptor.

- bit_canupload

Enables the device to upload firmware to the host. Default value is true. The value set in this option affects bit1:bitCanUpload of the bmAttributes field of the DFU functional descriptor.

- bit_candnload

Enables the device to download firmware from the host. Default value is true. The value set in this option affects bit0:bitCanDnload of the bmAttributes field of the DFU functional descriptor.

USB Function DFU Driver Build Configuration

In order to use the DFU class driver, the following build configuration components should be enabled.

```
nu.os.conn.usb.func.dfu.enable = true
```

Using DFU with some primary function drivers, such as mass storage, requires additional components to be enabled such as:

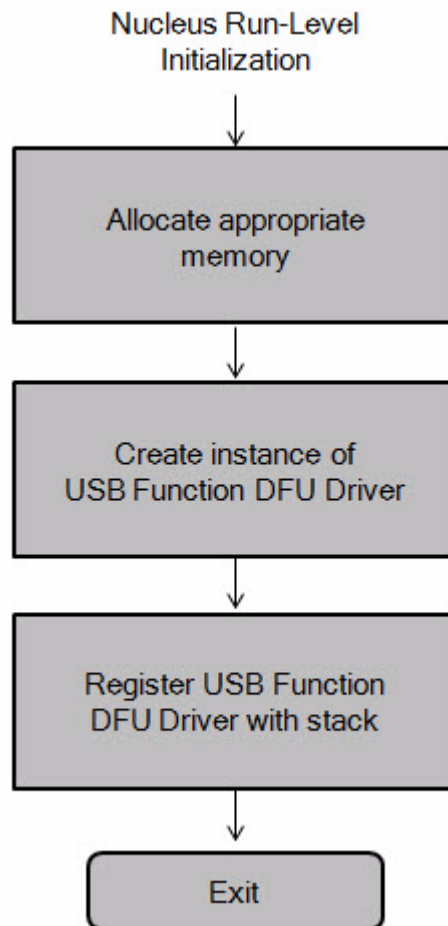
```
nu.os.conn.usb.func.ms.class.enable = true  
nu.os.conn.usb.func.ms.user.enable = true
```

Other dependent components, such as concerning hardware components, base drivers, and USB common and function stack are also required to be enabled. For more information about creating a custom configuration, see "Creating a Custom Configuration", in the chapter "Getting Started with Nucleus ReadyStart Using Sourcery Code Bench", in the *Nucleus ReadyStart Guide*.

Run level Initialization

To initialize the DFU component, it must have an initialization function that will run at its respective run level during Nucleus ReadyStart initialization. This function creates the instance of the DFU driver, adds separate functional descriptors for normal and DFU mode for the device, and then registers the driver with the USB function stack. The initialization sequence is shown in [Figure 11-3](#).

Figure 11-3. USB Function DFU Driver Initialization Sequence



USB Function DFU Driver Application Interface

Application interfaces for the USB DFU class driver use the standard Device Manager interface. The USB Function DFU Driver gets registered with two labels: USBF_DFU_RTM_LABEL and USBF_DFU_STDA_LABEL. The application can then interact with the driver using standard system calls.

Device Open/Close

After the DFU device driver gets registered, it can be opened and closed by the application using [DVC_Dev_Open](#) or [DVC_Dev_ID_Open](#) and [DVC_Dev_Close](#) respectively.

Read/Write Operation

Standard read and write operations are called using [DVC_Dev_Read](#) and [DVC_Dev_Write](#) functions. The DFU driver always returns a success status, as these functions are not used for any internal operations.

IOCTL Execution

In order to execute an IOCTL on a device, an application calls the `DVC_Dev_Ioctl` function. Additional information for IOCTL execution is provided with the [USB Function DFU Driver IOCTLs](#).

USB Function DFU Driver IOCTL Data Structures

This section covers data structures used by the [USB Function DFU Driver IOCTLs](#).

USBF_DFU_APP_CALLBACK

`USBF_DFU_APP_CALLBACK` is a data structure of type `_usbf_dfu_app_callback`. `USBF_DFU_APP_CALLBACK` contains pointers to the callback functions of the application.

```
typedef struct _usbf_dfu_app_callback
{
    NU_USBF_DFU_USR_CALLBACK dfu_user_callback;
    VOID *dfu_data_context;
} USBF_DFU_APP_CALLBACK;
```

The members of the `USBF_DFU_APP_CALLBACK` data structure are defined in [Table 11-1](#).

Table 11-1. USBF_DFU_APP_CALLBACK

Member	Description
<code>dfu_user_callback</code>	<code>NU_USBF_DFU_USR_CALLBACK</code> data structure containing pointers to the callback functions of the application. See the USBF_DFU_IOCTL_REG_CALLBACK IOCTL function description for a list of callbacks that must be registered with the DFU class driver.
<code>dfu_data_context</code>	Pointer to the user context data

Related Topics

[USBF_DFU_IOCTL_REG_CALLBACK](#)

[USB Function DFU Driver IOCTLs](#)

USB Function DFU Driver IOCTLs

This section provides details for IOCTLs that are defined against the `USBF_DFU_RTM_LABEL` and `USBF_DFU_STDA_LABEL` labels and are implemented in the USB Function DFU driver. Once a USB DFU device driver is open and an application has its IOCTL0 value, the application should and can exercise I/O control operations (IOCTLs) as required.

Note



The IOCTL definitions are present in file *os/include/connectivity/nu_connectivity.h* which must be included in the application source to perform these IOCTL operations.

The IOCTLs are exercised using the `DVC_Dev_Ioctl` function, as defined below:

```
STATUS DVC_Dev_Ioctl (DV_DEV_HANDLE dev_handle,  
                     INT             ioctl_num,  
                     VOID            *ioctl_data,  
                     INT             ioctl_data_len);
```

Where the IOCTL parameters listed for each IOCTL and passed to the function call are:

- `ioctl_data` - points to a data structure containing the desired data.
- `ioctl_data_len` - contains the length of the `ioctl_data` data structure.

And example of an IOCTL call is:

```
DVC_Dev_Ioctl(device_handle,  
              (usb_dfu_ioctl0_base + USBF_DFU_IOCTL_SET_STATUS),  
              (VOID*)&dfu_status  
              sizeof(UINT8));
```

The USB Function DFU Driver IOCTLs are listed below:

- [USBF_DFU_IOCTL_SET_STATUS](#)
- [USBF_DFU_IOCTL_GET_STATE](#)
- [USBF_DFU_IOCTL_GET_STATUS](#)
- [USBF_DFU_IOCTL_REG_CALLBACK](#)
- [USBF_DFU_IOCTL_UNREG_CALLBACK](#)

USBF_DFU_IOCTL_SET_STATUS

This IOCTL allows the application to set the status of the DFU while processing the firmware upload or download. For example, if the DFU application is not able to write the firmware properly into memory, it can set the Error as Firmware Error Write.

IOCTL Parameter

- Numeric Value for the status to be set by the application. [Table 11-2](#) shows the corresponding status that will be set for each assignable numeric value.

Table 11-2. USBF_DFU_IOCTL_SET_STATUS Lookup Table

Value	Status	Description
1	DFU_SPEC_STATUS_OK	An application may set this status if there is no error condition.
2	DFU_SPEC_STATUS_ERR_TRGT	A file is not targeted for use by this device.
3	DFU_SPEC_STATUS_ERR_FILE	A file is targeted for this device, but fails a vendor-specific verification test.
4	DFU_SPEC_STATUS_ERR_WRITE	The device is unable to write memory.
5	DFU_SPEC_STATUS_ERR_ERASE	The memory erase function failed.
6	DFU_SPEC_STATUS_ERR_CHK_ERASED	The memory erase check failed.
7	DFU_SPEC_STATUS_ERR_PROG	The program memory function failed.
8	DFU_SPEC_STATUS_ERR_VRFY	The programmed memory failed verification.
9	DFU_SPEC_STATUS_ERR_ADDRS	Can not program memory because the received address is out of range.
10	DFU_SPEC_STATUS_ERR_NOTDONE	Received the end of download file, but the device does not think it has all of the data yet.
11	DFU_SPEC_STATUS_ERR_FIRMWARE	The device firmware is corrupt. Device can not return to run-time (non-DFU) operation.
12	DFU_SPEC_STATUS_ERR_VNDR	A vendor-specific error has been received.
13	DFU_SPEC_STATUS_ERR_USBR	The device detected an unexpected USB reset signal.

Table 11-2. USBF_DFU_IOCTL_SET_STATUS Lookup Table (cont.)

Value	Status	Description
14	DFU_SPEC_STATUS_ERR_POR	The device detected an unexpected power on reset.
15	DFU_SPEC_STATUS_ERR_UNKNOWN	An unknown error occurred.
16	DFU_SPEC_STATUS_ERR_STALLEDPKT	The device stalled on an unexpected request.

Include File

os/include/connectivity/nu_connectivity.h

Parameter

[IN] Value of the Status to be set.

[OUT] Don't Care.

Related Topics

[USB Function DFU Driver IOCTLs](#)

USBF_DFU_IOCTL_GET_STATE

This IOCTL gets the current state of the DFU driver. This may help in checking the current device mode.

IOCTL Parameter

- Numeric Value returned by the DFU Class Driver. [Table 11-3](#) is the lookup table to check the internal state on the basis of value returned.

Table 11-3. USBF_DFU_IOCTL_GET_STATE Lookup Table

Value	Status	Description
1	DFUF_STATE_APPL_IDLE	The device is running is normal application.
2	DFUF_STATE_APPL_DETACH	The device is running is normal application, has received the DFU_DETACH request, and is waiting for a USB reset.
3	DFUF_STATE_DFU_IDLE	The device is operating in the DFU mode and is waiting for requests.
4	DFUF_STATE_DFU_DNLOAD_SYNC	The device has received a block and is waiting for the host to solicit the status via DFU_GETSTATUS.
5	DFUF_STATE_DFU_DNBUSY	The device is programming a control-write block into its nonvolatile memory.
6	DFUF_STATE_DFU_DNLOAD_IDLE	The device is processing a download operation and is expecting DFU_DNLOAD requests.
7	DFUF_STATE_DFU_MNFST_SYNC	The device has received the final block of firmware from the host and is waiting for receipt of DFU_GETSTATUS to begin the Manifestation phase; or device has completed the Manifestation phase and is waiting for receipt of DFU_GETSTATUS. (Devices that can enter this state after the Manifestation phase set bmAttributes bit bitManifestationTolerant to 1).
8	DFUF_STATE_DFU_MNFST	The device is in the manifestation phase.

Table 11-3. USBF_DFU_IOCTL_GET_STATE Lookup Table (cont.)

Value	Status	Description
9	DFUF_STATE_DFU_MNFST_WAIT_RSET	The device has programmed memory and is waiting for a USB reset or power on reset.
10	DFUF_STATE_DFU_UPLOAD_IDLE	The device is processing an upload operation and expecting DFU_UPLOAD requests.
11	DFUF_STATE_DFU_ERR	An error has occurred. Awaiting the DFU_CLRSTATUS request.

Include File

os/include/connectivity/nu_connectivity.h

Parameter

[IN] Don't Care.

[OUT] Internal State of DFU Driver.

Related Topics

[USB Function DFU Driver IOCTLs](#)

USBF_DFU_IOCTL_GET_STATUS

This IOCTL notifies the application of the current status of operation being performed by the DFU system.

IOCTL Parameter

- The values returned by the class driver are listed in [Table 11-2](#) under the [USBF_DFU_IOCTL_SET_STATUS](#) IOCTL definition.

Include File

os/include/connectivity/nu_connectivity.h

Parameter

[IN] Don't Care.

[OUT] Returns the current status of operation.

Related Topics

[USB Function DFU Driver IOCTLs](#)

[USBF_DFU_IOCTL_SET_STATUS](#)

USBF_DFU_IOCTL_REG_CALLBACK

This IOCTL allows the application to register its callback functions with the DFU class driver.

IOCTL Parameter

- The [USBF_DFU_APP_CALLBACK](#) data structure containing pointers to the functions to be registered. [Table 11-4](#) shows the list of functions, with their descriptions, that are required to be registered with the DFU Class driver.

Table 11-4. Callbacks To Be Registered To The DFU Driver

Status	Description
Data Transfer Callback	Used for transferring data between class driver and application and vice versa.
Event Callback	Used for informing the DFU application about the occurrence of event. For example the events of USB connect, disconnect or reset.
Status Callback	This callback function informs the status of operation to DFU Application.
Request Callback	This callback function is called when DFU class driver receives any request.
PollTime Callback	This is the time for which device can tell host to wait for before acquiring the next status of device.
User Context	It saves the pointer to the control block of application.

Include File

os/include/connectivity/nu_connectivity.h

Parameter

[IN] [USBF_DFU_APP_CALLBACK](#) data structure containing the callback function pointers and firmware control block.

[OUT] Don't Care.

Related Topics

[USBF_DFU_APP_CALLBACK](#)

[USBF_DFU_IOCTL_UNREG_CALLBACK](#)

[USB Function DFU Driver IOCTLs](#)

USBF_DFU_IOCTL_UNREG_CALLBACK

This IOCTL unregisters an application from the DFU class driver.

IOCTL Parameter

- No parameters are required or returned.

Include File

os/include/connectivity/nu_connectivity.h

Parameter

[IN] Don't care.

[OUT] Don't care.

Related Topics

[USB Function DFU Driver IOCTLs](#)

[USBF_DFU_IOCTL_REG_CALLBACK](#)

Chapter 12

USB Host and Functions

This chapter describes the following services provided in the USB software module for the USB host and functions:

- [Nucleus USB Services](#)
- [Nucleus USB Alternate Setting Services](#)
- [Nucleus USB Binary Object Store \(BOS\) Services](#)
- [Nucleus USB Config Services](#)
- [Nucleus USB Device Services](#)
- [Nucleus USB Driver Services](#)
- [Nucleus USB ENDPoint Services](#)
- [Nucleus USB Interface Association Descriptor \(IAD\) Services](#)
- [Nucleus USB Interface Services](#)
- [Nucleus USB IRP Services](#)
- [USB Memory Component](#)
- [Nucleus USB Pipe Services](#)
- [Nucleus USB Power Management Services](#)
- [USB Function Device Configuration Services](#)
- [Nucleus USB Bulk Streaming Services](#)
- [Nucleus USB User Services](#)
- [Nucleus USB Host Services](#)
- [Nucleus USB Host Control IRP Services](#)
- [Nucleus USB Host Class Driver Services](#)
- [Nucleus USB Host Hardware Driver Services](#)
- [Nucleus USB Host STACK Services](#)
- [Nucleus USB Host User Driver Services](#)
- [Nucleus USB Function Class Driver Services](#)

- [USB Function Device Configuration Services](#)
- [Nucleus USB Function Hardware Driver Services](#)
- [Nucleus USB Function STACK Services](#)
- [Nucleus USB Function Services](#)
- [Nucleus USB Function User Driver Services](#)
- [Dispatch Table Reference](#)
- [Nucleus USB Function Stack Initialization](#)
- [Nucleus USB Host Stack Initialization](#)
- [Nucleus USB Function and Host IOCTLs](#)

Caution



To guarantee future support and compatibility for your application, use only documented Nucleus interfaces, structures, macros, and so on.

Nucleus USB Services

This section provides a detailed reference of the following Nucleus USB Services:

- [NU_USB_Delete](#)
- [NU_USB_Get_Name](#)
- [NU_USB_Get_Object_Id](#)

Related Topics

[USB Host and Functions](#)

NU_USB_Delete

This function calls the function pointer installed in the dispatch table, so any derivatives of the NU_USB control block can be deleted using this function.

Usage

```
STATUS NU_USB_Delete (VOID *cb)
```

Arguments

- **cb**
Pointer to the control block. Can be any derivative of NU_USB.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_NOT_PRESENT**
Indicates that the control block to be deleted is invalid.

Description

This function must be used to delete Nucleus USB components that are derivatives of NU_USB. Every Nucleus USB component that can be instantiated has a <Component_Name>_Create API for creation.

All Nucleus USB components fall in to three categories:

- Those that derive from NU_USB.
- Those that do not derive from NU_USB.
- Those that do not have <Component_Name>_Create and <Component_Name>_Delete APIs.

The direct derivatives of NU_USB are NU_USB_STACK, NU_USB_HW, NU_USB_STACK, NU_USB_DRV, and NU_USB_USER. Apart from these there are indirect derivatives of NU_USB such as NU_USB_STACK, which is a derivative of NU_USB_STACK that derives from NU_USB. Hence NU_USB_STACK is also a derivative of NU_USB. This category of Nucleus USB components are deleted using NU_USB_Delete API and such components do not have any of their own <Component_Name>_Delete API.

There are several Nucleus USB components that are not derived from NU_USB (like NU_USB_HWENV) and so they have their own <Component_Name>_Delete APIs.

The third category of Nucleus USB components is those whose life time (including creation and extinction) is controlled by Nucleus USB itself (such as NU_USB_PIPE, NU_USB_CFG, and

so on), and hence such components do not have <Component_Name>_Create and <Component_Name>_Delete APIs.

Related Topics

[Nucleus USB Services](#)

NU_USB_Get_Name

This function returns the name (NULL terminated seven-character string) assigned during creation of the Nucleus USB component. This can only be invoked on those Nucleus USB components that are derivatives of NU_USB.

Usage

```
STATUS NU_USB_Get_Name (NU_USB *cb,  
                        CHAR *name_out)
```

Arguments

- **cb**
Pointer to the control block. Can be any derivative of NU_USB.
- **name_out**
Pointer to the array to hold the NULL terminated seven-character string.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Services](#)

NU_USB_Get_Object_Id

This function returns the object assigned during the creation of the Nucleus USB component. This can only be invoked on those Nucleus USB components that are derivatives of NU_USB.

Usage

```
STATUS NU_USB_Get_Object_Id (NU_USB *cb,  
                             UINT32 *id_out)
```

Arguments

- **cb**
Pointer to the control block. Can be any derivative of NU_USB.
- **id_out**
Pointer to the variable to hold the object ID.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Services](#)

Nucleus USB Alternate Setting Services

This section provides a detailed reference of the following Nucleus USB Alternate Setting Services:

- [NU_USB_ALT_SETTG_Find_Pipe](#)
- [NU_USB_ALT_SETTG_Get_bAlternateSetting](#)
- [NU_USB_ALT_SETTG_Get_Class](#)
- [NU_USB_ALT_SETTG_Get_Class_Desc](#)
- [NU_USB_ALT_SETTG_Get_Desc](#)
- [NU_USB_ALT_SETTG_Get_Endp](#)
- [NU_USB_ALT_SETTG_Get_Num_Endps](#)
- [NU_USB_ALT_SETTG_Get_Protocol](#)
- [NU_USB_ALT_SETTG_Get_String](#)
- [NU_USB_ALT_SETTG_Get_String_Desc](#)
- [NU_USB_ALT_SETTG_Get_String_Num](#)
- [NU_USB_ALT_SETTG_Get_SubClass](#)
- [NU_USB_ALT_SETTG_Set_Active](#)

Related Topics

[USB Host and Functions](#)

NU_USB_ALT_SETTG_Find_Pipe

This function finds a matching pipe among those contained in the alternate setting. `match_flag` specifies the search criteria and other arguments provide the search keys.

Usage

```
STATUS NU_USB_ALT_SETTG_Find_Pipe (NU_USB_ALT_SETTG *cb,
                                   UINT32          match_flag,
                                   UINT8           endp_num,
                                   UINT8           direction,
                                   UINT8           type,
                                   NU_USB_PIPE     **pipe_out)
```

Arguments

- `cb`
 Pointer to the alternate setting control block.
- `match_flag`
 OR any of the following:
 - USB_MATCH_EP_ADDRESS
 Search for the pipe with endpoint number specified in `endp_num`.
 - USB_MATCH_EP_TYPE
 Search for the pipe whose endpoint type is same as that specified in `type`.
 - USB_MATCH_EP_DIRECTION
 Search for the pipe whose endpoint direction is same as that specified in `type`.
- `endp_num`
 Endpoint number in the range of 0 to 15.
- `direction`
 Either `USB_DIR_IN` or `USB_DIR_OUT`.
- `type`
 Can be one of the following:
 - USB_EP_CTRL
 - USB_EP_ISO
 - USB_EP_BULK
 - USB_EP_INTR.
- `pipe_out`
 Pointer to the memory location to hold the pointer to the matching pipe.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_NOT_PRESENT**
Indicates that no matching pipe could be found.
- **NU_USB_INVLD_ARG**
Indicates that the control block is invalid.

Related Topics

[Nucleus USB Alternate Setting Services](#)

NU_USB_ALT_SETTG_Get_bAlternateSetting

This function returns the bAlternateSetting field of the associated interface descriptor in bAlternateSetting_out.

Usage

```
STATUS NU_USB_ALT_SETTG_Get_bAlternateSetting (  
    NU_USB_ALT_SETTG *cb,  
    UINT8 *bAlternateSetting_out)
```

Arguments

- **cb**
Pointer to the alternate setting control block.
- **bAlternateSetting_out**
Pointer to the variable to hold the bAlternateSetting field of the associated interface descriptor.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Alternate Setting Services](#)

NU_USB_ALT_SETTG_Get_Class

This function returns the bInterfaceClass field of the associated interface descriptor, in bInterfaceClass_out.

Usage

```
STATUS NU_USB_ALT_SETTG_Get_Class (NU_USB_ALT_SETTG *cb,  
                                  UINT8 *bInterfaceClass_out)
```

Arguments

- **cb**
Pointer to the alternate setting control block.
- **bInterfaceClass_out**
Pointer to the variable to hold the bInterfaceClass field of the associated interface descriptor.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Alternate Setting Services](#)

NU_USB_ALT_SETTG_Get_Class_Desc

This function returns a pointer to the class-specific interface descriptor (in class_desc_out) and its length in length_out.

Usage

```
STATUS NU_USB_ALT_SETTG_Get_Class_Desc (
                                NU_USB_ALT_SETTG *cb,
                                UINT8             **class_desc_out,
                                UINT32            *length_out)
```

Arguments

- **cb**
Pointer to the alternate setting control block.
- **class_desc_out**
Pointer to a memory location to hold the pointer to class specific interface descriptor.
- **length_out**
Pointer to the variable to hold the length of the descriptor in bytes.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Alternate Setting Services](#)

NU_USB_ALT_SETTG_Get_Desc

This function returns a pointer to the associated interface descriptor in `intf_desc_out`.

Usage

```
STATUS NU_USB_ALT_SETTG_Get_Desc (NU_USB_ALT_SETTG *cb,  
                                NU_USB_INTF_DESC **intf_desc_out)
```

Arguments

- `cb`
Pointer to the alternate setting control block.
- `intf_desc_out`
Pointer to the memory location to hold pointer to the interface descriptor.

Return Values

- `NU_SUCCESS`
Indicates successful completion of the service.

Related Topics

[Nucleus USB Alternate Setting Services](#)

NU_USB_ALT_SETTG_Get_Endp

This function returns the endpoint in an alternate setting with a given endpoint number.

Usage

```
STATUS NU_USB_ALT_SETTG_Get_Endp (NU_USB_ALT_SETTG *cb,
                                UINT8          number_endp,
                                NU_USB_ENDP     **endp_out)
```

Arguments

- **cb**
 Pointer to NU_USB_ALT_SETTG control block.
- **number_endp**
 Endpoint number to be retrieved.
- **endp_out**
 Pointer to the endpoint control block associated with this alternate setting identified by the specified endpoint number.

Return Values

- **NU_SUCCESS**
 Operation Completed Successfully, 'endp_out' contains the address of the endpoint control block when the function returns.
- **NU_USB_INVLD_ARG**
 One or more input arguments are invalid.
- **NU_NOT_PRESENT**
 The required endpoint is not present.

Related Topics

[Nucleus USB Alternate Setting Services](#)

NU_USB_ALT_SETTG_Get_Is_Active

This function returns NU_TRUE in is_active_out, if it is the currently active alternate setting on the interface, or else is_active_out will contain NU_FALSE upon return.

Usage

```
STATUS NU_USB_ALT_SETTG_Get_Is_Active (NU_USB_ALT_SETTG *cb,  
                                       BOOLEAN          *is_active_out)
```

Arguments

- cb
Pointer to the alternate setting control block.
- is_active_out
Pointer to the variable to hold active state of the alternate setting.

Return Values

- NU_SUCCESS
Indicates successful completion of the service.

Related Topics

[Nucleus USB Alternate Setting Services](#)

NU_USB_ALT_SETTG_Get_Num_Endps

This function returns the bNumEndpoints field of the associated interface descriptor, in number_endps_out.

Usage

```
STATUS NU_USB_ALT_SETTG_Get_Num_Endps (
                                         NU_USB_ALT_SETTG *cb,
                                         UINT8           *number_endps_out)
```

Arguments

- **cb**
Pointer to the alternate setting control block.
- **number_endps_out**
Pointer to the variable to hold the bNumEndpoints field of the associated interface descriptor.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Alternate Setting Services](#)

NU_USB_ALT_SETTG_Get_Protocol

This function returns the bInterfaceProtocol field of the associated interface descriptor, in bInterfaceProtocol_out.

Usage

```
STATUS NU_USB_ALT_SETTG_Get_Protocol (  
    NU_USB_ALT_SETTG *cb,  
    UINT8 *bInterfaceProtocol_out)
```

Arguments

- **cb**
Pointer to the alternate setting control block.
- **bInterfaceProtocol_out**
Pointer to the variable to hold the bInterface Protocol field of the associated interface descriptor.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Alternate Setting Services](#)

NU_USB_ALT_SETTG_Get_String

This function copies the interface string of the alternate setting, into string_out.

Usage

```
STATUS NU_USB_ALT_SETTG_Get_String (NU_USB_ALT_SETTG *cb,  
                                   CHAR *string_out)
```

Arguments

- **cb**
Pointer to the alternate setting control block.
- **string_out**
Pointer to the user supplied array of characters of size NU_USB_MAX_STRING_LEN.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Description

English is assumed to be the Unicode of the interface string descriptor. If the Unicode is not English, refer to [NU_USB_ALT_SETTG_Get_String_Desc](#) for further details on how to retrieve a displayable string.

Related Topics

[Nucleus USB Alternate Setting Services](#)

NU_USB_ALT_SETTG_Get_String_Desc

This function gets pointer to the string descriptor (into string_desc_out) for the interface string number (iInterface field of interface descriptor).

Usage

```
STATUS NU_USB_ALT_SETTG_Get_String_Desc (
                                NU_USB_ALT_SETTG    *cb,
                                UINT16               wLangId,
                                NU_USB_STRING_DESC    *string_desc_out)
```

Arguments

- **cb**
Pointer to the alternate setting control block.
- **wLangId**
2-byte Unicode language ID for the string.
- **string_desc_out**
Pointer to the memory location to hold pointer to string descriptor structure.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_NOT_PRESENT**
Indicates that the interface string number cannot be in the device's set of string descriptors.

Description

If the Unicode used for the string descriptors is English, [NU_USB_ALT_SETTG_Get_String](#) can be used to get the ASCII string directly. For other Unicodes the [NU_USB_ALT_SETTG_Get_String_Desc](#) function will help retrieve the descriptor so that the reader can write routines to convert from their Unicode format to ASCII string or such other display string notations.

Related Topics

[Nucleus USB Alternate Setting Services](#)

NU_USB_ALT_SETTG_Get_String_Num

This function returns the iInterface field of the interface descriptor, in string_num_out. If string_num_out is 0, there is no interface string descriptor associated with this alternate setting.

Usage

```
STATUS NU_USB_ALT_SETTG_Get_String_Num (NU_USB_ALT_SETTG *cb,  
                                         UINT8 *string_num_out)
```

Arguments

- **cb**
Pointer to the alternate setting control block.
- **string_num_out**
Pointer to the variable to hold iInterface field of the interface descriptor.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Alternate Setting Services](#)

NU_USB_ALT_SETTG_Get_SubClass

This function returns the bInterfaceSubClass field of the associated interface descriptor in bInterfaceSubClass_out.

Usage

```
STATUS NU_USB_ALT_SETTG_Get_SubClass (  
                                NU_USB_ALT_SETTG *cb,  
                                UINT8             *bInterfaceSubClass_out)
```

Arguments

- **cb**
Pointer to the alternate setting control block.
- **bInterfaceSubClass_out**
Pointer to the variable to hold the bInterfaceSubClass field of the associated interface descriptor.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Alternate Setting Services](#)

NU_USB_ALT_SETTG_Set_Active

This function makes the alternate setting the active alternate setting of the interface.

Usage

```
STATUS NU_USB_ALT_SETTG_Set_Active (NU_USB_ALT_SETTG *cb)
```

Arguments

- **cb**
Pointer to the alternate setting control block.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_USB_NO_BANDWIDTH**
Indicates that the device's bandwidth requirements of the specified configuration cannot be met.
- **NU_INVALID_SEMAPHORE**
Indicates that one of the internal semaphore pointers is invalid.
- **NU_SEMAPHORE_DELETED**
Indicates that one of the internal semaphores was deleted.
- **NU_UNAVAILABLE**
Indicates that one of the internal semaphores is unavailable.
- **NU_INVALID_SUSPEND**
Indicates that this API is called from a non-task thread.
- **NU_USB_INVLD_ARG**
Indicates that one or more arguments passed to this function are invalid.

Description

The resources necessary in the controller hardware are allocated and the required bandwidth on the bus is reserved by this function.

This function should be used only by host class drivers. It is the host's prerogative to select and activate an alternate setting on an interface.

Related Topics

[Nucleus USB Alternate Setting Services](#)

Nucleus USB Binary Object Store (BOS) Services

This section provides a detailed reference of the following Nucleus USB BOS Services:

- [NU_USB_BOS_Get_Num_DevCap](#)
- [NU_USB_BOS_Get_Total_Length](#)
- [NU_USB_DEVCAP_CntnrID_Get_CID](#)
- [NU_USB_DEVCAP_SuprSpd_Get_FS](#)
- [NU_USB_DEVCAP_SuprSpd_Get_Functionality](#)
- [NU_USB_DEVCAP_SuprSpd_Get_HS](#)
- [NU_USB_DEVCAP_SuprSpd_Get_LS](#)
- [NU_USB_DEVCAP_SuprSpd_Get_LTM](#)
- [NU_USB_DEVCAP_SuprSpd_Get_SS](#)
- [NU_USB_DEVCAP_SuprSpd_Get_U1ExitLat](#)
- [NU_USB_DEVCAP_SuprSpd_Get_U2ExitLat](#)
- [NU_USB_DEVCAP_USB2Ext_Get_LPM](#)

Related Topics

[USB Host and Functions](#)

NU_USB_BOS_Get_Num_DevCap

Call this API to get the number of device capability descriptors in a Binary Device Object Store (BOS).

Usage

```
STATUS NU_USB_BOS_Get_Num_DevCap (NU_USB_BOS *cb,
                                  UINT8      *num_devcap_out)
```

Arguments

- **cb**
Pointer to NU_USB_BOS control block.
- **num_devcap_out**
Pointer to the bNumDeviceCaps field of the BOS descriptor when the function returns.

Return Values

- **NU_SUCCESS**
Operation completed successfully, 'num_devcap_out' contains the total number of device capability descriptors in BOS pointed by 'cb'.
- **NU_INVLD_ARG**
One or more input arguments are invalid.
- **NU_NOT_PRESENT**
A BOS descriptor is not present.

Related Topics

[Nucleus USB Binary Object Store \(BOS\) Services](#)

NU_USB_BOS_Get_Total_Length

Call this API to get the total length of the Binary Device Object Store (BOS) descriptor. This includes the length of the BOS descriptor and all of its sub descriptors (device capability descriptors).

Usage

```
STATUS NU_USB_BOS_Get_Total_Length (NU_USB_BOS *cb,  
                                   UINT16      *totallength_out)
```

Arguments

- **cb**
Pointer to NU_USB_BOS control block.
- **totallength_out**
Pointer to the wTotalLength field of the BOS descriptor when the function returns.

Return Values

- **NU_SUCCESS**
Operation completed successfully, 'totallength_out' contains the total length of the BOS descriptor pointed by 'cb'.
- **NU_INVLD_ARG**
One or more input arguments are invalid.
- **NU_NOT_PRESENT**
A BOS descriptor is not present.

Related Topics

[Nucleus USB Binary Object Store \(BOS\) Services](#)

NU_USB_DEVCAP_CntnrID_Get_CID

Call this API to get the Container ID field of the Container ID device capability descriptor supported by a USB 3.0 device.

Usage

```
STATUS NU_USB_DEVCAP_CntnrID_Get_CID (NU_USB_BOS *cb,
                                     UINT8      **containerid_out)
```

Arguments

- **cb**
Pointer to NU_USB_BOS control block.
- **containerid_out**
Double pointer to an array that contains 16 byte values of the Container ID field of the Container ID device capability descriptor.

Return Values

- **NU_SUCCESS**
Operation completed successfully, 'containerid_out' contains value of the Container ID field of the Container ID device capability descriptor.
- **NU_INVLD_ARG**
One or more input arguments are invalid.
- **NU_NOT_PRESENT**
Container ID device capability descriptor is not present.

Related Topics

[Nucleus USB Binary Object Store \(BOS\) Services](#)

NU_USB_DEVCAP_SuprSpd_Get_FS

Call this API to check if Full Speed is supported by a USB 3.0 device.

Usage

```
STATUS NU_USB_DEVCAP_SuprSpd_Get_FS (NU_USB_BOS *cb,  
                                     BOOLEAN *fs_out)
```

Arguments

- **cb**
Pointer to NU_USB_BOS control block.
- **fs_out**
Pointer to a BOOLEAN. This contains the value of Bit 1 (Full Speed Supported) in wSpeedsSupported field of Super Speed device capability descriptor when the function returns. This can be set to one of the following values:

NU_TRUE

Bit 1 is set

NU_FALSE

Bit 1 is not set

Return Values

- **NU_SUCCESS**
Operation completed successfully, 'fs_out' contains value of Bit 1 of wSpeedsSupported field of SuperSpeed device capability descriptor.
- **NU_INVLD_ARG**
One or more input arguments are invalid.
- **NU_NOT_PRESENT**
SuperSpeed device capability descriptor is not present.

Related Topics

[Nucleus USB Binary Object Store \(BOS\) Services](#)

NU_USB_DEVCAP_SuprSpd_Get_Functionality

Call this API to get the lowest speed at which functionality is supported by a USB 3.0 device.

Usage

```
STATUS NU_USB_DEVCAP_SuprSpd_Get_Functionality (NU_USB_BOS *cb,  
                                                UINT8      *func_out)
```

Arguments

- **cb**
Pointer to NU_USB_BOS control block.
- **func_out**
Pointer to bFunctionalitySupport bit field of the SuperSpeed device capability descriptor.
 - 0: Low Speed
 - 1: Full Speed
 - 2: High Speed
 - 3: Super Speed

Return Values

- **NU_SUCCESS**
Operation completed successfully, 'func_out' contains value of bFunctionalitySupport bit field of the SuperSpeed device capability descriptor.
- **NU_INVLD_ARG**
One or more input arguments are invalid.
- **NU_NOT_PRESENT**
SuperSpeed device capability descriptor is not present.

Related Topics

[Nucleus USB Binary Object Store \(BOS\) Services](#)

NU_USB_DEVCAP_SuprSpd_Get_HS

Call this API to check if High Speed is supported by a USB 3.0 device.

Usage

```
STATUS NU_USB_DEVCAP_SuprSpd_Get_HS (NU_USB_BOS *cb,  
                                     BOOLEAN *hs_out)
```

Arguments

- **cb**
Pointer to NU_USB_BOS control block
- **hs_out**
Pointer to BOOLEAN. This contains the value of Bit 2 (High Speed Supported) in wSpeedsSupported field of Super Speed device capability descriptor when this function returns. This can be set to one of the following values:

NU_TRUE

Bit 2 is set

NU_FALSE

Bit 2 is not set

Return Values

- **NU_SUCCESS**
Operation completed successfully, 'hs_out' contains the value of Bit 2 of wSpeedsSupported bit field of Super Speed device capability descriptor.
- **NU_INVLD_ARG**
One or more input arguments are invalid.
- **NU_NOT_PRESENT**
SuperSpeed device capability descriptor is not present.

Related Topics

[Nucleus USB Binary Object Store \(BOS\) Services](#)

NU_USB_DEVCAP_SuprSpd_Get_LS

Call this API to check if Low Speed is supported by a USB 3.0 device.

Usage

```
STATUS NU_USB_DEVCAP_SuprSpd_Get_LS (NU_USB_BOS *cb,  
                                     BOOLEAN *ls_out)
```

Arguments

- **cb**
Pointer to NU_USB_BOS control block
- **ls_out**
Pointer to BOOLEAN. This contains the value of Bit 0 (Low Speed Supported) in wSpeedsSupported field of Super Speed device capability descriptor when this function returns. This can be set to one of the following values:

NU_TRUE

Bit 0 is set

NU_FALSE

Bit 0 is not set

Return Values

- **NU_SUCCESS**
Operation completed successfully, 'ls_out' contains value of Bit 0 of wSpeedsSupported bit field of Super Speed device capability descriptor.
- **NU_INVLD_ARG**
One or more input arguments are invalid.
- **NU_NOT_PRESENT**
SuperSpeed device capability descriptor is not present.

Related Topics

[Nucleus USB Binary Object Store \(BOS\) Services](#)

NU_USB_DEVCAP_SuprSpd_Get_LTM

Call this API to get the value of the LTM bit in the bmAttribute field of the Super Speed device capability descriptor.

Usage

```
STATUS NU_USB_DEVCAP_SuprSpd_Get_LTM (NU_USB_BOS *cb,  
                                      BOOLEAN *ltm_out)
```

Arguments

- **cb**
Pointer to NU_USB_BOS control block.
- **ltm_out**
Pointer to BOOLEAN. This contains the value of the LTM bit in the bmAttribute field of the Super Speed device capability descriptor when this function returns. This can be set to one of the following values:

NU_TRUE
LTM Bit is set
NU_FALSE
LTM Bit is not set

Return Values

- **NU_SUCCESS**
Operation completed successfully, 'ltm_out' contains the value of the LTM bit of the bmAttribute field of the Super Speed device capability descriptor.
- **NU_INVLD_ARG**
One or more input arguments are invalid.
- **NU_NOT_PRESENT**
SuperSpeed device capability descriptor is not present.

Related Topics

[Nucleus USB Binary Object Store \(BOS\) Services](#)

NU_USB_DEVCAP_SuprSpd_Get_SS

Call this API to check if Super Speed is supported by a USB 3.0 device.

Usage

```
STATUS NU_USB_DEVCAP_SuprSpd_Get_SS (NU_USB_BOS *cb,  
                                     BOOLEAN *ss_out)
```

Arguments

- **cb**
Pointer to NU_USB_BOS control block.
- **ss_out**
Pointer to BOOLEAN. This contains the value of bit 3 (Super Speed Supported) of wSpeedsSupported field of Super Speed device capability descriptor when the function returns. This can be set to one of the following values:

NU_TRUE

Bit 3 is set

NU_FALSE

Bit 3 is not set

Return Values

- **NU_SUCCESS**
Operation completed successfully, 'ss_out' contains value of Bit 3 of wSpeedsSupported field of the Super Speed device capability descriptor.
- **NU_INVLD_ARG**
One or more input arguments are invalid.
- **NU_NOT_PRESENT**
SuperSpeed device capability descriptor is not present.

Related Topics

[Nucleus USB Binary Object Store \(BOS\) Services](#)

NU_USB_DEVCAP_SuprSpd_Get_U1ExitLat

Call this API to get the U1 exit latency field supported by a USB 3.0 device.

Usage

```
STATUS NU_USB_DEVCAP_SuprSpd_Get_U1ExitLat (NU_USB_BOS *cb,  
                                             UINT8      *u1exitlat_out)
```

Arguments

- **cb**
Pointer to NU_USB_BOS control block
- **u1exitlat_out**
Pointer to the value of bU1ExitLat field of Super Speed device capability descriptor.

Return Values

- **NU_SUCCESS**
Operation completed successfully, 'u1exitlat_out' contains the value of the bU1ExitLat field of the Super Speed device capability descriptor.
- **NU_INVLD_ARG**
One or more input arguments are invalid.
- **NU_NOT_PRESENT**
SuperSpeed device capability descriptor is not present.

Related Topics

[Nucleus USB Binary Object Store \(BOS\) Services](#)

NU_USB_DEVCAP_SuprSpd_Get_U2ExitLat

Call this API to get the U2 exit latency field supported by a USB 3.0 device.

Usage

```
STATUS NU_USB_DEVCAP_SuprSpd_Get_U2ExitLat (NU_USB_BOS *cb,
                                             UINT8      *u2exitlat_out)
```

Arguments

- **cb**
Pointer to NU_USB_BOS control block.
- **u2exitlat_out**
Pointer to the value of bU2ExitLat field of Super Speed device capability descriptor.

Return Values

- **NU_SUCCESS**
Operation completed successfully, 'u2exitlat_out' contains the value of the bU2ExitLat field of the Super Speed device capability descriptor.
- **NU_INVLD_ARG**
One or more input arguments are invalid.
- **NU_NOT_PRESENT**
SuperSpeed device capability descriptor is not present.

Related Topics

[Nucleus USB Binary Object Store \(BOS\) Services](#)

NU_USB_DEVCAP_USB2Ext_Get_LPM

Call this API to get the value of the LPM bit in the bmAttribute field of the USB 2 extension device capability descriptor.

Usage

```
STATUS NU_USB_DEVCAP_USB2Ext_Get_LPM (NU_USB_BOS *cb,  
                                      BOOLEAN *lpm_out)
```

Arguments

- **cb**
Pointer to NU_USB_BOS control block.
- **lpm_out**
Pointer to the value of the LPM bit in the bmAttribute field of the USB 2 extension descriptor when the function returns. This can be set to one of the following values:

NU_TRUE

LMP Bit is set

NU_FALSE

LMP Bit is not set

Return Values

- **NU_SUCCESS**
Operation completed successfully, 'lpm_out' contains value of the LPM bit of bmAttribute field of USB 2 extension device capability descriptor.
- **NU_INVLD_ARG**
One or more input arguments are invalid.
- **NU_NOT_PRESENT**
If a USB2.0 extension device capability descriptor is not present.

Related Topics

[Nucleus USB Binary Object Store \(BOS\) Services](#)

Nucleus USB Config Services

This section provides a detailed reference of the following Nucleus USB Config Services:

- [NU_USB_CFG_Find_Alt_Setting](#)
- [NU_USB_CFG_Get_Cfg_Value](#)
- [NU_USB_CFG_Get_Desc](#)
- [NU_USB_CFG_Get_Device](#)
- [NU_USB_CFG_Get_IAD](#)
- [NU_USB_CFG_Get_Intf](#)
- [NU_USB_CFG_Get_Is_Active](#)
- [NU_USB_CFG_Get_Is_Self_Powered](#)
- [NU_USB_CFG_Get_Is_Wakeup](#)
- [NU_USB_CFG_Get_Max_Power](#)
- [NU_USB_CFG_Get_Num_IADs](#)
- [NU_USB_CFG_Get_Num_Intfs](#)
- [NU_USB_CFG_Get_String](#)
- [NU_USB_CFG_Get_String_Desc](#)
- [NU_USB_CFG_Get_String_Num](#)
- [NU_USB_CFG_Get_wTotalLength](#)
- [NU_USB_CFG_Set_Is_Active](#)

Related Topics

[USB Host and Functions](#)

NU_USB_CFG_Find_Alt_Setting

This function finds a matching alternate setting among various alternate settings of all the interfaces.

Usage

```
STATUS NU_USB_CFG_Find_Alt_Setting (NU_USB_CFG      *cb,  
                                   UINT32           match_flag,  
                                   UINT8            intf_num,  
                                   UINT8            alt_settg,  
                                   UINT8            bInterfaceClass,  
                                   UINT8            bInterfaceSubClass,  
                                   UINT8            bInterfaceProtocol,  
                                   NU_USB_ALT_SETTG **alt_settg_out)
```

Arguments

- **cb**
Pointer to the configuration control block. The pointer to the control block of the matching alternate setting is returned in `alt_settg_out`.
- **match_flag**
The `match_flag` specifies the search criteria. You can OR one or more of the following flags:
 - USB_MATCH_ONLY_ACTIVE_ALT_STTG**
Match only against the active alternate setting. Do not consider non-active alternate settings for a match.
 - USB_MATCH_CLASS**
Search for the alternate setting whose `bInterfaceClass` equals the specified `bInterfaceClass`.
 - USB_MATCH_SUB_CLASS**
Search for the alternate setting whose `bInterfaceSubClass` equals the specified `bInterfaceSubClass`. `match_flag` must contain `USB_MATCH_CLASS`.
 - USB_MATCH_PROTOCOL**
Search for the alternate setting whose `bInterfaceProtocol` equals the specified `bInterfaceProtocol`. `match_flag` must also contain `USB_MATCH_SUB_CLASS`.
- **intf_num**
Interface number from which the search should begin.
- **alt_settg**
Alternate setting number of the specified interface from which the search should begin.
- **bInterfaceClass**
The desired value of the `bInterfaceClass` in the matching alternate setting.

- **bInterfaceSubClass**
The desired value of the bInterfaceSubClass in the matching alternate setting. Irrelevant, if match_flag does not contain USB_MATCH_SUB_CLASS.
- **bInterfaceProtocol**
The desired value of the bInterfaceProtocol in the matching alternate setting. Irrelevant, if match_flag does not contain USB_MATCH_PROTOCOL.
- **alt_settg_out**
Pointer to a memory location to hold the pointer to the control block of the matching alternate setting.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_NOT_PRESENT**
Indicates that no matching alternate setting could be found.
- **NU_USB_INVLD_ARG**
Indicates that the control block is invalid.

Related Topics

[Nucleus USB Config Services](#)

NU_USB_CFG_Get_Cfg_Value

This function returns the bConfigurationValue field of the associated configuration descriptor in `cfg_value_out`.

Usage

```
STATUS NU_USB_CFG_Get_Cfg_Value (NU_USB_CFG *cb,  
                                UINT8      *cfg_value_out)
```

Arguments

- `cb`
Pointer to the configuration control block.
- `cfg_value_out`
Pointer to the variable to hold the bConfigurationValue.

Return Values

- `NU_SUCCESS`
Indicates successful completion of the service.

Related Topics

[Nucleus USB Config Services](#)

NU_USB_CFG_Get_Desc

This function returns pointer to configuration descriptor (in `cfg_desc_out`) associated with the specified configuration control block.

Usage

```
STATUS NU_USB_CFG_Get_Desc (NU_USB_CFG          *cb,
                           NU_USB_CFG_DESC **cfg_desc_out)
```

Arguments

- `cb`
 Pointer to the configuration control block.
- `cfg_desc_out`
 Pointer to a memory location to hold the pointer to the configuration descriptor structure.

Return Values

- `NU_SUCCESS`
 Indicates successful completion of the service.

Related Topics

[Nucleus USB Config Services](#)

NU_USB_CFG_Get_Device

This function returns the pointer to the control block of the associated device in device_out.

Usage

```
STATUS NU_USB_CFG_Get_Device (NU_USB_CFG      *cb,  
                             NU_USB_DEVICE **device_out)
```

Arguments

- **cb**
Pointer to the device control block.
- **device_out**
Pointer to the memory location to hold pointer to the device control block.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Config Services](#)

NU_USB_CFG_Get_IAD

This function returns pointer to an Interface Association Descriptor (IAD) with specified interface number.

Usage

```
STATUS NU_USB_CFG_Get_IAD (NU_USB_CFG *cb,  
                           UINT8      intf_num,  
                           NU_USB_IAD **iad_out)
```

Arguments

- **cb**
Pointer to NU_USB_CFG control block.
- **intf_num**
Interface number of associated interface.
- **iad_out**
Pointer to required IAD control block.

Return Values

NU_SUCCESS

Operation Completed Successfully, 'iad_out' contains a pointer to the IAD control block when the function returns.

- **NU_USB_INVLD_ARG**
No valid input arguments.

Related Topics

[Nucleus USB Config Services](#)

NU_USB_CFG_Get_Intf

This function returns the pointer (in intf_out) to the control block of the interface associated with the specified interface.

Usage

```
STATUS NU_USB_CFG_Get_Intf (NU_USB_CFG *cb,  
                           UINT8      intf_num,  
                           NU_USB_INTF **intf_out)
```

Arguments

- **cb**
Pointer to the configuration control block.
- **intf_num**
Interface number whose interface control block pointer is desired.
- **intf_out**
Pointer to a memory location to hold pointer to the interface control block.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Config Services](#)

NU_USB_CFG_Get_Is_Active

This function returns NU_TRUE in is_active_out, if it is the currently active configuration of the device, or else is_active_out will contain NU_FALSE on return.

Usage

```
STATUS NU_USB_CFG_Get_Is_Active (NU_USB_CFG *cb,  
                                BOOLEAN      *is_active_out)
```

Arguments

- cb
Pointer to the configuration control block.
- is_active_out
Pointer to the variable to hold active state of the configuration.

Return Values

- NU_SUCCESS
Indicates successful completion of the service.

Related Topics

[Nucleus USB Config Services](#)

NU_USB_CFG_Get_Is_Self_Powered

This function returns NU_TRUE in is_self_powered_out if the bmAttributes field of the associated configuration descriptor indicates it is self-powered and NU_FALSE otherwise.

Usage

```
STATUS NU_USB_CFG_Get_Is_Self_Powered (NU_USB_CFG *cb,  
                                       BOOLEAN      *is_self_powered_out)
```

Arguments

- cb
Pointer to the configuration control block.
- is_self_powered_out
Pointer to the variable to hold the self powered nature of the configuration.

Return Values

- NU_SUCCESS
Indicates successful completion of the service.

Related Topics

[Nucleus USB Config Services](#)

NU_USB_CFG_Get_Is_Wakeup

This function returns NU_TRUE in is_wakeup_out, if the bmAttributes field of the associated configuration descriptor indicates that it is capable of waking up and NU_FALSE otherwise.

Usage

```
STATUS NU_USB_CFG_Get_Is_Wakeup (NU_USB_CFG *cb,  
                                BOOLEAN      *is_wakeup_out)
```

Arguments

- cb
Pointer to the configuration control block.
- is_wakeup_out
Pointer to the variable to hold the wakeup capability of the configuration.

Return Values

- NU_SUCCESS
Indicates successful completion of the service.

Related Topics

[Nucleus USB Config Services](#)

NU_USB_CFG_Get_Max_Power

This function returns the bMaxPower field of the associated configuration descriptor in max_power_out.

Usage

```
STATUS NU_USB_CFG_Get_Max_Power (NU_USB_CFG *cb,  
                                UINT8      *max_power_out)
```

Arguments

- cb
Pointer to the configuration control block.
- max_power_out
Pointer to the variable to hold the bMaxPower.

Return Values

- NU_SUCCESS
Indicates successful completion of the service.

Related Topics

[Nucleus USB Config Services](#)

NU_USB_CFG_Get_Num_IADs

This function returns the number of Interface Associate Descriptors (IADs) in a particular configuration.

Usage

```
STATUS NU_USB_CFG_Get_Num_IADs (NU_USB_CFG *cb,
                                UINT8      *number_iads_out)
```

Arguments

- **cb**
Pointer to NU_USB_CFG control block.
- **number_iads_out**
Number of IAD present in this configuration.

Return Values

- **NU_SUCCESS**
Operation Completed Successfully, 'number_iad_out' contains the number of IADs present in this configuration.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Config Services](#)

NU_USB_CFG_Get_Num_Intfs

This function returns the bNumInterfaces field of the associated configuration descriptor in number_intfs_out.

Usage

```
STATUS NU_USB_CFG_Get_Num_Intfs (NU_USB_CFG *cb,  
                                UINT8      *number_intfs_out)
```

Arguments

- cb
Pointer to the configuration control block.
- number_intfs_out
Pointer to the variable to hold bNumInterfaces field.

Return Values

- NU_SUCCESS
Indicates successful completion of the service.

Related Topics

[Nucleus USB Config Services](#)

NU_USB_CFG_Get_String

This function copies the configuration string into string_out.

Usage

```
STATUS NU_USB_CFG_Get_String (NU_USB_CFG *cb,  
                             CHAR      *string_out)
```

Arguments

- **cb**
 Pointer to the configuration control block.
- **string_out**
 Pointer to the user supplied array of characters of size NU_USB_MAX_STRING_LEN.

Return Values

- **NU_SUCCESS**
 Indicates successful completion of the service.
- **NU_NOT_PRESENT**
 Indicates that the configuration string number cannot be found in the device's set of string descriptors.

Description

English is assumed to be the Unicode of the configuration string descriptor. If the Unicode is not English, refer to [NU_USB_CFG_Get_String_Desc](#) for further information on how to retrieve a displayable string.

Related Topics

[Nucleus USB Config Services](#)

NU_USB_CFG_Get_String_Desc

This function gets the pointer to the string descriptor (into string_desc_out) for the configuration string number (iConfiguration field of configuration descriptor).

Usage

```
STATUS NU_USB_CFG_Get_String_Desc (NU_USB_CFG          *cb,  
                                  UINT16                wLangId,  
                                  NU_USB_STRING_DESC     *string_desc_out)
```

Arguments

- **cb**
Pointer to the configuration control block.
- **wLangId**
2-byte Unicode language ID for the string.
- **string_desc_out**
Pointer to the memory location to hold pointer to string descriptor structure.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_NOT_PRESENT**
Indicates that the configuration string number cannot be found in the device's set of string descriptors.

Description

If the Unicode used for the string descriptors is English, [NU_USB_CFG_Get_String](#) can be used to get the ASCII string directly. For other Unicodes the `NU_USB_CFG_Get_String_Desc` function will help retrieve the descriptor and then the reader may write routines to convert from their Unicode format to ASCII string or such other display string notations.

Related Topics

[Nucleus USB Config Services](#)

NU_USB_CFG_Get_String_Num

This function returns the iConfiguration field of the configuration descriptor in string_num_out. If string_num_out is 0, it there is no configuration string descriptor associated with this configuration.

Usage

```
STATUS NU_USB_CFG_Get_String_Num (NU_USB_CFG *cb,
                                UINT8      *string_num_out)
```

Arguments

- **cb**
Pointer to the configuration control block.
- **string_num_out**
Pointer to the variable to hold iConfiguration field of the configuration descriptor.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Config Services](#)

NU_USB_CFG_Get_wTotalLength

This function returns the wTotalLength field of the associated configuration descriptor in wtotalLength_out.

Usage

```
STATUS NU_USB_CFG_Get_wTotalLength (NU_USB_CFG *cb,  
                                   UINT8      *wTotalLength_out)
```

Arguments

- cb
Pointer to the configuration control block.
- wTotalLength_out
Pointer to the variable to hold the wTotalLength.

Return Values

- NU_SUCCESS
Indicates successful completion of the service.

Related Topics

[Nucleus USB Config Services](#)

NU_USB_CFG_Set_Is_Active

This function makes the configuration either active or inactive depending on the state of `is_active`. See this function's Description section for details.

Usage

```
STATUS NU_USB_CFG_Set_Is_Active (NU_USB_CFG *cb,
                                BOOLEAN      is_active)
```

Arguments

- `cb`
 Pointer to the configuration control block that is to activated/deactivated.
- `is_active`
 Set this to `NU_TRUE` to activate the configuration, otherwise set this to `NU_FALSE` to make the configuration inactive.

Return Values

- `NU_SUCCESS`
 Indicates successful completion of the service.
- `NU_USB_NO_BANDWIDTH`
 Indicates that the device's bandwidth requirements of the specified configuration cannot be met.
- `NU_INVALID_SEMAPHORE`
 Indicates that one of the internal semaphore pointers is corrupted.
- `NU_SEMAPHORE_DELETED`
 Indicates that one of the internal semaphores was deleted.
- `NU_UNAVAILABLE`
 Indicates that one of the internal semaphores is unavailable.
- `NU_INVALID_SUSPEND`
 Indicates that this API is called from a non-task thread.
- `NU_USB_INVLD_ARG`
 Indicates that the device control block or configuration control block passed to this function is invalid.

Description

If `is_active` is `NU_TRUE`, this function sets the configuration as active on the device. The Nucleus USB bandwidth and necessary resources in the controller hardware are allocated to make the configuration active.

If `is_active` is `NU_FALSE`, and if the configuration is currently active, this function makes the current configuration inactive and places the device in an unconfigured state.

Nucleus USB interface drivers are not expected to make this call since they only own an interface on the device and only Nucleus USB device drivers are entitled to make this call.

Only host class drivers should use this function. It is the host's prerogative to select and activate/deactivate a configuration on the device.

Related Topics

[Nucleus USB Config Services](#)

Nucleus USB Device Services

This section provides a detailed reference of the following Nucleus USB Device Services:

- [NU_USB_DEVICE_Claim](#)
- [NU_USB_DEVICE_Get_Active_Cfg](#)
- [NU_USB_DEVICE_Get_Active_Cfg_Num](#)
- [NU_USB_DEVICE_Get_bcdDevice](#)
- [NU_USB_DEVICE_Get_bcdUSB](#)
- [NU_USB_DEVICE_Get_bDeviceClass](#)
- [NU_USB_DEVICE_Get_bDeviceProtocol](#)
- [NU_USB_DEVICE_Get_bDeviceSubClass](#)
- [NU_USB_DEVICE_Get_bMaxPacketSize0](#)
- [NU_USB_DEVICE_Get_BOS](#)
- [NU_USB_DEVICE_Get_BOS_Desc](#)
- [NU_USB_DEVICE_Get_Cfg](#)
- [NU_USB_DEVICE_Get_CntnrID_Desc](#)
- [NU_USB_DEVICE_Get_Current_Requirement](#)
- [NU_USB_DEVICE_Get_Desc](#)
- [NU_USB_DEVICE_Get_Function_Addr](#)
- [NU_USB_DEVICE_Get_Hw](#)
- [NU_USB_DEVICE_Get_idProduct](#)
- [NU_USB_DEVICE_Get_idVendor](#)
- [NU_USB_DEVICE_Get_Is_Claimed](#)
- [NU_USB_DEVICE_Get_Manf_String](#)
- [NU_USB_DEVICE_Get_Manf_String_Desc](#)
- [NU_USB_DEVICE_Get_Manf_String_Num](#)
- [NU_USB_DEVICE_Get_Num_Cfgs](#)
- [NU_USB_DEVICE_Get_OTG_Desc](#)
- [NU_USB_DEVICE_Get_OTG_Status](#)

- `NU_USB_DEVICE_Get_Parent`
- `NU_USB_DEVICE_Get_Port_Number`
- `NU_USB_DEVICE_Get_Product_String`
- `NU_USB_DEVICE_Get_Product_String_Desc`
- `NU_USB_DEVICE_Get_Product_String_Num`
- `NU_USB_DEVICE_Get_Serial_Num_String`
- `NU_USB_DEVICE_Get_Serial_Num_String_Desc`
- `NU_USB_DEVICE_Get_Serial_Num_String_Num`
- `NU_USB_DEVICE_Get_Speed`
- `NU_USB_DEVICE_Get_Stack`
- `NU_USB_DEVICE_Get_Status`
- `NU_USB_DEVICE_Get_String`
- `NU_USB_DEVICE_Get_String_Desc`
- `NU_USB_DEVICE_Get_SuprSpd_Desc`
- `NU_USB_DEVICE_Get_USB2Ext_Desc`
- `NU_USB_DEVICE_Release`
- `NU_USB_DEVICE_Set_Active_Cfg`
- `NU_USB_DEVICE_Set_bcdUSB`
- `NU_USB_DEVICE_Set_bDeviceClass`
- `NU_USB_DEVICE_Set_bMaxPacketSize0`
- `NU_USB_DEVICE_Set_Desc`
- `NU_USB_DEVICE_Set_Device_Qualifier`
- `NU_USB_DEVICE_Set_Hw`
- `NU_USB_DEVICE_Set_Link_State`
- `NU_USB_DEVICE_Set_Manf_String`
- `NU_USB_DEVICE_Set_Product_String`
- `NU_USB_DEVICE_Set_Stack`
- `NU_USB_DEVICE_Set_Status`

- [NU_USB_DEVICE_Set_String](#)
- [NU_USB_DEVICE_Set_Serial_Num_String](#)

Related Topics

[USB Host and Functions](#)

NU_USB_DEVICE_Claim

This function sets the specified driver as the driver owning the device.

Usage

```
STATUS NU_USB_DEVICE_Claim (NU_USB_DEVICE *cb,  
                           NU_USB_DRV   *drv)
```

Arguments

- **cb**
Pointer to the device control block.

Note



This pointer is passed to the USB driver when the stack invokes USB driver's `NU_USB_DRV_Initialize_Device` function.

- **drv**
Pointer to the control block of the driver that has decided to own the device.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Description

A Nucleus USB device class driver may call this function from its `_NU_USB_DRV_Initialize_Device` once it decides to own the device. Drivers usually decide to take ownership of the device after successfully locating all necessary pipes and notifying relevant user drivers.

Note



Nucleus USB interface class drivers are not expected to make this function call. They are entitled to own only interfaces, not devices.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_Active_Cfg

This function returns a pointer to the control block of the currently active configuration on the device in `cfg_out`.

Usage

```
STATUS NU_USB_DEVICE_Get_Active_Cfg (NU_USB_DEVICE *cb,  
                                     NU_USB_CFG     **cfg_out)
```

Arguments

- `cb`
Pointer to `NU_USB_DEVICE` control block.
- `cfg_out`
Output argument containing the pointer to the current active configuration.

Return Values

- `NU_SUCCESS`
Indicates successful completion of the service, 'cfg_out' contains a pointer to the current active configuration.
- `NU_USB_INVLD_ARG`
One or more input arguments are invalid.
- `NU_NOT_PRESENT`
Indicates that an active configuration is not present on the device, 'cfg_out' is set to `NU_NULL` in this case.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_Active_Cfg_Num

This function returns the bConfigurationValue of the configuration (in active_cfg_num_out) that is currently active on the device. If the device is not in the onfigured state, 0 is returned in active_cfg_num_out.

Usage

```
STATUS NU_USB_DEVICE_Get_Active_Cfg_Num (
                                         NU_USB_DEVICE *cb,
                                         UINT8        *active_cfg_num_out)
```

Arguments

- cb
Pointer to the device control block.
- active_cfg_num_out
Pointer to a variable to hold the active configuration number.

Return Values

- NU_SUCCESS
Indicates successful completion of the service.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_bcdDevice

This function returns the release number in BCD form (bcdDevice field of the device descriptor), in bcdDevice_out.

Usage

```
STATUS NU_USB_DEVICE_Get_bcdDevice (NU_USB_DEVICE *cb,  
                                   UINT16          *bcdDevice_out)
```

Arguments

- cb
Pointer to the device control block.
- bcdDevice_out
Pointer to the variable to hold the device release number.

Return Values

- NU_SUCCESS
Indicates successful completion of the service.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_bcdUSB

This function returns the Nucleus USB version supported by the device in BCD format (field bcdUSB of device descriptor).

Usage

```
STATUS NU_USB_DEVICE_Get_bcdUSB (NU_USB_DEVICE *cb,  
                                UINT16          *bcdUSB_out)
```

Arguments

- **cb**
Pointer to the device control block.
- **bcdUSB_out**
Pointer to the variable to hold the BCD version of the Nucleus USB supported by the device.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_bDeviceClass

This function returns the class code of the device (bDeviceClass field of the device descriptor), in bDeviceClass_out.

Usage

```
STATUS NU_USB_DEVICE_Get_bDeviceClass (NU_USB_DEVICE *cb,  
                                       UINT8          *bDeviceClass_out)
```

Arguments

- cb
Pointer to the device control block.
- bDeviceClass_out
Pointer to the variable to hold the class code of the device.

Return Values

- NU_SUCCESS
Indicates successful completion of the service.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_bDeviceProtocol

This function returns the class code of the device (bDeviceProtocol field of the device descriptor), in bDeviceProtocol_out.

Usage

```
STATUS NU_USB_DEVICE_Get_bDeviceProtocol (  
                                         NU_USB_DEVICE *cb,  
                                         UINT8        *bDeviceProtocol_out)
```

Arguments

- **cb**
Pointer to the device control block.
- **bDeviceProtocol_out**
Pointer to the variable to hold the protocol code of the device.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_bDeviceSubClass

This function returns the sub class code of the device (bDeviceSubClass field of the device descriptor), in bDeviceSubClass_out.

Usage

```
STATUS NU_USB_DEVICE_Get_bDeviceSubClass (
                                         NU_USB_DEVICE *cb,
                                         UINT8        *bDeviceSubClass_out)
```

Arguments

- **cb**
Pointer to the device control block.
- **bDeviceSubClass_out**
Pointer to the variable to hold the sub class code of the device.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_bMaxPacketSize0

This function returns the maximum packet size supported by the default control endpoint of the device.

Usage

```
STATUS NU_USB_DEVICE_Get_bMaxPacketSize0 (NU_USB_DEVICE *cb,  
                                           #if ( CFG_NU_OS_CONN_USB_COM_STACK_SS_ENABLE == NU_TRUE )  
                                           UINT16 *bMaxPacketSize0_out)  
                                           #else  
                                           UINT8  *bMaxPacketSize0_out)  
                                           #endif
```

Arguments

- **cb**
Pointer to the NU_USB_DEVICE control block.
- **bMaxPacketSize0_out**
Pointer to the variable to hold the maximum packet size of endpoint 0.
(UINT8 - without Super Speed support) or (UINT16 - with Super Speed support) variable containing the maximum packet size of the default control endpoint when the function returns.

Return Values

- **NU_SUCCESS**
Operation Completed Successfully, 'bMaxPacketSize0_out' contains maximum packet size of the default control endpoint.
- **NU_USB_INVLD_DESC**
Device descriptor is invalid.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Description

In USB 2.0 and earlier standards, devices support up to 64 bytes of the maximum packet size of the default control endpoint. According to the USB 3.0 standard, the maximum packet size of the default control endpoint of a Super Speed device must be 512 bytes. This API has a dual signature which depends on the current configuration of the USB stack. If the USB stack is built without Super Speed support, then the Maxp of endpoint 0 is returned as a UINT8 variable; otherwise, it is returned as a UINT16 variable.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_BOS

Call this function to get a pointer to the NU_USB_BOS control block. A difference must be observed between this API and the [NU_USB_DEVICE_Get_BOS_Desc](#) API which returns a pointer to the BOS descriptor.

Usage

```
STATUS NU_USB_DEVICE_Get_BOS (NU_USB_DEVICE *cb,
                             NU_USB_BOS    **bos_out)
```

Arguments

- **cb**
A pointer to NU_USB_DEVICE control block.
- **bos_out**
Double pointer to the NU_USB_BOS control block. This points to a valid NU_USB_BOS control block or NU_NULL when the function returns.

Return Values

- **NU_SUCCESS**
Operation Completed Successfully, 'bos_out' points to a valid BOS control block.
- **NU_NOT_PRESENT**
A BOS descriptor is not present.
- **NU_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_BOS_Desc

This API is used for extracting the BOS descriptor from the NU_USB_DEVICE control block. This will only get the BOS descriptor not the device capability descriptor. Separate APIs are provided to get the device capability descriptors.

Usage

```
STATUS NU_USB_DEVICE_Get_BOS_Desc (NU_USB_DEVICE *cb,  
                                  NU_USB_BOS_DESC **bos_desc_out)
```

Arguments

- **cb**
A pointer to NU_USB_DEVICE control block.
- **bos_desc_out**
Double pointer to NU_USB_BOS_DESC control block. This points to a valid NU_USB_BOS_DESC control block or NU_NULL when the function returns.

Return Values

- **NU_SUCCESS**
Operation Completed Successfully, 'bos_out' points to a valid BOS descriptor.
- **NU_USB_INVLD_DESC**
The descriptor type of BOS descriptor is not USB_DT_BOS.
- **NU_NOT_PRESENT**
A BOS descriptor is not present.
- **NU_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_Cfg

This function returns pointer to the configuration control block (in `cfg_out`) for the specified `bConfiguration` value.

Usage

```
STATUS NU_USB_DEVICE_Get_Cfg (NU_USB_DEVICE *cb,
                              UINT8         cfg_num,
                              NU_USB_CFG    **cfg_out)
```

Arguments

- `cb`
Pointer to the device control block.
- `cfg_num`
Configuration number.
- `cfg_out`
Pointer to a memory location to hold pointer to the control block of the requested configuration.

Return Values

- `NU_SUCCESS`
Indicates successful completion of the service.
- `NU_NOT_PRESENT`
Indicates that the requested `bConfigurationValue` is not present in the device or if 0 is specified for `cfg_num`, it means that the device is not in the configured state.

Description

If 0 is specified for `cfg_num`, then the pointer to the control block of the currently active configuration is returned in `cfg_out`.

Functions defined in the "[Nucleus USB Config Services](#)" can be used to operate upon the returned configuration control block.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_CntnrID_Desc

This API is used for extracting the Container ID device capability descriptor from the NU_USB_DEVICE control block. According to USB 3.0 specifications, this descriptor is only mandatory for Hubs.

Usage

```
STATUS NU_USB_DEVICE_Get_CntnrID_Desc (
    NU_USB_DEVICE *cb,
    NU_USB_DEVCAP_CONTAINERID_DESC **cid_desc_out)
```

Arguments

- **cb**
A pointer to NU_USB_DEVICE control block.
- **cid_desc_out**
Double pointer to a Container ID device capability descriptor. This points to a valid Container ID descriptor or NU_NULL when the function returns.

Return Values

- **NU_SUCCESS**
Operation completed successfully, 'cid_desc_out' points to a valid Container ID descriptor.
- **NU_NOT_PRESENT**
A Container ID device capability descriptor is not present.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.
- **NU_USB_INVLD_DESC**
The descriptor type of ContainerID device capability descriptor is not USB_DT_DEVCAP or the device capability type of the ContainerID device capability descriptor is not USB_DCT_CONTID.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_Current_Requirement

This function gets the value of the current required by a particular configuration.

Usage

```
STATUS NU_USB_DEVICE_Get_Current_Requirement (  
    NU_USB_DEVICE *cb,  
    UINT8         cfg_num,  
    UINT32        *req_current)
```

Arguments

- **cb**
A pointer to NU_USB_DEVICE control block.
- **cfg_num**
Configuration number.
- **req_current**
Pointer to UINT32 containing current requirement of a given configuration.

Return Values

- **NU_SUCCESS**
Operation completed successfully, 'req_current' contains required current for a configuration, identified by 'cfg_num'.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_Desc

This function returns a pointer to the device descriptor as defined by the Nucleus USB standard in `device_desc_out`.

Usage

```
STATUS NU_USB_DEVICE_Get_Desc (NU_USB_DEVICE      *cb,  
                               NU_USB_DEVICE_DESC **device_desc_out)
```

Arguments

- `cb`
Pointer to the device control block.
- `device_desc_out`
Pointer to the memory location to hold pointer to device descriptor.

Return Values

- `NU_SUCCESS`
Indicates successful completion of the service.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_Function_Addr

This function returns the function address (in `function_address_out`) assigned by the host to the device during enumeration. Root hub's function address is defined by the macro `USB_ROOT_HUB`.

Usage

```
STATUS NU_USB_DEVICE_Get_Function_Addr (
                                         NU_USB_DEVICE *cb,
                                         UINT8         *function_address_out)
```

Arguments

- `cb`
Pointer to the device control block.
- `function_address_out`
Pointer to the variable to hold the function address of the device. Function addresses are in the range of 0 to 127. Function address 0 indicates that the device is in un-addressed state.

Return Values

- `NU_SUCCESS`
Indicates successful completion of the service.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_Hw

This function returns the pointer to the control block of the associated hardware in `hw_out`.

Usage

```
STATUS NU_USB_DEVICE_Get_Hw (NU_USB_DEVICE *cb,  
                             NU_USB_HW      **hw_out)
```

Arguments

- `cb`
Pointer to the device control block.
- `hw_out`
Pointer to a memory location to hold the pointer to the control block of the associated hardware.

Return Values

- `NU_SUCCESS`
Indicates successful completion of the service.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_idProduct

This function returns the product ID (idProduct field of the device descriptor) in idProduct_out.

Usage

```
STATUS NU_USB_DEVICE_Get_idProduct (NU_USB_DEVICE *cb,  
                                     UINT16      *idProduct_out)
```

Arguments

- cb
Pointer to the device control block.
- idProduct_out
Pointer to the variable to hold the product id.

Return Values

- NU_SUCCESS
Indicates successful completion of the service.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_idVendor

This function returns the vendor ID (idVendor field of the device descriptor), in idVendor_out.

Usage

```
STATUS NU_USB_DEVICE_Get_idVendor (NU_USB_DEVICE *cb,  
                                  UINT16          *idVendor_out)
```

Arguments

- cb
Pointer to the device control block.
- idVendor_out
Pointer to the variable to hold the vendor id.

Return Values

- NU_SUCCESS
Indicates successful completion of the service.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_Is_Claimed

This function returns NU_FALSE in is_claimed_out if no driver currently owns the device and NU_TRUE otherwise. If a driver currently owns the device, a pointer to its control block is returned in drv_r_out.

Usage

```
STATUS NU_USB_DEVICE_Get_Is_Claimed (NU_USB_DEVICE *cb,
                                     BOOLEAN         *is_claimed_out,
                                     NU_USB_DRV_R    **drv_r_out)
```

Arguments

- cb
 Pointer to the device control block.
- is_claimed_out
 Pointer to a variable to hold the ownership status of the device.
- drv_r_out
 Pointer to a memory location to hold the pointer to the driver control block that currently owns the device.

Return Values

- NU_SUCCESS
 Indicates successful completion of the service.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_Manf_String

This function copies the manufacturer string of the device into string_out.

Usage

```
STATUS NU_USB_DEVICE_Get_Manf_String (NU_USB_DEVICE *cb,  
                                     CHAR           *string_out)
```

Arguments

- **cb**
Pointer to the device control block.
- **string_out**
Pointer to the user supplied array of characters of size NU_USB_MAX_STRING_LEN.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_NOT_PRESENT**
Indicates that the manufacturer string number cannot be found in the device's set of string descriptors.

Description

English is assumed to be the Unicode of the manufacturer string descriptor. If the Unicode is not English, refer to [NU_USB_DEVICE_Get_Manf_String_Desc](#) for further information on how to retrieve a displayable string.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_Manf_String_Desc

This function gets pointer to the string descriptor (into string_desc_out) for the manufacturer string number (iManufacturer field of device descriptor).

Usage

```
STATUS NU_USB_DEVICE_Get_Manf_String_Desc (
    NU_USB_DEVICE *cb,
    UINT16 wLangId,
    NU_USB_STRING_DESC *string_desc_out)
```

Arguments

- **cb**
Pointer to the device control block.
- **wLangId**
2-byte Unicode language ID for the string.
- **string_desc_out**
Pointer to memory location to hold pointer to string descriptor structure.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_NOT_PRESENT**
Indicates that the manufacturer string number cannot be found in the device's set of string descriptors.

Description

If the Unicode used for string descriptors is English, [NU_USB_DEVICE_Get_Manf_String](#) can be used to directly get the ASCII string. For other Unicodes, this function will help retrieve the descriptor and then reader can write routines to convert from their Unicode format to ASCII string or such other display string notations.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_Manf_String_Num

This function returns the iManufacturer field of the device descriptor in string_num_out. If string_num_out is 0, there is no manufacturer string descriptor associated with this device.

Usage

```
STATUS NU_USB_DEVICE_Get_Manf_String_Num (NU_USB_DEVICE *cb,  
                                           UINT8          *string_num_out)
```

Arguments

- **cb**
Pointer to the device control block.
- **string_num_out**
Pointer to the variable to hold iManufacturer field of the device descriptor.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_Num_Cfgs

This function returns the number configurations that the device supports in number_cfgs_out.

Usage

```
STATUS NU_USB_DEVICE_Get_Num_Cfgs (NU_USB_DEVICE *cb,  
                                   UINT8          *number_cfgs_out)
```

Arguments

- **cb**
Pointer to the device control block.
- **number_cfgs_out**
Pointer to the variable to hold number of configurations.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_OTG_Desc

This function returns a pointer to the OTG descriptor of the device.

Usage

```
STATUS NU_USB_DEVICE_Get_OTG_Desc (NU_USB_DEVICE *cb,  
                                   NU_USB_OTG_DESC **otg_desc_out)
```

Arguments

- **cb**
A pointer to NU_USB_DEVICE control block.
- **otg_desc_out**
Output argument, pointing to the OTG descriptor of the device on return.

Return Values

- **NU_SUCCESS**
Operation completed successfully, 'otg_desc_out' points to the OTG descriptor of the device when the function returns.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_OTG_Status

This function returns the current OTG status of the device in the location pointed to by `otg_desc_out`.

Usage

```
STATUS NU_USB_DEVICE_Get_OTG_Status (NU_USB_DEVICE *cb,
                                     UINT8          *otg_status_out)
```

Arguments

- `cb`
A pointer to `NU_USB_DEVICE` control block.
- `otg_status_out`
A pointer to the OTG status when the function returns.

Return Values

- `NU_SUCCESS`
Operation completed successfully, 'cfg_out' contains a pointer to the current active configuration.
- `NU_USB_INVLD_ARG`
One or more input arguments are invalid.

Description

The OTG status is encoded as follows:

- b0: SRP support by the device.
- b1: HNP support by the device.
- b2-b3: Reserved.
- b4: SRP is enabled on the device.
- b5: HNP is enabled on the device.
- b6-b7: Reserved.

If b0 and b1 are set, that indicates that SRP/HNP is supported by the device. If they are not set, the device doesn't support these protocols and is not OTG capable. Bits b4 and b5 must be ignored if b0 and b1 are not set respectively.

If b4 and b5 are set, that indicates that SRP/HNP is enabled on the device by the host. If they are not set, that indicates that SRP/HNP is disabled on the device by the host.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_Parent

This function returns the pointer to the device control block of the hub to which it is physically connected in parent_out. There is no parent hub to a root hub. Therefore, if this function is called for a root hub it returns NU_USB_INVLD_ARG.

Usage

```
STATUS NU_USB_DEVICE_Get_Parent (NU_USB_DEVICE *cb,  
                                NU_USB_DEVICE **parent_out)
```

Arguments

- **cb**
Pointer to the device control block.
- **parent_out**
Pointer to a memory location to hold the pointer to the device control block of the hub to which the device is physically connected.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_USB_INVLD_ARG**
Indicates that either 'cb' points to an invalid device control block or that it points to the root hub control block.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_Port_Number

This function returns port number on the hub to which it is physically connected in port_num_out. There is no parent hub to a root hub, so if this function is called for a root hub it returns NU_USB_INVLD_ARG.

Usage

```
STATUS NU_USB_DEVICE_Get_Port_Number (NU_USB_DEVICE *cb,  
                                     UINT8          *port_num_out)
```

Arguments

- **cb**
Pointer to the device control block.
- **port_num_out**
Pointer to the variable to hold the port number on the hub to which the device is physically connected.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_USB_INVLD_ARG**
Indicates that either 'cb' points to an invalid device control block or that it points to the root hub control block.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_Product_String

This function copies the product string of the device into string_out.

Usage

```
STATUS NU_USB_DEVICE_Get_Product_String (NU_USB_DEVICE *cb,  
                                         CHAR *string_out)
```

Arguments

- **cb**
Pointer to the device control block.
- **string_out**
Pointer to the user supplied array of characters of size NU_USB_MAX_STRING_LEN.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_NOT_PRESENT**
Indicates that the product string number cannot be found in the device's set of string descriptors.

Description

English is assumed to be the Unicode of the manufacturer string descriptor. If the Unicode is not English, refer to [NU_USB_DEVICE_Get_Product_String_Desc](#) for further information on how to retrieve a displayable string.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_Product_String_Desc

This function gets pointer to the string descriptor (into string_desc_out) for the product string number (iProduct field of device descriptor).

Usage

```
STATUS NU_USB_DEVICE_Get_Product_String_Desc (
    NU_USB_DEVICE *cb,
    UINT16 wLangId,
    NU_USB_STRING_DESC *string_desc_out)
```

Arguments

- **cb**
Pointer to the device control block.
- **wLangId**
2-byte Unicode language ID for the string.
- **string_desc_out**
Pointer to memory location to hold pointer to string descriptor structure.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_NOT_PRESENT**
Indicates that the product string number cannot be found in the device's set of string descriptors.

Description

If the Unicode used for the string descriptors is English, [NU_USB_DEVICE_Get_Product_String](#) can be used to get the ASCII string directly. For other Unicodes, this function will help retrieve the descriptor, and then the reader can write routines to convert from their Unicode format to ASCII string or such other display string notations.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_Product_String_Num

This function returns the iProduct field of the device descriptor in string_num_out. If string_num_out is 0, there is no product string descriptor associated with this device.

Usage

```
STATUS NU_USB_DEVICE_Get_Product_String_Num (
                                         NU_USB_DEVICE *cb,
                                         UINT8         *string_num_out)
```

Arguments

- **cb**
Pointer to the device control block.
- **string_num_out**
Pointer to the variable to hold iProduct field of the device descriptor.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_Serial_Num_String

This function copies the serial number string of the device into string_out.

Usage

```
STATUS NU_USB_DEVICE_Get_Serial_Num_String (NU_USB_DEVICE *cb,  
                                             CHAR *string_out)
```

Arguments

- **cb**
Pointer to the device control block.
- **string_out**
Pointer to the user supplied array of characters of size NU_USB_MAX_STRING_LEN.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_NOT_PRESENT**
Indicates that the serial number string number cannot be found in the device's set of string descriptors.

Description

English is assumed to be the Unicode of the serial number string descriptor. If the Unicode is not English, refer to [NU_USB_DEVICE_Get_Serial_Num_String_Desc](#) for further information on how to retrieve a displayable string.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_Serial_Num_String_Desc

This function gets pointer to the string descriptor (into string_desc_out) for the serial number string number (iSerialNumber field of device descriptor).

Usage

```
STATUS NU_USB_DEVICE_Get_Serial_Num_String_Desc (
    NU_USB_DEVICE *cb,
    UINT16 wLangId,
    NU_USB_STRING_DESC **string_desc_out)
```

Arguments

- **cb**
Pointer to the device control block.
- **wLangId**
2-byte Unicode language ID for the string.
- **string_desc_out**
Pointer to the memory location to hold pointer to string descriptor structure.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_NOT_PRESENT**
Indicates that the serial number string number cannot be found in the device's set of string descriptors.

Description

If the Unicode used for the string descriptors is English, [NU_USB_DEVICE_Get_Serial_Num_String](#) can be used to directly get the ASCII string. For other Unicodes, this function will help retrieve the descriptor so that the reader can write routines to convert from their Unicode format to the ASCII string or such other display string notations.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_Serial_Num_String_Num

This function returns the iSerialNumber field of the device descriptor in string_num_out. If string_num_out is 0, there is no serial number string descriptor associated with this device.

Usage

```
STATUS NU_USB_DEVICE_Get_Serial_Num_String_Num (  
                                                    NU_USB_DEVICE *cb,  
                                                    UINT8      *string_num_out)
```

Arguments

- **cb**
Pointer to the device control block.
- **string_num_out**
Pointer to the variable to hold iSerialNumber field of the device descriptor.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_Speed

This function returns the speed of the device in speed_out.

Usage

```
STATUS NU_USB_DEVICE_Get_Speed (NU_USB_DEVICE *cb,  
                                UINT8          *speed_out)
```

Arguments

- **cb**
Pointer to the device control block.
- **speed_out**
Pointer to the variable to hold the speed of the device. The possible return values are USB_SPEED_UNKNOWN, USB_SPEED_LOW, USB_SPEED_FULL, and USB_SPEED_HIGH.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_Stack

This function returns the pointer to the control block of the associated stack, in `stack_out`.

Usage

```
STATUS NU_USB_DEVICE_Get_Stack (NU_USB_DEVICE *cb,  
                                NU_USB_STACK **stack_out)
```

Arguments

- `cb`
Pointer to the device control block.
- `stack_out`
Pointer to a memory location to hold the pointer to the control block of the associated stack.

Return Values

- `NU_SUCCESS`
Indicates successful completion of the service.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_Status

This function returns the status of the device in status_out. status_out is a 16-bit field and the meaning of each bit is described as follows.

Usage

```
STATUS NU_USB_DEVICE_Get_Status (NU_USB_DEVICE *cb,  
                                UINT16          *status_out)
```

Arguments

- cb
Pointer to the device control block.
- status_out
Pointer to the variable to hold the device status.

Return Values

- NU_SUCCESS
Indicates successful completion of the service.

Description

D0 is the least significant bit.

Figure 12-1. NU_USB_DEVICE_Get_Status Bit Description

D7	D6	D5	D4	D3	D2	D1	D0
Reserved (Reset to zero)						Remote Wakeup	Self Powered
D15	D14	D13	D12	D11	D10	D9	D8
Reserved (Reset to zero)							

The Self Powered field indicates whether the device is currently self-powered. If D0 is reset to zero, the device is bus-powered. If D0 is set to one, the device is self-powered.

The Remote Wakeup field indicates whether the device is currently enabled to request remote wakeup. The default mode for devices that support remote wakeup is disabled. If D1 is reset to zero, the ability of the device to signal remote wakeup is disabled. If D1 is set to one, the ability of the device to signal remote wakeup is enabled.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_String

This function copies the string (into string_out) corresponding to the specified string number. The string descriptors are assumed to be in English. The total number of the string descriptors in the device cannot be more than NU_USB_MAX_STRINGS.

Usage

```
STATUS NU_USB_DEVICE_Get_String (NU_USB_DEVICE *cb,
                                UINT8          string_num,
                                CHAR           *string_out)
```

Arguments

- **cb**
Pointer to the device control block.
- **string_num**
Number of the string as assigned in device or configuration or interface descriptor.
- **string_out**
Pointer to the user supplied array of characters of size NU_USB_MAX_STRING_LEN.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_NOT_PRESENT**
Indicates that the specified string number cannot be found in the device's set of string descriptors.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_String_Desc

This function gets pointer to the string descriptor (into string_desc_out) for the specified string number.

Usage

```
STATUS NU_USB_DEVICE_Get_String_Desc (
    NU_USB_DEVICE *cb,
    UINT8 string_num,
    UINT16 wLangId,
    NU_USB_STRING_DESC **string_desc_out)
```

Arguments

- **cb**
Pointer to the device control block.
- **string_num**
Number of the string as assigned in device or configuration or interface descriptor.
- **wLangId**
2-byte Unicode language ID for the string.
- **string_desc_out**
Pointer to memory location to hold pointer to string descriptor structure.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_NOT_PRESENT**
Indicates that the specified string number cannot be found in the device's set of string descriptors.

Description

If the Unicode used for the string descriptors is English, [NU_USB_DEVICE_Get_String](#) can be used to get the ASCII string directly. For other Unicodes, this function will help retrieve the descriptor so that the reader can write routines to convert from their Unicode format to the ASCII string or such other display string notations. USB devices may support multiple languages for their string descriptors. This API, when invoked with a string_num value of 0, retrieves the descriptor containing all Unicode language IDs supported by the device.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_SuprSpd_Desc

This API is used for extracting the Super Speed device capability descriptor from the NU_USB_DEVICE control block. According to the USB 3.0 specifications, all Super Speed devices will implement a Super Speed device capability descriptor.

Usage

```
STATUS NU_USB_DEVICE_Get_SuprSpd_Desc (
    NU_USB_DEVICE *cb,
    NU_USB_DEVCAP_SUPERSPEED_DESC **ss_desc_out)
```

Arguments

- **cb**
A pointer to NU_USB_DEVICE control block.
- **ss_desc_out**
Double pointer to a Super Speed device capability descriptor. This points to a valid Super Speed descriptor or NU_NULL when the function returns.

Return Values

- **NU_SUCCESS**
Operation completed successfully, 'ss_desc_out' points to a valid Super Speed descriptor.
- **NU_USB_INVLD_DESC**
The descriptor type of SuperSpeed device capability descriptor is not USB_DT_DEVCAP or the Device capability type of SuperSpeed device capability descriptor is not USB_DCT_USBSS.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Get_USB2Ext_Desc

This API is used for extracting the USB 2 extension device capability descriptor from the NU_USB_DEVICE control block. According to the USB 3.0 specifications, a Super Speed device will include a USB 2 extension device capability descriptor.

Usage

```
STATUS NU_USB_DEVICE_Get_USB2Ext_Desc (
    NU_USB_DEVICE *cb,
    NU_USB_DEVCAP_USB2EXT_DESC **usb2ext_desc_out)
```

Arguments

- **cb**
A pointer to NU_USB_DEVICE control block.
- **usb2ext_desc_out**
Double pointer to a USB 2 extension device capability descriptor. This points to a valid USB 2 extension descriptor or NU_NULL when the function returns.

Return Values

- **NU_SUCCESS**
Operation completed successfully, 'usb2ext_desc_out' points to a valid USB 2 extension descriptor.
- **NU_USB_INVLD_DESC**
The descriptor type of USB 2 extension device capability descriptor is not USB_DT_DEVCAP or the device capability type of the USB 2 extension descriptor is not USB_DCT_USB2EXT.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.
- **NU_NOT_PRESENT**
The USB 2 extension device capability descriptor is not present.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Release

This function releases the driver's ownership of the device.

Usage

```
STATUS NU_USB_DEVICE_Release (NU_USB_DEVICE *cb)
```

Arguments

- **cb**
Pointer to the device control block.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Description

A USB device driver that currently owns the device is only expected to make this function call. Once the ownership is released, any other driver may claim ownership of the device. The delete function of a class driver typically calls this function for all of the owned devices.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Set_Active_Cfg

This function sets the specified configuration as active on the device.

Usage

```
STATUS NU_USB_DEVICE_Set_Active_Cfg (NU_USB_DEVICE *cb,  
                                     NU_USB_CFG      *cfg)
```

Arguments

- **cb**
Pointer to the device control block.
- **cfg**
Pointer to the control block of the configuration that is made active.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_USB_NO_BANDWIDTH**
Indicates that the device's bandwidth requirements of the specified configuration cannot be met.
- **NU_INVALID_SEMAPHORE**
Indicates that the one of internal semaphore pointer is corrupted.
- **NU_SEMAPHORE_DELETED**
Indicates that one of the internal semaphores was deleted.
- **NU_UNAVAILABLE**
Indicates that one of the internal semaphores is unavailable.
- **NU_INVALID_SUSPEND**
Indicates that this API is called from a non-task thread.
- **NU_USB_INVLD_ARG**
Indicates that device control block or configuration control blocks passed to this function are invalid.

Description

The Nucleus USB bandwidth and necessary resources in the controller hardware are allocated to make the configuration active.

Nucleus USB interface drivers are not allowed to make this call as they own an interface on the device. Only Nucleus USB device drivers are entitled to make this call.

Only host class drivers should use this function. It is the host's prerogative to select and activate a configuration on the device.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Set_bcdUSB

This function sets the bcdUSB field of device descriptor. The value of this field depends on speed of underlying USB hardware controller.

Usage

```
STATUS NU_USB_DEVICE_Set_bcdUSB (NU_USB_DEVICE *cb,  
                                UINT16          bcdUSB)
```

Arguments

- **cb**
Pointer to the device control block.
- **bcdUSB**
Value of bcdUSB to be set in device descriptor.

Return Values

- **NU_USB_INVLD_ARG**
An argument is invalid
- **NU_SUCCESS**
Indicates successful completion of the service.

Description

This function is called during USB function stack initialization if USB hardware is capable of operating at high speed.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Set_bDeviceClass

This function sets the device subclass field of device descriptor.

Usage

```
STATUS NU_USB_DEVICE_Set_bDeviceClass (NU_USB_DEVICE *cb,  
                                       UINT8          bDeviceClass)
```

Arguments

- **cb**
Pointer to the device control block.
- **bDeviceClass**
Device class of USB function device.

Return Values

- **NU_USB_INVLD_ARG**
An argument is invalid.
- **NU_SUCCESS**
Indicates successful completion of the service.

Description

This function is called to set the device class of USB function device in device descriptor. Usually this field defaults to zero.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Set_bMaxPacketSize0

This function sets the maximum packet size of the default control endpoint of the device.

Usage

```
STATUS NU_USB_DEVICE_Set_bMaxPacketSize0 (NU_USB_DEVICE *cb,  
                                           UINT8          bMaxPacketSize0)
```

Arguments

- **cb**
Pointer to NU_USB_DEVICE control block.
- **bMaxPacketSize0**
Maximum packet size of the default control endpoint to be set.

Return Values

- **NU_SUCCESS**
Operation completed successfully. The specified maximum packet size is saved in the device descriptor.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Set_Desc

This function is called to initialize the device descriptor of the device. This function copies value of device descriptors to the 'device_descriptor' field of NU_USB_DEVICE control block.

Usage

```
STATUS NU_USB_DEVICE_Set_Desc (NU_USB_DEVICE      *cb,
                              NU_USB_DEVICE_DESC *device_desc)
```

Arguments

- **cb**
Pointer to the device control block.
- **device_desc**
Pointer to device descriptor control block.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Description

Initially an instance of NU_USB_DEVICE control block do not has a valid device descriptor when used in function mode. Nucleus USB stack calls this function to initialize the device descriptor field of NU_USB_DEVICE control block.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Set_Device_Qualifier

This function initializes the device qualifier descriptor in USB device control block. This function is only called if target board has a high speed USB function controller.

Usage

```
STATUS NU_USB_DEVICE_Set_Device_Qualifier (  
    NU_USB_DEVICE *cb,  
    NU_USB_DEV_QUAL_DESC *dev_qualifier)
```

Arguments

- **cb**
Pointer to the device control block.
- **dev_qualifier**
Pointer to device qualifier descriptor.

Return Values

- **NU_USB_INVLD_ARG**
An argument is invalid
- **NU_SUCCESS**
Indicates successful completion of the service.

Description

This function is called during USB function stack initialization if underlying hardware is capable of operating at high speed.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Set_Hw

This function sets the pointer to the control block of the associated hardware, in the device control block.

Usage

```
STATUS NU_USB_DEVICE_Set_Hw (NU_USB_DEVICE *cb,  
                             NU_USB_HW      *hw)
```

Arguments

- **cb**
Pointer to the device control block.
- **hw**
Pointer to the control block of the associated hardware.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Set_Link_State

This API updates the 'link_state' variable in NU_USB_DEVICE control block to the newly reported link state.

Usage

```
STATUS NU_USB_DEVICE_Set_Link_State (NU_USB_DEVICE *device,  
                                     UINT8          link_state )
```

Arguments

- device
Pointer to NU_USB_DEVICE control block.
- link_state
Current state of the link to be set by the caller.

Return Values

- NU_SUCCESS
Operation completed successfully, the link state of the device is updated.
- NU_USB_INVLD_ARG
One or more input arguments are invalid.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Set_Manf_String

This function initializes the manufacturer string descriptor in USB device control block and saves its index in device descriptor.

Usage

```
STATUS NU_USB_DEVICE_Set_Manf_String (NU_USB_DEVICE *cb,
                                     UINT8          str_index,
                                     NU_USB_STRING  *string)
```

Arguments

- **cb**
Pointer to the device control block.
- **str_index**
Index of string descriptor.
- **string**
Pointer to string descriptor control block.

Return Values

- **NU_USB_INVLD_ARG**
An argument is invalid.
- **NU_SUCCESS**
Indicates successful completion of the service.

Description

This function is called by USB function stack to initialize the manufacturer string descriptor at a particular index in string descriptor array of device control block. This function also saves the index of manufacturer string in device descriptor.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Set_Product_String

This function initializes the product string descriptor in USB device control block and saves its index in device descriptor.

Usage

```
STATUS NU_USB_DEVICE_Set_Product_String (NU_USB_DEVICE *cb,  
                                         UINT8          str_index,  
                                         NU_USB_STRING *string)
```

Arguments

- **cb**
Pointer to the device control block.
- **str_index**
Index of string descriptor.
- **string**
Pointer to string descriptor control block.

Return Values

- **NU_USB_INVLD_ARG**
An argument is invalid
- **NU_SUCCESS**
Indicates successful completion of the service.

Description

This function is called by USB function stack to initialize the product string descriptor at a particular index in string descriptor array of device control block. This function also saves the index of product string in device descriptor.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Set_Stack

This function sets the pointer to the control block of the associated stack.

Usage

```
STATUS NU_USB_DEVICE_Set_Stack (NU_USB_DEVICE *cb,  
                                NU_USB_STACK *stack)
```

Arguments

- **cb**
Pointer to the device control block.
- **stack**
Pointer to the associated stack control block.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Set_Status

This function enables or disables the remote wake up capability of the device.

Usage

```
STATUS NU_USB_DEVICE_Set_Status (NU_USB_DEVICE *cb,  
                                UINT16          status)
```

Arguments

- **cb**
Pointer to the device control block.
- **status**
The device status.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Description

Bit D1 of the status should be 1 to enable and 0 to disable remote wake up. The rest of the bits in the status are reserved for now.

This function should be used on the host software. It is the host's prerogative to enable or disable remote wakeup capability.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Set_String

This function initializes the string descriptor on specified index of string descriptors list.

Usage

```
STATUS NU_USB_DEVICE_Set_String (NU_USB_DEVICE *cb,
                                UINT8          str_index,
                                NU_USB_STRING  *string)
```

Arguments

- **cb**
Pointer to the device control block.
- **str_index**
Index of string descriptor
- **string**
Pointer to string descriptor control block.

Return Values

- **NU_USB_INVLD_ARG**
An argument is invalid
- **NU_SUCCESS**
Indicates successful completion of the service.

Description

This function is called by USB function stack to initialize a string descriptor at a particular index in string descriptor array of device control block.

Related Topics

[Nucleus USB Device Services](#)

NU_USB_DEVICE_Set_Serial_Num_String

This function initializes the serial string descriptor in USB device control block and saves its index in device descriptor.

Usage

```
STATUS NU_USB_DEVICE_Set_Serial_Num_String (NU_USB_DEVICE *cb,  
                                             UINT8          str_index,  
                                             NU_USB_STRING *string)
```

Arguments

- **cb**
Pointer to the device control block.
- **str_index**
Index of string descriptor
- **string**
Pointer to string descriptor control block.

Return Values

- **NU_USB_INVLD_ARG**
An argument is invalid
- **NU_SUCCESS**
Indicates successful completion of the service.

Description

This function is called by USB function stack to initialize the serial number string descriptor at a particular index in string descriptor array of device control block. This function also saves the index of serial number string in device descriptor.

Related Topics

[Nucleus USB Device Services](#)

Nucleus USB Driver Services

This section provides a detailed reference of the following Nucleus USB Driver Services:

- [NU_USB_DRV_Register_User](#)
- [NU_USB_DRV_Get_Users](#)
- [NU_USB_DRV_Get_Users_Count](#)
- [NU_USB_DRV_Deregister_User](#)

Related Topics

[USB Host and Functions](#)

NU_USB_DRV_R_Deregister_User

This function deregisters the specified user driver of this class driver.

Usage

```
STATUS NU_USB_DRV_R_Deregister_User (NU_USB_DRV_R *cb,  
                                     NU_USB_USER *user)
```

Arguments

- **cb**
Pointer to the driver control block.
- **user**
Pointer to the user driver control block.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_USB_INVLD_ARG**
Indicates that the control block is not found in the list of the registered user drivers of the driver.

Related Topics

[Nucleus USB Driver Services](#)

NU_USB_DRV_Get_Users

This function returns pointers to control blocks of the registered user drivers.

Usage

```
UNSIGNED NU_USB_DRV_Get_Users (NU_USB_DRV *cb,
                               NU_USB_USER **users,
                               UNSIGNED   num_requested)
```

Arguments

- **cb**
Pointer to the driver control block.
- **users**
Pointer to the user supplied array of memory locations to hold pointers to the user control blocks.
- **num_requested**
The size of the users array.

Return Values

- Number of user control blocks copied in to user drivers array.

Related Topics

[Nucleus USB Driver Services](#)

NU_USB_DRV_Get_Users_Count

This function returns the number of registered user drivers of this class driver.

Usage

```
UNSIGNED NU_USB_DRV_Get_Users_Count (NU_USB_DRV *cb)
```

Arguments

- **cb**
Pointer to the driver control block.

Return Values

- Number of user drivers.

Related Topics

[Nucleus USB Driver Services](#)

NU_USB_DRV_Register_User

This function registers a user driver of the class driver.

Usage

```
STATUS NU_USB_DRV_Register_User (NU_USB_DRV *cb,  
                                NU_USB_USER *user)
```

Arguments

- **cb**
Pointer to the driver control block.
- **user**
Pointer to the user control block.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_USB_MAX_EXCEEDED**
Indicates that the maximum number of user drivers registered with this class driver has exceeded **USB_MAX_USERS**.

Related Topics

[Nucleus USB Driver Services](#)

Nucleus USB ENDPoint Services

This section provides a detailed reference of the following Nucleus USB ENDPoint Services:

- [NU_USB_ENDP_Get_Alt_Settg](#)
- [NU_USB_ENDP_Get_Bulk_MaxStreams](#)
- [NU_USB_ENDP_Get_BytesPerInterval](#)
- [NU_USB_ENDP_Get_Class_Desc](#)
- [NU_USB_ENDP_Get_Companion_Desc](#)
- [NU_USB_ENDP_Get_Desc](#)
- [NU_USB_ENDP_Get_Device](#)
- [NU_USB_ENDP_Get_Direction](#)
- [NU_USB_ENDP_Get_Interval](#)
- [NU_USB_ENDP_Get_Iso_MaxPktPerIntrvl](#)
- [NU_USB_ENDP_Get_MaxBurst](#)
- [NU_USB_ENDP_Get_Max_Packet_Size](#)
- [NU_USB_ENDP_Get_Number](#)
- [NU_USB_ENDP_Get_Num_Transactions](#)
- [NU_USB_ENDP_Get_Pipe](#)
- [NU_USB_ENDP_Get_Status](#)
- [NU_USB_ENDP_Get_Sync_Type](#)
- [NU_USB_ENDP_Get_Transfer_Type](#)
- [NU_USB_ENDP_Get_Usage_Type](#)

Related Topics

[USB Host and Functions](#)

NU_USB_ENDP_Get_Alt_Settg

This function returns a pointer to the control block of the associated alternate setting.

Usage

```
STATUS NU_USB_ENDP_Get_Alt_Settg (NU_USB_ENDP      *cb,  
                                  NU_USB_ALT_SETTG  **alt_setting_out)
```

Arguments

- **cb**
Pointer to endpoint control block.
- **alt_setting_out**
Pointer to the memory location to hold the pointer to the control block of the associated alternate setting.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB ENDPoint Services](#)

NU_USB_ENDP_Get_Bulk_MaxStreams

Call this API to get the Max Streams of SuperSpeed BULK endpoint. In the case of a SuperSpeed BULK endpoint the maximum number of streams is encoded in Bit 4:0 of the bmAttributes field of the SuperSpeed endpoint companion descriptor.

Usage

```
STATUS NU_USB_ENDP_Get_Bulk_MaxStreams (NU_USB_ENDP *cb,  
                                         UINT32      *maxstreams_out)
```

Arguments

- **cb**
Pointer to NU_USB_ENDP control block.
- **maxstreams_out**
A pointer to the value Max streams (Bit 4-0 of the bmAttributes field) supported by SuperSpeed BULK endpoint.

Return Values

- **NU_SUCCESS**
Operation completed successfully, 'maxstreams_out' contains the maximum number of streams supported by SuperSpeed BULK endpoint.
- **NU_USB_NOT_SUPPORTED**
Endpoint is not a BULK endpoint.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.
- **NU_NOT_PRESENT**
There is no SuperSpeed endpoint descriptor associated with this endpoint.

Related Topics

[Nucleus USB ENDPoint Services](#)

NU_USB_ENDP_Get_BytesPerInterval

Call this API to get the wBytesPerInterval field of a SuperSpeed periodic (Interrupt and Isochronous) endpoints. This field actually determines how many bytes this endpoint can transfer in a single service interval.

Usage

```
STATUS NU_USB_ENDP_Get_BytesPerInterval (
                                NU_USB_ENDP *cb,
                                INT16      *bytesperinterval_out)
```

Arguments

- **cb**
Pointer to NU_USB_ENDP control block.
- **bytesperinterval_out**
A pointer to the value of wBytesPerInterval by a periodic SuperSpeed endpoint.

Return Values

- **NU_SUCCESS**
Operation completed successfully, 'bytesperinterval_out' contains the value of wBytesPerInterval supported by the SuperSpeed periodic endpoint.
- **NU_USB_NOT_SUPPORTED**
Endpoint is not periodic (interrupt or isochronous).
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.
- **NU_NOT_PRESENT**
There is no SuperSpeed endpoint descriptor associated with this endpoint.

Related Topics

[Nucleus USB ENDPoint Services](#)

NU_USB_ENDP_Get_Class_Desc

This function returns the pointer to the class specific endpoint descriptor, if any, associated with the endpoint. If there is no class specific endpoint descriptor associated with the given endpoint, the memory location pointed to by the length out parameter will be initialized to zero.

Usage

```
STATUS NU_USB_ENDP_Get_Class_Desc (NU_USB_ENDP *cb,  
                                  UINT8         **class_desc_out,  
                                  UINT32         *length_out)
```

Arguments

- **cb**
Pointer to the endpoint control block.
- **class_desc_out**
Pointer to the memory location to hold the pointer to the class specific endpoint descriptor.
- **length_out**
Pointer to the variable to hold the length of the class specific descriptor in bytes. Will be initialized to zero if the endpoint does not have an associated class specific descriptor,

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB ENDPoint Services](#)

NU_USB_ENDP_Get_Companion_Desc

Calls this API to get the SuperSpeed endpoint companion descriptor associated with an endpoint. This API returns the whole descriptor. Apart from it there are several other APIs available for getting individual fields of the SuperSpeed endpoint companion descriptor.

Usage

```
STATUS NU_USB_ENDP_Get_Companion_Desc (
    NU_USB_ENDP          *cb,
    NU_USB_SSEPCOMPANION_DESC **epcompanion_desc_out)
```

Arguments

- **cb**
Pointer to NU_USB_ENDP control block.
- **epcompanion_desc_out**
Pointer to NU_USB_SSEPCOMPANION_DESC. This points to a valid SuperSpeed endpoint companion descriptor or NU_NULL if a descriptor is not present.

Return Values

- **NU_SUCCESS**
Operation completed successfully, 'epcompanion_desc_out' points to a valid SuperSpeed endpoint companion descriptor.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.
- **NU_NOT_PRESENT**
There is no SuperSpeed endpoint descriptor associated with this endpoint.

Description

It is recommended that you call this function to get access to SuperSpeed endpoint companion descriptor instead of accessing it directly through the endpoint control block.

Related Topics

[Nucleus USB ENDPoint Services](#)

NU_USB_ENDP_Get_Desc

This function returns a pointer to the endpoint descriptor.

Usage

```
STATUS NU_USB_ENDP_Get_Desc (NU_USB_ENDP      *cb,  
                             NU_USB_ENDP_DESC **endp_desc_out)
```

Arguments

- **cb**
Pointer to the endpoint control block.
- **endp_desc_out**
Pointer to a memory location to hold the pointer to the endpoint descriptor.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB ENDPoint Services](#)

NU_USB_ENDP_Get_Device

This function returns a pointer to the control block of the associated device.

Usage

```
STATUS NU_USB_ENDP_Get_Device (NU_USB_ENDP    *cb,  
                               NU_USB_DEVICE **device_out)
```

Arguments

- **cb**
Pointer to the endpoint control block.
- **device_out**
Pointer to the memory location to hold the pointer to the control block of the associated device.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB ENDPoint Services](#)

NU_USB_ENDP_Get_Direction

This function returns the direction of the endpoint, in `direction_out`.

Usage

```
STATUS NU_USB_ENDP_Get_Direction (NU_USB_ENDP *cb,  
                                UINT8         *direction_out)
```

Arguments

- `cb`
Pointer to the endpoint control block.
- `direction_out`
Pointer to the variable to hold the direction, which can be either `USB_DIR_IN` or `USB_DIR_OUT`.

Return Values

- `NU_SUCCESS`
Indicates successful completion of the service.

Related Topics

[Nucleus USB ENDPoint Services](#)

NU_USB_ENDP_Get_Interval

This function returns the interval of the endpoint in interval_out.

Usage

```
STATUS NU_USB_ENDP_Get_Interval (NU_USB_ENDP *cb,
                                UINT8        *interval_out)
```

Arguments

- **cb**
Pointer to the endpoint control block.
- **interval_out**
Pointer to the variable to hold the value of the endpoint interval.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB ENDPoint Services](#)

NU_USB_ENDP_Get_Iso_MaxPktPerIntrvl

Call this API to get the maximum number of packets within a service interval that this SuperSpeed isochronous endpoint supports.

Usage

```
STATUS NU_USB_ENDP_Get_Iso_MaxPktPerIntrvl (NU_USB_ENDP *cb,  
                                             UINT8      *max_packets_out)
```

Arguments

- **cb**
Pointer to NU_USB_ENDP control block.
- **max_packets_out**
A pointer to the maximum packets per service interval supported by the SuperSpeed isochronous endpoint.

Return Values

- **NU_SUCCESS**
Operation completed successfully, 'max_packets_out' contains the maximum packets per service interval supported by the SuperSpeed isochronous endpoint.
- **NU_USB_NOT_SUPPORTED**
Endpoint is not an isochronous endpoint.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.
- **NU_NOT_PRESENT**
There is no SuperSpeed endpoint descriptor associated with this endpoint.

Related Topics

[Nucleus USB ENDPoint Services](#)

NU_USB_ENDP_Get_MaxBurst

This API returns the bMaxBurst field of the SuperSpeed Endpoint Companion descriptor.

Usage

```
STATUS NU_USB_ENDP_Get_MaxBurst (NU_USB_ENDP *cb,  
                                UINT8      *maxburst_out)
```

Arguments

- **cb**
 Pointer to NU_USB_ENDP control block.
- **maxburst_out**
 A pointer to the value of bMaxBurst field of the SuperSpeed endpoint companion descriptor if the function completes without an error.

Return Values

- **NU_SUCCESS**
 Operation completed successfully, 'maxburst_out' contains the value of bMaxBurst field of the SuperSpeed endpoint companion descriptor.
- **NU_USB_INVLD_ARG**
 One or more input arguments are invalid.
- **NU_NOT_PRESENT**
 There is no SuperSpeed endpoint descriptor associated with this endpoint.

Related Topics

[Nucleus USB ENDPoint Services](#)

NU_USB_ENDP_Get_Max_Packet_Size

This function returns the max packet size of the endpoint, in max_packet_size_out.

Usage

```
STATUS NU_USB_ENDP_Get_Max_Packet_Size (NU_USB_ENDP *cb,  
                                         UINT16      *max_packet_size_out)
```

Arguments

- **cb**
Pointer to the endpoint control block.
- **max_packet_size_out**
Pointer to the variable to hold the max packet size of the endpoint.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB ENDPoint Services](#)

NU_USB_ENDP_Get_Number

This function returns the endpoint number in the range of 1 to 15, in number_out.

Usage

```
STATUS NU_USB_ENDP_Get_Number (NU_USB_ENDP *cb,  
                               UINT8       *number_out)
```

Arguments

- **cb**
Pointer to the endpoint control block.
- **number_out**
Pointer to the variable to hold the endpoint number.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB ENDPoint Services](#)

NU_USB_ENDP_Get_Num_Transactions

The function returns the number of transactions permitted per micro frame for this endpoint.

Usage

```
STATUS NU_USB_ENDP_Get_Num_Transactions (
                                         NU_USB_ENDP *cb,
                                         UINT8      *num_transactions_out)
```

Arguments

- **cb**
Pointer to the endpoint control block.
- **num_transactions_out**
Pointer to the variable to hold the value of the number of transactions.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB ENDPoint Services](#)

NU_USB_ENDP_Get_Pipe

This function returns a pointer to the control block of the associated pipe in pipe_out.

Usage

```
STATUS NU_USB_ENDP_Get_Pipe (NU_USB_ENDP *cb,  
                             NU_USB_PIPE **pipe_out)
```

Arguments

- **cb**
Pointer to endpoint control block.
- **pipe_out**
Pointer to memory location to hold the pointer to the associated pipe control block.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB ENDPoint Services](#)

NU_USB_ENDP_Get_SSEPC_bmAttributes

Call this API to get the bmAttributes field of a SuperSpeed endpoint companion descriptor for the particular endpoint.

Usage

```
STATUS NU_USB_ENDP_Get_SSEPC_bmAttributes (NU_USB_ENDP *cb,  
                                           UINT8      *bmAttributes_out)
```

Arguments

- **cb**
Pointer to NU_USB_ENDP control block.
- **bmAttributes_out**
A pointer to the value of the bmAttributes field of a SuperSpeed endpoint.

Return Values

- **NU_SUCCESS**
Operation completed successfully, 'bmAttributes_out' contains the value of the bmAttributes field of a SuperSpeed endpoint.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.
- **NU_NOT_PRESENT**
There is no SuperSpeed endpoint descriptor associated with this endpoint.

Related Topics

[Nucleus USB ENDPoint Services](#)

NU_USB_ENDP_Get_Status

This function returns the status of the endpoint in status_out. status_out is a 16-bit field and the meaning of each bit is described in [Figure 12-2](#).

Usage

```
STATUS NU_USB_ENDP_Get_Status (NU_USB_ENDP *cb,  
                               UINT16      *status_out)
```

Arguments

- cb
Pointer to the endpoint control block.
- status_out
Pointer to the variable to hold the value of the endpoint status.

Return Values

- NU_SUCCESS
Indicates successful completion of the service.

Description

D0 is the least significant bit.

Figure 12-2. NU_USB_ENDP_Get_Status Bit Description

D7	D6	D5	D4	D3	D2	D1	D0
Reserved (Reset to zero)							Halt
D15	D14	D13	D12	D11	D10	D9	D8
Reserved (Reset to zero)							

If the endpoint is currently halted, then the Halt bit is set to one. Otherwise, the Halt bit is reset to zero. When the endpoint is halted, the endpoint returns STALL pids to any tokens sent by the host.

Related Topics

[Nucleus USB ENDPoint Services](#)

NU_USB_ENDP_Get_Sync_Type

This function returns the sync type of the endpoint, in sync_type_out.

Usage

```
STATUS NU_USB_ENDP_Get_Sync_Type (NU_USB_ENDP *cb,  
                                UINT8 *sync_type_out)
```

Arguments

- cb
Pointer to endpoint control block.
- sync_type_out
Pointer to the variable to hold the sync type.

Return Values

- NU_SUCCESS
Indicates successful completion of the service.

Related Topics

[Nucleus USB ENDPoint Services](#)

NU_USB_ENDP_Get_Transfer_Type

This function returns the type of the transfer in transfer_type_out.

Usage

```
STATUS NU_USB_ENDP_Get_Transfer_Type (NU_USB_ENDP *cb,  
                                     UINT8      *transfer_type_out)
```

Arguments

- **cb**
Pointer to the endpoint control block.
- **transfer_type_out**
Pointer to the variable to hold transfer type, which is one of the following:
 - USB_EP_CTRL
 - USB_EP_ISO
 - USB_EP_BULK
 - USB_EP_INTR

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB ENDPoint Services](#)

NU_USB_ENDP_Get_Usage_Type

This function returns the usage type of the endpoint, in `usage_type_out`.

Usage

```
STATUS NU_USB_ENDP_Get_Usage_Type (NU_USB_ENDP *cb,  
                                   UINT8         *usage_type_out)
```

Arguments

- `cb`
Pointer to endpoint control block.
- `usage_type_out`
Pointer to the variable to hold the usage type.

Return Values

- `NU_SUCCESS`
Indicates successful completion of the service.

Related Topics

[Nucleus USB ENDPoint Services](#)

Nucleus USB Interface Association Descriptor (IAD) Services

This section provides a detailed reference of the following Nucleus USB IAD Services:

- [NU_USB_IAD_Check_Interface](#)
- [NU_USB_IAD_Get_Desc](#)
- [NU_USB_IAD_Get_First_Interface](#)
- [NU_USB_IAD_Get_Last_Interface](#)

Related Topics

[USB Host and Functions](#)

NU_USB_IAD_Check_Interface

This function checks if a specified interface ID belongs to this Interface Association Descriptor (IAD).

Usage

```
STATUS NU_USB_IAD_Check_Interface (NU_USB_IAD *cb,  
                                  UINT8      intf_num)
```

Arguments

- **cb**
Pointer to NU_USB_IAD control block.
- **intf_num**
Interface ID to be verified.

Return Values

- **NU_SUCCESS**
Indicates that the interface belongs to this IAD.
- **NU_USB_INVLD_ARG**
One or more invalid input arguments, or the interface does not belong to this IAD.

Related Topics

[Nucleus USB Interface Association Descriptor \(IAD\) Services](#)

NU_USB_IAD_Get_Desc

This function returns the Interface Association Descriptor, associated with a specific IAD control block.

Usage

```
STATUS NU_USB_IAD_Get_Desc (NU_USB_IAD      *cb,  
                           NU_USB_IAD_DESC **desc_out)
```

Arguments

- **cb**
Pointer to NU_USB_IAD control block.
- **desc_out**
Double pointer to the address of the IAD descriptor when the function returns.

Return Values

- **NU_SUCCESS**
Operation completed successfully. 'desc_out' contains the address of the IAD descriptor.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Interface Association Descriptor \(IAD\) Services](#)

NU_USB_IAD_Get_First_Interface

This function returns the interface number of the first interface in a specified IAD.

Usage

```
STATUS NU_USB_IAD_Get_First_Interface (NU_USB_IAD *cb,  
                                       UINT8      *intf_num_out)
```

Arguments

- **cb**
Pointer to NU_USB_IAD control block.
- **intf_num_out**
Pointer to a variable containing the number of the first interface in IAD.

Return Values

- **NU_SUCCESS**
Operation completed successfully. 'intf_num_out' contains the number of the first interface in IAD.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Interface Association Descriptor \(IAD\) Services](#)

NU_USB_IAD_Get_Last_Interface

This function returns the interface ID of the last interface in an IAD.

Usage

```
STATUS NU_USB_IAD_Get_Last_Interface (NU_USB_IAD *cb,
                                     UINT8      *intf_num_out)
```

Arguments

- **cb**
Pointer to NU_USB_IAD control block.
- **intf_num_out**
Pointer to a variable containing the number of the last interface in IAD.

Return Values

- **NU_SUCCESS**
Operation completed successfully. 'intf_num_out' contains the number of the last interface in IAD.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Interface Association Descriptor \(IAD\) Services](#)

Nucleus USB Interface Services

This section provides a detailed reference of the following Nucleus USB Interface Services:

- [NU_USB_INTF_Claim](#)
- [NU_USB_INTF_Find_Alt_Setting](#)
- [NU_USB_INTF_Get_Active_Alt_Setting](#)
- [NU_USB_INTF_Get_Active_Alt_Setting_Num](#)
- [NU_USB_INTF_Get_Alt_Setting](#)
- [NU_USB_INTF_Get_Cfg](#)
- [NU_USB_INTF_Get_Class](#)
- [NU_USB_INTF_Get_Desc](#)
- [NU_USB_INTF_Get_Device](#)
- [NU_USB_INTF_Get_IAD](#)
- [NU_USB_INTF_Get_Intf_Num](#)
- [NU_USB_INTF_Get_Is_Claimed](#)
- [NU_USB_INTF_Get_Num_Alt_Settings](#)
- [NU_USB_INTF_Get_Protocol](#)
- [NU_USB_INTF_Get_String](#)
- [NU_USB_INTF_Get_String_Desc](#)
- [NU_USB_INTF_Get_String_Num](#)
- [NU_USB_INTF_Get_SubClass](#)
- [NU_USB_INTF_Release](#)
- [NU_USB_INTF_Set_Interface](#)

Related Topics

[USB Host and Functions](#)

NU_USB_INTF_Claim

This function sets the driver as owner for the interface.

Usage

```
STATUS NU_USB_INTF_Claim (NU_USB_INTF *cb,  
                          NU_USB_DRV *drv)
```

Arguments

- **cb**
Pointer to the interface control block. NOTE: This pointer is passed to the USB driver when the stack invokes USB driver's `_NU_USB_DRV_Initialize_Intf` function.
- **drv**
Pointer to the control block of the driver that decided to own the interface.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Description

A USB interface class driver may call this function from its `_NU_USB_DRV_Initialize_Intf` function, once it decides to own the interface. Drivers usually decide to take ownership of the interface after successfully locating all necessary pipes and notifying the relevant user drivers.

Related Topics

[Nucleus USB Interface Services](#)

NU_USB_INTF_Find_Alt_Setting

This function finds a matching alternate setting among various alternate settings of the interfaces.

Usage

```
STATUS NU_USB_INTF_Find_Alt_Setting (
    NU_USB_INTF      *cb,
    UINT32            match_flag,
    UINT8             alt_settg,
    UINT8             bInterfaceClass,
    UINT8             bInterfaceSubClass,
    UINT8             bInterfaceProtocol,
    NU_USB_ALT_SETTG **alt_settg_out
```

Arguments

- **cb**

Pointer to the interface control block. The pointer to control block of the matching alternate setting is returned in alt_settg_out.

- **match_flag**

The match_flag specifies the search criteria. You can OR any of the following flags:

USB_MATCH_ONLY_ACTIVE_ALT_STTG

Matches only the active alternate setting. This does not attempt to match non-active alternate settings.

USB_MATCH_CLASS

Searches alternate setting whose bInterfaceClass equals the specified bInterface Class.

USB_MATCH_SUB_CLASS

Searches alternate setting whose bInterfaceSubClass equals the specified bInterfaceSubClass. The match_flag must also contain USB_MATCH_CLASS.

USB_MATCH_PROTOCOL

Searches for alternate setting whose bInterfaceProtocol equals the specified bInterfaceProtocol. The match_flag must also contain USB_MATCH_SUB_CLASS.

- **alt_settg**

Alternate Setting Number of the specified interface, from which the search should begin.

- **bInterfaceClass**

The desired value of the bInterfaceClass in the matching alternate setting.

- **bInterfaceSubClass**

The desired value of the bInterfaceSubClass in the matching alternate setting. Irrelevant, if match_flag does not contain USB_MATCH_SUB_CLASS.

- **bInterfaceProtocol**
The desired value of the bInterfaceProtocol in the matching alternate setting. Irrelevant, if match_flag does not contain USB_MATCH_PROTOCOL.
- **alt_settg_out**
Pointer to a memory location to hold the pointer to the control block of the matching alternate setting.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_NOT_PRESENT**
Indicates that no matching alternate setting could be found.
- **NU_USB_INVLD_ARG**
Indicates that the control block is invalid.

Related Topics

[Nucleus USB Interface Services](#)

NU_USB_INTF_Get_Active_Alt_Setting

This function returns a pointer to the control block of the alternate setting (in alt_setting_out) that is currently the active one. NU_NULL is returned in alt_setting_out if no alternate setting is active on the interface.

Usage

```
STATUS NU_USB_INTF_Get_Active_Alt_Setting (  
    NU_USB_INTF          *cb,  
    NU_USB_ALT_SETTG     **alt_setting_out)
```

Arguments

- **cb**
Pointer to the interface control block.
- **alt_setting_out**
Pointer to the memory location to hold the pointer to the control block of the active alternate setting.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Interface Services](#)

NU_USB_INTF_Get_Active_Alt_Setting_Num

This function returns bAlternateSettingNum of the alternate setting (in alt_setting_out) that is currently active. NU_NOT_PRESENT is returned, if no alternate setting is active on the interface.

Usage

```
STATUS NU_USB_INTF_Get_Active_Alt_Setting_Num (
                                         NU_USB_INTF *cb,
                                         UINT8      *alt_setting_num_out)
```

Arguments

- **cb**
Pointer to the interface control block.
- **alt_setting_num_out**
Pointer to the variable to hold bAlternateSettingNum of the currently active alternate setting.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_NOT_PRESENT**
Indicates that there is no active alternate setting on the interface.

Related Topics

[Nucleus USB Interface Services](#)

NU_USB_INTF_Get_Alt_Setting

This function returns a pointer to the control block of the alternate setting (in alt_setting_out) that corresponds to the specified bAlternateSettingNum.

Usage

```
STATUS NU_USB_INTF_Get_Alt_Setting (NU_USB_INTF      *cb,  
                                   UINT8              alt_setting_num,  
                                   NU_USB_ALT_SETTG **alt_setting_out)
```

Arguments

- cb
Pointer to the interface control block.
- alt_setting_num
bAlternateSettingNum of the desired alternate setting.
- alt_setting_out
Pointer to a memory location to hold pointer to pointer to the control block of the desired alternate setting

Return Values

- NU_SUCCESS
Indicates successful completion of the service.

Related Topics

[Nucleus USB Interface Services](#)

NU_USB_INTF_Get_Cfg

This function returns the pointer to the control block of the associated configuration, in `cfg_out`.

Usage

```
STATUS NU_USB_INTF_Get_Cfg (NU_USB_INTF *cb,  
                           NU_USB_CFG **cfg_out)
```

Arguments

- `cb`
 Pointer to the interface control block.
- `cfg_out`
 Pointer to a memory location to hold the pointer to the control block of the associated configuration.

Return Values

- `NU_SUCCESS`
 Indicates successful completion of the service.

Related Topics

[Nucleus USB Interface Services](#)

NU_USB_INTF_Get_Class

This function returns the bInterfaceClass field of the currently active alternate setting's interface descriptor in bInterfaceClass_out.

Usage

```
STATUS NU_USB_INTF_Get_Class (NU_USB_INTF *cb,  
                             UINT8      *bInterfaceClass_out)
```

Arguments

- **cb**
Pointer to the interface control block.
- **bInterfaceClass_out**
Pointer to the variable to hold bInterfaceClass field of the currently active alternate setting's interface descriptor.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Interface Services](#)

NU_USB_INTF_Get_Desc

This function returns the interface descriptor of the currently active alternate setting's interface descriptor.

Usage

```
STATUS NU_USB_INTF_Get_Desc (NU_USB_INTF      *cb,  
                             NU_USB_INTF_DESC **intf_desc_out)
```

Arguments

- **cb**
Pointer to the interface control block.
- **intf_desc_out**
Pointer to hold the interface descriptor of the currently active alternate setting's interface descriptor.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Interface Services](#)

NU_USB_INTF_Get_Device

This function returns the pointer to the control block of the associated device, in device_out.

Usage

```
STATUS NU_USB_INTF_Get_Device (NU_USB_INTF    *cb,  
                               NU_USB_DEVICE **device_out)
```

Arguments

- **cb**
Pointer to the interface control block.
- **device_out**
Pointer to a memory location to hold the pointer to the control block of the associated device.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Interface Services](#)

NU_USB_INTF_Get_IAD

This function returns a pointer to the associated Interface Association Descriptor (IAD) if it is present, otherwise it returns a null pointer.

Usage

```
STATUS NU_USB_INTF_Get_IAD (NU_USB_INTF *cb,  
                           NU_USB_IAD **iad_out)
```

Arguments

- **cb**
Pointer to NU_USB_INTF control block.
- **iad_out**
Pointer to NU_USB_IAD, containing the interface association descriptor or NU_NULL when the function returns.

Return Values

- **NU_SUCCESS**
Operation completed successfully, 'iad_out' points to a valid interface association descriptor.
- **NU_USB_REJECTED**
Interface association descriptor is not present, 'iad_out' points to NU_NULL.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Interface Services](#)

NU_USB_INTF_Get_Intf_Num

This function returns the bInterfaceNumber field of the currently active alternate setting's interface descriptor, in intf_num_out.

Usage

```
STATUS NU_USB_INTF_Get_Intf_Num (NU_USB_INTF *cb,  
                                UINT8        *intf_num_out)
```

Arguments

- **cb**
Pointer to the interface control block.
- **intf_num_out**
Pointer to the variable to hold the bInterfaceNumber field of the currently active alternate setting's interface descriptor.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Interface Services](#)

NU_USB_INTF_Get_Is_Claimed

This function returns NU_FALSE in is_claimed_out, if no driver currently owns the interface and NU_TRUE otherwise. If a driver currently owns the interface, its pointer is returned in drv_r_out.

Usage

```
STATUS NU_USB_INTF_Get_Is_Claimed (NU_USB_INTF *cb,
                                   BOOLEAN      *is_claimed_out,
                                   NU_USB_DRV_R **drv_r_out)
```

Arguments

- cb
 Pointer to the interface control block.
- is_claimed_out
 Pointer to a variable to hold the ownership status of the interface.
- drv_r_out
 Pointer to a memory location to hold the pointer to the driver control block that currently owns the interface.

Return Values

- NU_SUCCESS
 Indicates successful completion of the service.

Related Topics

[Nucleus USB Interface Services](#)

NU_USB_INTF_Get_Num_Alt_Settings

The count of the alternate settings that this interface has is returned by this function, in `number_alt_settings_out`.

Usage

```
STATUS NU_USB_INTF_Get_Num_Alt_Settings (
                                         NU_USB_INTF *cb,
                                         UINT8      number_alt_settings_out)
```

Arguments

- `cb`
Pointer to the interface control block.
- `number_alt_settings_out`
Number of alternate settings this interface has. It will be in the range of 1 - `USB_MAX_ALT_SETTINGS`.

Return Values

- `NU_SUCCESS`
Indicates successful completion of the service.

Related Topics

[Nucleus USB Interface Services](#)

NU_USB_INTF_Get_Protocol

This function returns the bInterfaceProtocol field of the currently active alternate setting's interface descriptor in bInterfaceProtocol_out.

Usage

```
STATUS NU_USB_INTF_Get_Protocol (NU_USB_INTF *cb,
                                UINT8      *bInterfaceProtocol_out)
```

Arguments

- **cb**
Pointer to the interface control block.
- **bInterfaceProtocol_out**
Pointer to the variable to hold the bInterfaceProtocol field of the currently active alternate setting's interface descriptor.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Interface Services](#)

NU_USB_INTF_Get_String

This function copies the interface string corresponding to the currently active alternate setting, in to string_out.

Usage

```
STATUS NU_USB_INTF_Get_String (NU_USB_INTF *cb,  
                              CHAR          *string_out)
```

Arguments

- **cb**
Pointer to the interface control block.
- **string_out**
Pointer to the user supplied array of characters of size NU_USB_MAX_STRING_LEN.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_NOT_PRESENT**
Indicates that the interface string number cannot be found in the device's set of string descriptors.

Description

English is assumed to be the Unicode of the configuration string descriptor. If the Unicode is not English, refer to the [NU_USB_INTF_Get_String_Desc](#) for further information on how to retrieve a displayable string.

Related Topics

[Nucleus USB Interface Services](#)

NU_USB_INTF_Get_String_Desc

This function gets pointer to the string descriptor (into string_desc_out) for the interface string number (iInterface field of interface descriptor) of the currently active alternate setting.

Usage

```
STATUS NU_USB_INTF_Get_String_Desc (NU_USB_INTF      *cb,
                                   UINT16             wLangId,
                                   NU_USB_STRING_DESC **string_desc_out)
```

Arguments

- **cb**
Pointer to the interface control block.
- **wLangId**
2-byte Unicode language ID for the string.
- **string_desc_out**
Pointer to the memory location to hold pointer to string descriptor structure.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_NOT_PRESENT**
Indicates that the interface string number cannot be found in the device's set of string descriptors.

Description

If the Unicode used for the string descriptors is English, [NU_USB_INTF_Get_String](#) can be used to get the ASCII string directly. For other Unicodes, the [NU_USB_INTF_Get_String_Desc](#) function will help retrieve the descriptor so that the reader can write routines to convert from their Unicode format to the ASCII string or such other display string notations.

Related Topics

[Nucleus USB Interface Services](#)

NU_USB_INTF_Get_String_Num

This function returns the `iInterface` field of the interface descriptor of the currently active alternate setting, in `string_num_out`. If `string_num_out` is 0, there is no interface string descriptor associated with this alternate setting.

Usage

```
STATUS NU_USB_INTF_Get_String_Num (NU_USB_INTF *cb,  
                                  UINT8         *string_num_out)
```

Arguments

- `cb`
Pointer to the interface control block.
- `string_num_out`
Pointer to the variable to hold the `iInterface` field of the interface descriptor.

Return Values

- `NU_SUCCESS`
Indicates successful completion of the service.

Related Topics

[Nucleus USB Interface Services](#)

NU_USB_INTF_Get_SubClass

This function returns the bInterfaceSubClass field of the currently active alternate setting's interface descriptor, in bInterfaceSubClass_out.

Usage

```
STATUS NU_USB_INTF_Get_SubClass (NU_USB_INTF *cb,
                                UINT8      *bInterfaceSubClass_out)
```

Arguments

- **cb**
Pointer to the interface control block.
- **bInterfaceSubClass_out**
Pointer to the variable to hold the bInterfaceSubClass field of the currently active alternate setting's interface descriptor.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Interface Services](#)

NU_USB_INTF_Release

This function releases the ownership of the interface.

Usage

```
STATUS NU_USB_INTF_Release (NU_USB_INTF *cb,  
                             NU_USB_DRV *drv)
```

Arguments

- **cb**
Pointer to the interface control block.
- **drv**
Pointer to the control block of the driver that currently owns the interface.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Description

A USB interface driver that currently owns the interface is only expected to make this function call. Once ownership is released, any other driver may claim ownership of the interface. The delete function of a class driver typically calls this function for all owned interfaces.

Related Topics

[Nucleus USB Interface Services](#)

NU_USB_INTF_Set_Interface

This function makes the specified alternate setting as the active alternate setting of the interface.

Usage

```
STATUS NU_USB_INTF_Set_Interface (NU_USB_INTF *cb,
                                UINT8      alt_setting_num)
```

Arguments

- **cb**
Pointer to the interface control block.
- **alt_setting_num**
bAlternateSettingNum of the alternate setting that is to be made active.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_USB_NO_BANDWIDTH**
Indicates that the device's bandwidth requirements of the specified configuration cannot be met.
- **NU_INVALID_SEMAPHORE**
Indicates that one of the internal semaphore pointers is invalid.
- **NU_SEMAPHORE_DELETED**
Indicates that one of the internal semaphores was deleted.
- **NU_UNAVAILABLE**
Indicates that one of the internal semaphores is unavailable.
- **NU_INVALID_SUSPEND**
Indicates that this API is called from a non-task thread.
- **NU_USB_INVLD_ARG**
Indicates that one or more arguments passed to this function are invalid.

Description

The resources necessary in the controller hardware are allocated and the required bandwidth on the bus is reserved by this function.

This function should be used only by class drivers on the host software. It is the host's prerogative to select and activate an alternate setting on an interface.

Related Topics

[Nucleus USB Interface Services](#)

Nucleus USB IRP Services

This section provides a detailed reference of the following Nucleus USB IRP Services:

- [NU_USB_IRP_Create](#)
- [NU_USB_IRP_Delete](#)
- [NU_USB_IRP_Get_Accept_Short_Packets](#)
- [NU_USB_IRP_Get_Actual_Length](#)
- [NU_USB_IRP_Get_Buffer_Type_Cachable](#)
- [NU_USB_IRP_Get_Callback](#)
- [NU_USB_IRP_Get_Context](#)
- [NU_USB_IRP_Get_Data](#)
- [NU_USB_IRP_Get_Interval](#)
- [NU_USB_IRP_Get_Length](#)
- [NU_USB_IRP_Get_Pipe](#)
- [NU_USB_IRP_Get_Status](#)
- [NU_USB_IRP_Get_Use_Empty_Pkt](#)
- [NU_USB_IRP_Set_Accept_Short_Pkt](#)
- [NU_USB_IRP_Set_Actual_Length](#)
- [NU_USB_IRP_Set_Buffer_Type_Cachable](#)
- [NU_USB_IRP_Set_Callback](#)
- [NU_USB_IRP_Set_Context](#)
- [NU_USB_IRP_Set_Data](#)
- [NU_USB_IRP_Set_Interval](#)
- [NU_USB_IRP_Set_Length](#)
- [NU_USB_IRP_Set_Pipe](#)
- [NU_USB_IRP_Set_Status](#)
- [NU_USB_IRP_Set_Use_Empty_Pkt](#)

Related Topics

[USB Host and Functions](#)

NU_USB_IRP_Create

This function initializes the IRP with the specified values.

Usage

```
STATUS NU_USB_IRP_Create (NU_USB_IRP      *cb,  
                          UINT32          length,  
                          UINT8           *data,  
                          BOOLEAN         accept_short_packets,  
                          BOOLEAN         use_empty_packets,  
                          NU_USB_IRP_CALLBACK callback,  
                          VOID            *context,  
                          UINT32          interval)
```

Arguments

- **cb**
Pointer to IRP control block.
- **length**
Length of data transfer associated with IRP. This length need not be a multiple of four. (see the discussion below for ‘data’). But it should necessarily be equal to or less than the size of the data buffer pointed to by ‘data’.
- **data**
Pointer to memory location, which contains the data to be transmitted or the location that would store the received data. The memory location has to be a multiple of four and the size of the memory buffer should also be a multiple of four. This is necessary since most controllers access from a four-byte boundary and four bytes at a time.
- **accept_short_packets**
Flag specifying to accept short packets in this transmission. If the data received from an endpoint is less than the maximum packet size, its called as data under run error. This flag if set, causes data under run to be ignored by the hardware.
- **use_empty_packets**
Flag specifying to use empty packets as end of transfer markers. If the length of the data to be transmitted is an exact multiple of the endpoint’s maximum packet size, then this flag determines, if an empty packet is to be sent to mark the end of the transfer.
- **callback**
Pointer to a function invoked when the IRP transfer attempt is completed. This callback function should not make any calls that require NU_SUSPEND option and also it should not involve any CPU intensive processing.
- **context**
Context information that would help the callback function in identifying the completed IRP uniquely. This ‘context’ is passed to the IRP callback function upon IRP completion.

- interval
Polling interval for this IRP. Irrelevant for non-periodic IRPs.

Return Values

- NU_SUCCESS
Indicates successful completion of the service.

Related Topics

[Nucleus USB IRP Services](#)

NU_USB_IRP_Delete

This function releases resources that were allocated for the IRP in NU_USB_IRP_Create.

Usage

```
STATUS NU_USB_IRP_Delete (NU_USB_IRP *cb)
```

Arguments

- **cb**
Pointer to the IRP control block or its derivative.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB IRP Services](#)

NU_USB_IRP_Get_Accept_Short_Packets

This function returns if the associated IRP accepts short packets.

Usage

```
STATUS NU_USB_IRP_Get_Accept_Short_Packets (  
    NU_USB_IRP *cb,  
    BOOLEAN    *accept_short_packets_out)
```

Arguments

- **cb**
Pointer to IRP control block.
- **accept_short_packets_out**
Pointer to a variable to hold returned value of accept short packets flag of the IRP.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB IRP Services](#)

NU_USB_IRP_Get_Actual_Length

This function returns the actual length of the data transferred by the IRP in length_out.

Usage

```
STATUS NU_USB_IRP_Get_Actual_Length (NU_USB_IRP *cb,  
                                     UINT32      *length_out)
```

Arguments

- **cb**
Pointer to the IRP control block.
- **length_out**
Pointer to a variable to hold the returned actual length.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB IRP Services](#)

NU_USB_IRP_Get_Buffer_Type_Cachable

This function returns the buffer type of an IRP as cacheable or non cacheable.

Usage

```
STATUS NU_USB_IRP_Get_Buffer_Type_Cachable( NU_USB_IRP *cb,  
                                             BOOLEAN    buffer_type)
```

Arguments

- **cb**
Pointer to IRP control block.
- **buffer_type**
Parameter to hold type of buffer. NU_TRUE for cacheable buffer and NU_FALSE for non cacheable buffer.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB IRP Services](#)

NU_USB_IRP_Get_Callback

This function returns the callback function pointer associated with the IRP in `callback_out`.

Usage

```
STATUS NU_USB_IRP_Get_Callback (NU_USB_IRP          *cb,  
                               NU_USB_IRP_CALLBACK *callback_out)
```

Arguments

- `cb`
Pointer to the IRP control block.
- `callback_out`
Pointer to a memory location to hold the pointer to callback function associated with the IRP.

Return Values

- `NU_SUCCESS`
Indicates successful completion of the service.

Related Topics

[Nucleus USB IRP Services](#)

NU_USB_IRP_Get_Context

This function returns the context pointer associated with the IRP in context_out.

Usage

```
STATUS NU_USB_IRP_Get_Context (NU_USB_IRP *cb,  
                              VOID **context_out)
```

Arguments

- **cb**
Pointer to the IRP control block.
- **context_out**
Pointer to a memory location to hold the pointer to context associated with the IRP.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB IRP Services](#)

NU_USB_IRP_Get_Data

This function returns the data pointer associated with IRP in data_out.

Usage

```
STATUS NU_USB_IRP_Get_Data (NU_USB_IRP *cb,  
                           UINT8      **data_out)
```

Arguments

- **cb**
Pointer to IRP control block.
- **data_out**
Pointer to a memory location to hold the pointer to data associated with the IRP.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB IRP Services](#)

NU_USB_IRP_Get_Interval

This function returns the polling interval of the data transfer associated with the IRP in `interval_out`.

Usage

```
STATUS NU_USB_IRP_Get_Interval (NU_USB_IRP *cb,
                               UINT32      *interval_out)
```

Arguments

- `cb`
Pointer to the IRP control block.
- `interval_out`
Pointer to a variable to hold the interval field associated with the IRP.

Return Values

- `NU_SUCCESS`
Indicates successful completion of the service.

Related Topics

[Nucleus USB IRP Services](#)

NU_USB_IRP_Get_Length

This function returns the length of the data transfer associated with the IRP in length_out.

Usage

```
STATUS NU_USB_IRP_Get_Length (NU_USB_IRP *cb,  
                              UINT32      *length_out)
```

Arguments

- **cb**
Pointer to IRP control block.
- **length_out**
Pointer to a variable to hold the length field of the associated IRP.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB IRP Services](#)

NU_USB_IRP_Get_Pipe

This function returns pointer to the control block of the pipe associated with the IRP, in pipe_out.

Usage

```
STATUS NU_USB_IRP_Get_Pipe (NU_USB_IRP  *cb,  
                           NU_USB_PIPE **pipe_out)
```

Arguments

- cb
Pointer to the IRP control block.
- pipe_out
Pointer to a memory location to hold the pointer to control block of the pipe associated with the IRP.

Return Values

- NU_SUCCESS
Indicates successful completion of the service.

Related Topics

[Nucleus USB IRP Services](#)

NU_USB_IRP_Get_Status

This function returns the status of the data transfer associated with the IRP in `status_out`. The callback function is expected to invoke this function to know the status of the IRP transfer attempt.

Usage

```
STATUS NU_USB_IRP_Get_Status (NU_USB_IRP *cb,  
                             STATUS      *status_out)
```

Arguments

- `cb`
Pointer to the IRP control block.
- `status_out`
Pointer to a variable to hold the returned status of IRP.

Return Values

- `NU_SUCCESS`
Indicates successful completion of the service.

Related Topics

[Nucleus USB IRP Services](#)

NU_USB_IRP_Get_Use_Empty_Pkt

This function returns if the associated IRP uses an empty packet to mark the end of transmission.

Usage

```
STATUS NU_USB_IRP_Get_Use_Empty_Pkt (NU_USB_IRP *cb,  
                                     BOOLEAN *use_empty_packet_out)
```

Arguments

- **cb**
Pointer to the IRP control block.
- **use_empty_packet_out**
Pointer to a variable to hold returned value of use empty packet flag of the IRP.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB IRP Services](#)

NU_USB_IRP_Set_Accept_Short_Pkt

This function sets the accept short packets flag of the IRP.

Usage

```
STATUS NU_USB_IRP_Set_Accept_Short_Pkt (NU_USB_IRP *cb,  
                                         BOOLEAN    accept_short_packets)
```

Arguments

- **cb**
Pointer to the IRP control block.
- **accept_short_packets**
Flag to be set.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Description

If the data received from an endpoint is less than the maximum packet size, its called as data under run error. This flag if set, causes data under run to be ignored by the hardware.

Related Topics

[Nucleus USB IRP Services](#)

NU_USB_IRP_Set_Actual_Length

This function sets the actual length of the data transfer associated with an IRP. The hardware driver invokes this function after the completion of the IRP transfer attempt to set the actual data length transferred.

Usage

```
STATUS NU_USB_IRP_Set_Actual_Length (NU_USB_IRP *cb,  
                                     UINT32      length)
```

Arguments

- **cb**
Pointer to the IRP control block.
- **length**
Actual length to be set.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB IRP Services](#)

NU_USB_IRP_Set_Buffer_Type_Cachable

This function sets the buffer type of an IRP as cacheable or non cacheable.

Usage

```
STATUS NU_USB_IRP_Set_Buffer_Type_Cachable(NU_USB_IRP *cb,  
                                           BOOLEAN    buffer_type)
```

Arguments

- **cb**
Pointer to IRP control block.
- **buffer_type**
Buffer type to be set. Set this to NU_TRUE for cacheable buffer, otherwise set this to NU_FALSE for non cacheable buffer.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB IRP Services](#)

NU_USB_IRP_Set_Callback

This function sets the callback function pointer associated with the IRP.

Usage

```
STATUS NU_USB_IRP_Set_Callback (NU_USB_IRP          *cb,  
                               NU_USB_IRP_CALLBACK callback)
```

Arguments

- **cb**
Pointer to the IRP control block.
- **callback**
Pointer to the callback function.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Description

This function is invoked whenever the hardware completes the IRP transfer attempt. This callback function should not make any calls that require the NU_SUSPEND option and it should not involve any CPU intensive processing.

Related Topics

[Nucleus USB IRP Services](#)

NU_USB_IRP_Set_Context

This function sets the context data associated with the IRP.

Usage

```
STATUS NU_USB_IRP_Set_Context (NU_USB_IRP *cb,  
                              VOID *context)
```

Arguments

- **cb**
Pointer to the IRP control block.
- **context**
Pointer to a location to store context.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Description

The 'context' is passed to the callback function of the IRP whenever it is invoked by the hardware driver. The 'context' can be used by the callback to identify any private information associated with the IRP.

Related Topics

[Nucleus USB IRP Services](#)

NU_USB_IRP_Set_Data

This function sets the pointer to the data buffer, which contains the data to be transmitted or the pointer to the data buffer at which the received data has to be stored for the IRP.

Usage

```
STATUS NU_USB_IRP_Set_Data (NU_USB_IRP *cb,  
                           UINT8      *data)
```

Arguments

- **cb**
Pointer to the IRP control block.
- **data**
Pointer to a location to store data.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB IRP Services](#)

NU_USB_IRP_Set_Interval

This function sets the interval associated with the IRP. This is irrelevant if the IRP is meant for non-periodic transfers.

Usage

```
STATUS NU_USB_IRP_Set_Interval (NU_USB_IRP *cb,  
                                UINT32      interval)
```

Arguments

- **cb**
Pointer to the IRP control block.
- **interval**
Interval to be set.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB IRP Services](#)

NU_USB_IRP_Set_Length

This function sets the length of the data transfer associated with the IRP.

Usage

```
STATUS NU_USB_IRP_Set_Length (NU_USB_IRP *cb,
                              UINT32      length)
```

Arguments

- **cb**
Pointer to the IRP control block.
- **length**
Length to be set.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Description

This length need not be a multiple of four, though the size of the data buffer has to be a multiple of four. The length should be equal to or less than the size of the data buffer associated with the IRP.

Related Topics

[Nucleus USB IRP Services](#)

NU_USB_IRP_Set_Pipe

This function sets the pipe of the IRP. This pipe identifies the endpoint and the destination for the IRP.

Usage

```
STATUS NU_USB_IRP_Set_Pipe (NU_USB_IRP  *cb,  
                           NU_USB_PIPE *pipe)
```

Arguments

- **cb**
Pointer to the IRP control block.
- **pipe**
Pointer to pipe control block to be set.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB IRP Services](#)

NU_USB_IRP_Set_Status

This function sets the status of the data transfer associated with the IRP. This function is only invoked by the hardware driver to report the status of the IRP transfer attempt.

Usage

```
STATUS NU_USB_IRP_Set_Status (NU_USB_IRP *cb,
                             STATUS      status)
```

Arguments

- **cb**
Pointer to the IRP control block.
- **status**
Status to be set.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_USB_NO_ERR**
IRP transfer is successful.
- **NU_USB_CRC_ERR**
Transfer encountered CRC error.
- **NU_USB_BITSTUFF_ERR**
Transfer encountered bit stuffing error.
- **NU_USB_TOGGLE_ERR**
Transfer encountered data0/data1 toggle error.
- **NU_USB_STALL_ERR**
IRP received a stall handshake.
- **NU_USB_NO_RESPONSE**
No response received for the token.
- **NU_USB_INVLD_PID**
Packet ID sent out by the hardware is invalid.
- **NU_USB_UNEXPECTED_PID**
Unexpected packet id received.
- **NU_USB_DATA_OVERRUN**
The amount of data returned by the endpoint exceeded either the size of the maximum data packet allowed from the endpoint or the remaining buffer size.

- **NU_USB_DATA_UNDERRUN**
The endpoint returned less than MaximumPacketSize and that amount was not sufficient to fill the specified buffer. If accept short packets is true for the IRP, data under run is ignored.
- **NU_USB_BFR_OVERRUN**
Hardware received data from endpoint faster than it could be written to system memory.
- **NU_USB_BFR_UNDERRUN**
Hardware could not retrieve data from system memory fast enough to keep up with data USB data rate.
- **NU_USB_NOT_ACCESSED**
Hardware did not attempt the transfer.
- **NU_USB_EP_HALTED**
Endpoint is halted due to unknown error.
- **NU_USB_IRP_CANCELLED**
Endpoint is closed while the transfer is pending or some one invoked flush pipe while this IRP is pending transfer.
- **NU_USB_UNKNOWN_ERR**
Unknown error during the transfer.

Related Topics

[Nucleus USB IRP Services](#)

NU_USB_IRP_Set_Use_Empty_Pkt

This function sets the use empty packet flag of an IRP.

Usage

```
STATUS NU_USB_IRP_Set_Use_Empty_Pkt (NU_USB_IRP *cb,  
                                     BOOLEAN use_empty_packet)
```

Arguments

- **cb**
Pointer to the IRP control block.
- **use_empty_packet**
Flag to be set.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Description

If the length of the data to be transmitted is an exact multiple of the endpoint's maximum packet size, then this flag determines if an empty packet is to be sent to mark the end of the transfer.

Related Topics

[Nucleus USB IRP Services](#)

USB Memory Component

This section provides a detailed reference of the following USB Memory Component:

- [USB_Allocate_Aligned_Memory](#)
- [USB_Allocate_Memory](#)
- [USB_Allocate_Object](#)
- [USB_Deallocate_Memory](#)

Related Topics

[USB Host and Functions](#)

USB_Allocate_Aligned_Memory

This function allocates a requested size of aligned memory block from USB memory pools. Cacheable or non cacheable memory can be allocated by passing the appropriate value in the 'mem_type' argument.

Usage

```
STATUS usb_allocate_aligned_memory (UINT8  mem_type,
                                   UINT32 size,
                                   UINT32 alignment,
                                   VOID   **ptr_out)
```

Arguments

- **mem_type**
 Type of requested memory (cacheable or non cacheable). Use the following flags:
 USB_MEM_TYPE_CACHED
 Use this flag to allocate memory from USB cached memory pool.
 USB_MEM_TYPE_UNCACHED
 Use this flag to allocate memory from USB uncached memory pool.
- **size**
 Size of requested memory block in bytes.
- **alignment**
 Required alignment of the requested memory block.
- **ptr_out**
 Double pointer to the address of the allocated memory block when the function returns.

Return Values

- **NU_SUCCESS**
 Operation completed successfully, 'ptr_out' contains a valid memory block address when the function returns.
- **NU_USB_INVLD_MEM_POOL**
 USB memory pools are not initialized.
- **NU_USB_INVLD_ARG**
 One or more input arguments are invalid.

Related Topics

[USB Memory Component](#)

USB_Allocate_Memory

This function allocates a requested size of memory block from the USB memory pools. Cacheable or noncacheable memory can be allocated by passing the appropriate value in the 'mem_type' argument.

Usage

```
STATUS usb_allocate_memory (UINT8  mem_type,  
                           UINT32 size,  
                           VOID   **ptr_out)
```

Arguments

- **mem_type**
Type of requested memory (cacheable or non cacheable). Use the following flags:
USB_MEM_TYPE_CACHED
Use this flag to allocate memory from USB cached memory pool.
USB_MEM_TYPE_UNCACHED
Use this flag to allocate memory from USB uncached memory pool.
- **size**
Size of requested memory block in bytes.
- **ptr_out**
Double pointer to the address of the allocated memory block when the function returns.

Return Values

- **NU_SUCCESS**
Operation completed successfully, 'ptr_out' contains a valid memory block address when the function returns.
- **NU_USB_INVLD_MEM_POOL**
USB memory pools are not initialized.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[USB Memory Component](#)

USB_Allocate_Object

This function allocates a requested size of a memory block from the USB cached memory pool. It is recommended to use this call to allocate control blocks or any other objects.

Usage

```
STATUS usb_allocate_object (UINT32 size,  
                           VOID    **ptr_out)
```

Arguments

- **size**
Size of requested memory block in bytes.
- **ptr_out**
Double pointer to the address of the allocated memory block when the function returns.

Return Values

- **NU_SUCCESS**
Operation completed successfully, 'ptr_out' contains a valid memory block address when the function returns.
- **NU_USB_INVLD_MEM_POOL**
USB memory pools are not initialized.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[USB Memory Component](#)

USB_Deallocate_Memory

This function frees an allocated block of memory. Once a memory block is freed, its control goes back to the system and the system can allocate it to another caller.

Usage

```
STATUS usb_deallocate_memory (VOID *mem_ptr)
```

Arguments

- **mem_ptr**
Address of a memory block to be deallocated.

Return Values

- **NU_SUCCESS**
Operation completed successfully, the memory block pointed by 'mem_ptr' is now free and can be allocated to another caller.
- **NU_USB_INVLD_MEM_POOL**
USB memory pools are not initialized.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[USB Memory Component](#)

Nucleus USB Pipe Services

This section provides a detailed reference of the following Nucleus USB Pipe Services:

- [NU_USB_PIPE_Flush](#)
- [NU_USB_PIPE_Get_Device](#)
- [NU_USB_PIPE_Get_Endp](#)
- [NU_USB_PIPE_Get_Is_Active](#)
- [NU_USB_PIPE_Get_Is_Stalled](#)
- [NU_USB_PIPE_Set_Device](#)
- [NU_USB_PIPE_Set_Endp](#)
- [NU_USB_PIPE_Set_Is_Active](#)
- [NU_USB_PIPE_Stall](#)
- [NU_USB_PIPE_Submit_IRP](#)
- [NU_USB_PIPE_Unstall](#)

Related Topics

[USB Host and Functions](#)

NU_USB_PIPE_Flush

This function flushes the given pipe. This function initiates the deletion of all work items pending the hardware's work schedule for the specified pipe.

Usage

```
STATUS NU_USB_PIPE_Flush (NU_USB_PIPE *cb)
```

Arguments

- **cb**
Pointer to the pipe control block.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_USB_INVLD_ARG**
Indicates that the pipe control block is invalid.

Description

As and when the work items of an IRP are deleted from the work schedule of the hardware, the corresponding IRP's status is set to **NU_USB_IRP_CANCELLED** and its callback is then invoked.

Related Topics

[Nucleus USB Pipe Services](#)

NU_USB_PIPE_Get_Device

This function returns the NU_USB_DEVICE control block pointer with which this pipe is associated.

Usage

```
STATUS NU_USB_PIPE_Get_Device (NU_USB_PIPE    *cb,
                              NU_USB_DEVICE **device_out)
```

Arguments

- **cb**
 Pointer to the pipe control block.
- **device_out**
 Pointer to a variable to hold pointer to device control block.

Return Values

- **NU_SUCCESS**
 Indicates successful completion of the service.
- **NU_USB_INVLD_ARG**
 Indicates that the pipe control block is invalid.

Related Topics

[Nucleus USB Pipe Services](#)

NU_USB_PIPE_Get_Endp

This function returns the NU_USB_ENDP control block pointer with which this pipe is associated.

Usage

```
STATUS NU_USB_PIPE_Get_Endp (NU_USB_PIPE *cb,  
                             NU_USB_ENDP **endp_out)
```

Arguments

- **cb**
Pointer to the pipe control block.
- **endp_out**
Pointer to a variable to hold pointer to endpoint control block.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_USB_INVLD_ARG**
Indicates that the pipe control block is invalid.

Related Topics

[Nucleus USB Pipe Services](#)

NU_USB_PIPE_Get_Is_Active

This function returns if a given pipe is active or in a stalled state.

Usage

```
STATUS NU_USB_PIPE_Get_Is_Active (NU_USB_PIPE *cb,  
                                BOOLEAN      *is_active_out)
```

Arguments

- **cb**
Pointer to the pipe control block.
- **is_active_out**
Pointer to a variable to store returned active flag of pipe.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_USB_INVLD_ARG**
Indicates that the pipe control is invalid.

Related Topics

[Nucleus USB Pipe Services](#)

NU_USB_PIPE_Get_Is_Stalled

This function returns if a given pipe is active or in a stalled state.

Usage

```
STATUS NU_USB_PIPE_Get_Is_Stalled (NU_USB_PIPE *cb,  
                                   BOOLEAN      *is_stalled_out)
```

Arguments

- **cb**
Pointer to the pipe control block.
- **is_stalled_out**
Pointer to a variable to store returned stalled state of the pipe.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_USB_INVLD_ARG**
Indicates that the pipe control is invalid.

Related Topics

[Nucleus USB Pipe Services](#)

NU_USB_PIPE_Set_Device

This function sets the pointer to the control block of the device with which it is associated.

Usage

```
STATUS NU_USB_PIPE_Set_Device (NU_USB_PIPE    *cb,  
                               NU_USB_DEVICE  *device)
```

Arguments

- **cb**
Pointer to the pipe control block.
- **device**
Pointer to device control block.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_USB_INVLD_ARG**
Indicates that the pipe control block is invalid.

Related Topics

[Nucleus USB Pipe Services](#)

NU_USB_PIPE_Set_Endp

This function sets the NU_USB_ENDP control block pointer of this pipe.

Usage

```
STATUS NU_USB_PIPE_Set_Endp (NU_USB_PIPE *cb,  
                             NU_USB_ENDP *endp)
```

Arguments

- **cb**
Pointer to the pipe control block.
- **endp**
Pointer to endpoint control block.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_USB_INVLD_ARG**
Indicates that the pipe control block is invalid.

Related Topics

[Nucleus USB Pipe Services](#)

NU_USB_PIPE_Set_Is_Active

This function sets the pipe in the active or stalled state.

Usage

```
STATUS NU_USB_PIPE_Set_Is_Active (NU_USB_PIPE *cb,  
                                BOOLEAN      is_active)
```

Arguments

- **cb**
Pointer to the pipe control block.
- **is_active**
Flag to be set.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_USB_INVLD_ARG**
Indicates that the pipe control block is invalid.

Related Topics

[Nucleus USB Pipe Services](#)

NU_USB_PIPE_Stall

This function sets the endpoint associated with this pipe in a stall state.

Usage

```
STATUS NU_USB_PIPE_Stall (NU_USB_PIPE *cb)
```

Arguments

- **cb**
Pointer to the pipe control block.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_USB_INVLD_ARG**
Indicates that the pipe control is invalid.

Description

On function software, this results in stall handshakes being sent and on the host software, the pipe ceases to be a part of the hardware's work schedule.

Related Topics

[Nucleus USB Pipe Services](#)

NU_USB_PIPE_Submit_IRP

This function submits the given IRP on a pipe.

Usage

```
STATUS NU_USB_PIPE_Submit_IRP (NU_USB_PIPE *cb,  
                               NU_USB_IRP *irp)
```

Arguments

- **cb**
Pointer to the pipe control block.
- **irp**
Pointer to the IRP control block to be submitted.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_USB_INVLD_ARG**
Indicates that one or more args passed to this function are invalid.
- **NU_NOT_PRESENT**
Indicates that the pipe is invalid.
- **NU_NO_MEMORY**
Indicates failure of memory allocation.

Description

The IRP is translated into work items that are attempted by the hardware as per its work schedule. Once all of IRP's work items are complete, the hardware invokes the callback specified in the IRP to notify IRP transfer completion. Hence, successful completion of [NU_USB_PIPE_Submit_IRP](#) does not mean successful completion of the IRP transfer, it only means successful translation of the IRP into hardware work items. The IRP and its contents should not be changed until its callback is invoked. The completion status of the IRP can be known by invoking [NU_USB_IRP_Get_Status](#) in the callback function. The number of bytes transferred can also be known using [NU_USB_IRP_Get_Actual_Length](#).

Related Topics

[Nucleus USB Pipe Services](#)

NU_USB_PIPE_Unstall

This function clears an endpoint associated with this pipe from a stalled state.

Usage

```
STATUS NU_USB_PIPE_Unstall (NU_USB_PIPE *cb)
```

Arguments

- **cb**
Pointer to the pipe control block.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_USB_INVLD_ARG**
Indicates that the pipe control is invalid.

Description

On the function software, the function controller stops sending stall responses for the associated endpoint. On the host software, the pipe again becomes part of the hardware's work schedule.

Related Topics

[Nucleus USB Pipe Services](#)

Nucleus USB Power Management Services

This section provides a detailed reference of the following Nucleus USB Power Management Services:

- [NU_USB_PMG_Get_Dev_Pwr_Attrib](#)
- [NU_USB_PMG_Update_Dev_LTM_Enable](#)
- [NU_USB_PMG_Update_Dev_Pwr_Src](#)
- [NU_USB_PMG_Update_Dev_U1_Enable](#)
- [NU_USB_PMG_Update_Dev_U2_Enable](#)
- [NU_USB_PMG_Update_Intf_Pwr_Attrib](#)
- [NU_USB_PMG_Set_Dev_Pwr_Attrib](#)
- [NU_USB_PMG_Set_Link_State](#)

Related Topics

[USB Host and Functions](#)

NU_USB_PMG_Get_Dev_Pwr_Attrib

This API retrieves the power attributes of a NU_USB_DEVICE.

Usage

```
STATUS NU_USB_PMG_Get_Dev_Pwr_Attrib (NU_USB_DEVICE *cb,  
                                       NU_USB_DEV_PW_ATTRIB **pwr_attrib)
```

Arguments

- **cb**
Address of NU_USB_DEVICE control block.
- **pwr_attrib**
Double pointer to the NU_USB_PWR_ATTRIB structure. When the function returns, it points to the NU_USB_PWR_ATTRIB structure defined in NU_USB_DEVICE control block.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Power Management Services](#)

NU_USB_PMG_Update_Dev_LTM_Enable

This API is used for updating the value of LTM_Enable capability in the NU_USB_DEV_PW_ATTRIB control block.

Usage

```
STATUS NU_USB_PMG_Update_Dev_LTM_Enable (NU_USB_DEVICE *cb,  
                                          BOOLEAN      ltm_enable)
```

Arguments

- **cb**
Address of NU_USB_DEVICE control block.
- **ltm_enable**
This can be set to one of the following values:
 - NU_TRUE
Enable the LTM capability of the device
 - NU_FALSE
Disable the LTM capability of the device

Return Values

- **NU_SUCCESS**
Operation completed successfully, the “ltm_enable” value is set in the NU_USB_DEV_PW_ATTRIB control block.
- **NU_USB_INVLD_ARG**
One or more input arguments invalid.

Related Topics

[Nucleus USB Power Management Services](#)

NU_USB_PMG_Update_Dev_Pwr_Src

This API is used for updating the value of the power source in NU_USB_DEV_PW_ATTRIB control block.

Usage

```
STATUS NU_USB_PMG_Update_Dev_Pwr_Src (NU_USB_DEVICE *cb,  
                                      BOOLEAN          self_powered)
```

Arguments

- **cb**
Address of NU_USB_DEVICE control block.
- **self_powered**
This can be set to one of the following values:

NU_TRUE
USB_SELF_POWERED
NU_FALSE
USB_BUS_POWERED

Return Values

- **NU_SUCCESS**
Operation completed successfully, the power source value is set.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Power Management Services](#)

NU_USB_PMG_Update_Dev_U1_Enable

This API is used for updating the value of U1_Enable capability in the NU_USB_DEV_PW_ATTRIB control block.

Usage

```
STATUS NU_USB_PMG_Update_Dev_U1_Enable (NU_USB_DEVICE *cb,  
                                         BOOLEAN      u1_enable)
```

Arguments

- **cb**
Address of NU_USB_DEVICE control block.
- **u1_enable**
This can be set to one of the following values:
 - NU_TRUE
Enable device capability to initiate u1 entry.
 - NU_FALSE
Disable device capability to initiate u1 entry.

Return Values

- **NU_SUCCESS**
Operation completed successfully, the value of u1_enable is set in the NU_USB_DEV_PW_ATTRIB control block.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Power Management Services](#)

NU_USB_PMG_Update_Dev_U2_Enable

This API is used for updating the value of u2_enable capability in the NU_USB_DEV_PW_ATTRIB control block.

Usage

```
STATUS NU_USB_PMG_Update_Dev_U2_Enable (NU_USB_DEVICE *cb,  
                                         BOOLEAN      u2_enable)
```

Arguments

- **cb**
Address of NU_USB_DEVICE control block.
- **u2_enable**
This can be set to one of the following values:
 - NU_TRUE
Enable device capability to initiate u2 entry.
 - NU_FALSE
Disable device capability to initiate u2 entry.

Return Values

- **NU_SUCCESS**
Operation completed successfully, the value of u2_enable is set in the NU_USB_DEV_PW_ATTRIB control block.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Power Management Services](#)

NU_USB_PMG_Update_Intf_Pwr_Attrib

This API is used for updating the power attributes of a NU_USB_INTF.

Usage

```
STATUS NU_USB_PMG_Update_Intf_Pwr_Attrib (  
    NU_USB_DEVICE *cb,  
    UINT8 intf_number,  
    NU_USB_INTF_PW_ATTRIB *pwr_attrib)
```

Arguments

- **cb**
Address of NU_USB_DEVICE control block.
- **intf_number**
Interface number.
- **pwr_attrib**
Pointer to the NU_USB_INTF_PW_ATTRIB structure containing the values to set in the NU_USB_INTERFACE structure.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Power Management Services](#)

NU_USB_PMG_Set_Dev_Pwr_Attrib

This API is used for setting the power attributes of a NU_USB_DEVICE.

Usage

```
STATUS NU_USB_PMG_Set_Dev_Pwr_Attrib (NU_USB_DEVICE      *cb,  
                                     NU_USB_PWR_ATTRIB *pwr_attrib)
```

Arguments

- **cb**
Address of NU_USB_DEVICE control block.
- **pwr_attrib**
Pointer to the NU_USB_INTF_PW_ATTRIB structure containing the values to set in the NU_USB_INTERFACE structure.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Power Management Services](#)

NU_USB_PMG_Set_Link_State

This API updates the 'link_state' variable in the NU_USB_DEVICE control block and commands hardware to select a suitable power mode according to the link state.

Usage

```
STATUS NU_USB_PMG_Set_Link_State (NU_USB_DEVICE *cb,  
                                UINT8          link_state)
```

Arguments

- **cb**
Address of NU_USB_DEVICE control block
- **link_state**
Variable containing the current link state to be saved.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Description

This API is called once the USB controller notifies the controller driver that the link state has been updated. The details of this notification are hardware implementation dependent.

Related Topics

[Nucleus USB Power Management Services](#)

Nucleus USB Stack Services

This section provides a detailed reference of the following Nucleus USB Stack Services:

- [NU_USB_STACK_Add_Hw](#)
- [NU_USB_STACK_Deregister_Drvr](#)
- [NU_USB_STACK_Function_Suspend](#)
- [NU_USB_STACK_Register_Drvr](#)
- [NU_USB_STACK_Remove_Hw](#)

Related Topics

[USB Host and Functions](#)

NU_USB_STACK_Add_Hw

This function adds a new USB hardware controller to the stack.

Usage

```
STATUS NU_USB_STACK_Add_Hw (NU_USB_STACK *cb,  
                           NU_USB_HW *controller)
```

Arguments

- **cb**
Pointer to the stack control block.
- **controller**
Pointer to the hardware control block.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_USB_MAX_EXCEEDED**
Indicates that more than the number of controllers that are configured are being added.
- **NU_USB_INVLD_ARG**
Indicates that one or more args passed to this function are invalid.
- **NU_INVALID_SEMAPHORE**
Indicates the semaphore pointer is invalid.
- **NU_SEMAPHORE_DELETED**
Semaphore was deleted while the task was suspended.
- **NU_UNAVAILABLE**
Indicates the semaphore is unavailable.
- **NU_INVALID_SUSPEND**
Indicates that this API is called from a non-task context.

Description

The controller is made operational in case the host stack root hub contained in the controller is enumerated. Any USB devices connected to the root hub undergo enumeration and may be activated by their matching USB drivers, if any.

Related Topics

[Nucleus USB Stack Services](#)

NU_USB_STACK_Deregister_Drvr

This function deregisters the USB driver from the stack.

Usage

```
STATUS NU_USB_STACK_Deregister_Drvr (NU_USB_STACK *cb,  
                                     NU_USB_DRVVR *usb_driver)
```

Arguments

- **cb**
Pointer to the stack control block.
- **usb_driver**
Pointer to the class driver control block to be deregistered.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_USB_INVLD_ARG**
Indicates that one or more args passed to this function are invalid.

Description

The class driver must have been registered with the stack earlier. No newly detected devices/interfaces are notified to the deregistered driver thereafter.

Related Topics

[Nucleus USB Stack Services](#)

NU_USB_STACK_Function_Suspend

Call this function to handle a Set Feature (Function_Suspend) standard request. This API can be called in both host and function modes with appropriate arguments.

Usage

```
STATUS NU_USB_STACK_Function_Suspend ( NU_USB_STACK *cb,
                                       NU_USB_INTF *intf,
                                       BOOLEAN func_suspend,
                                       BOOLEAN rmt_wakeup)
```

Arguments

- **cb**
 Pointer to NU_USB_STACK control block.
- **intf**
 Pointer to NU_USB_INTF control block.
- **func_suspend**
 Sets the function suspend bit. This can be set to one of the following values:
 - NU_TRUE
 Set function suspend bit position to 1.
 - NU_FALSE
 Set function suspend bit position to 0.
- **rmt_wakeup**
 Sets the remote wake-up bit. This can be set to one of the following values:
 - NU_TRUE
 Set remote wake-up bit position to 1.
 - NU_FALSE
 Set remote wake-up bit position to 0.

Return Values

- **NU_SUCCESS**
 Operation completed successfully.
- **NU_USB_INVLD_ARG**
 No valid input argument.

Related Topics

[Nucleus USB Stack Services](#)

NU_USB_STACK_Register_Drvr

This function registers the USB driver with the stack. The driver's Initialize_intf/Initialize_Device functions gets invoked if any unclaimed interfaces/devices match this new driver.

Usage

```
STATUS NU_USB_STACK_Register_Drvr (NU_USB_STACK *cb,  
                                   NU_USB_DRVVR *class_driver)
```

Arguments

- **cb**
Pointer to the stack control block.
- **class_driver**
Pointer to the class driver control block to be registered.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_USB_MAX_EXCEEDED**
Indicates that more than the configured number of USB drivers are being registered with the stack
- **NU_USB_INVLD_ARG**
Indicates that one or more args passed to this function are invalid.

Related Topics

[Nucleus USB Stack Services](#)

NU_USB_STACK_Remove_Hw

This function removes the controller hardware from the stack that was previously added to it. In the case of the host stack, the root hub is de-enumerated causing the whole associated device topology to be de-enumerated.

Usage

```
STATUS NU_USB_STACK_Remove_Hw (NU_USB_STACK *cb,  
                               NU_USB_HW    *controller)
```

Arguments

- **cb**
Pointer to the stack control block.
- **controller**
Pointer to the hardware control block.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_NOT_PRESENT**
Indicates that the hardware controller is not added to the stack.

Related Topics

[Nucleus USB Stack Services](#)

Nucleus USB Bulk Streaming Services

This section provides a detailed reference of the following Nucleus USB Bulk Streaming services:

- [NU_USB_STRM_Create](#)
- [NU_USB_STRM_Get_State](#)
- [NU_USB_STRM_Get_Stream_ID](#)
- [NU_USB_STRM_GRP_Acquire_Arbitrate](#)
- [NU_USB_STRM_GRP_Acquire_Forceful](#)
- [NU_USB_STRM_GRP_Create](#)
- [NU_USB_STRM_GRP_Delete](#)
- [NU_USB_STRM_GRP_Release_Strm](#)
- [NU_USB_STRM_Set_State](#)
- [NU_USB_STRM_Set_Stream_ID](#)

Related Topics

[USB Host and Functions](#)

NU_USB_STRM_Create

This function creates and initializes a stream.

Usage

```
STATUS NU_USB_STRM_Create (NU_USB_STREAM      *cb,  
                           UINT32             length,  
                           UINT8              *data,  
                           NU_USB_IRP_CALLBACK callback)
```

Arguments

- **cb**
Pointer to NU_USB_STREAM control block
- **length**
Length of data to be received/sent on this stream.
- **data**
Pointer to data buffer.
- **callback**
Function to be called by the controller driver once the operation on the stream is completed.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Bulk Streaming Services](#)

NU_USB_STRM_Get_State

This function gets the current state of the stream.

Usage

```
STATUS NU_USB_STRM_Get_State (NU_USB_STREAM *cb,  
                             UINT8          *state_out)
```

Arguments

- **cb**
Pointer to NU_USB_STREAM control block.
- **state_out**
Pointer to a variable containing the stream state when the function returns.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Bulk Streaming Services](#)

NU_USB_STRM_Get_Stream_ID

This function gets the stream ID of stream identified by 'cb'.

Usage

```
STATUS NU_USB_STRM_Get_Stream_ID (NU_USB_STREAM *cb,  
                                UINT8          *stream_id_out)
```

Arguments

- **cb**
Pointer to NU_USB_STREAM control block.
- **stream_id_out**
Pointer to a variable containing the stream ID when the function returns.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Bulk Streaming Services](#)

NU_USB_STRM_GRP_Acquire_Arbitrate

This function searches for a free stream available in a stream group, reserves it in hardware, marks it used in software data structures, and gives the caller access to the acquired stream.

Usage

```
STATUS NU_USB_STRM_GRP_Acquire_Arbitrate (
    NU_USB_STREAM_GRP *stream_grp,
    UINT16             *stream_id_out,
    NU_USB_STREAM      **stream)
```

Arguments

- **stream_grp**
Pointer to NU_USB_STREAM_GRP control block to be created.
- **stream_id_out**
Pointer to the ID of the acquired stream.
- **stream**
Double pointer to the acquired stream.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_NO_STRM_AVAILABLE**
No free stream is available.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Bulk Streaming Services](#)

NU_USB_STRM_GRP_Acquire_Forceful

This function gives access to the requested stream identified by 'stream_id' if it is already free, otherwise returns an error.

Usage

```
STATUS NU_USB_STRM_GRP_Acquire_Forceful (NU_USB_STREAM_GRP *stream_grp,
                                         UINT16             stream_id,
                                         NU_USB_STREAM      **stream_out)
```

Arguments

- stream_grp
Pointer to NU_USB_STREAM_GRP control block.
- stream_id
Pointer to the stream ID to be acquired.
- stream_out
Double pointer to the acquired stream.

Return Values

- NU_SUCCESS
Operation completed successfully.
- NU_USB_STRM_BUSY
Stream is already acquired.
- NU_USB_INVLD_ARG
One or more input arguments are invalid.

Related Topics

[Nucleus USB Bulk Streaming Services](#)

NU_USB_STRM_GRP_Create

This function creates a USB bulk stream group. A stream group can be associated with one endpoint at a time.

Usage

```
STATUS NU_USB_STRM_GRP_Create (NU_USB_STREAM_GRP *stream_grp,  
                              UINT16             num_streams)
```

Arguments

- **stream_grp**
Pointer to NU_USB_STREAM_GRP to be created.
- **num_streams**
Number of streams in this stream group.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Bulk Streaming Services](#)

NU_USB_STRM_GRP_Delete

This function deletes an existing stream group.

Usage

```
STATUS NU_USB_STRM_GRP_Delete (NU_USB_STREAM_GRP *stream_grp)
```

Arguments

- `stream_grp`
Pointer to `NU_USB_STREAM_GRP` to be deleted.

Return Values

- `NU_SUCCESS`
Operation completed successfully.
- `NU_USB_INVLD_ARG`
One or more input arguments are invalid.

Related Topics

[Nucleus USB Bulk Streaming Services](#)

NU_USB_STRM_GRP_Release_Strm

This function releases an already acquired stream.

Usage

```
STATUS NU_USB_STRM_GRP_Release_Strm (NU_USB_STREAM_GRP *stream_grp,  
                                     UINT16               stream_id)
```

Arguments

- `stream_grp`
Pointer to NU_USB_STREAM_GRP control block.
- `stream_id`
Stream ID to be released.

Return Values

- `NU_SUCCESS`
Operation completed successfully.
- `NU_USB_INVLD_ARG`
One or more input arguments are invalid.

Related Topics

[Nucleus USB Bulk Streaming Services](#)

NU_USB_STRM_Set_State

This function sets the current state of the stream.

Usage

```
STATUS NU_USB_STRM_Set_State (NU_USB_STREAM *cb,
                             UINT8          state)
```

Arguments

- **cb**
Pointer to NU_USB_STREAM control block.
- **state**
The new state of the stream.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Bulk Streaming Services](#)

NU_USB_STRM_Set_Stream_ID

This function sets the stream ID of the stream identified by “cb”.

Usage

```
STATUS NU_USB_STRM_Set_Stream_ID (NU_USB_STREAM *cb,  
                                UINT8          stream_id)
```

Arguments

- **cb**
Pointer to NU_USB_STREAM control block.
- **stream_id**
The ID of the stream.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Bulk Streaming Services](#)

Nucleus USB User Services

This section provides a detailed reference of the following Nucleus USB User Services:

- [NU_USB_USER_Connect](#)
- [NU_USB_USER_Disconnect](#)
- [NU_USB_USER_Get_Protocol](#)
- [NU_USB_USER_Get_Subclass](#)

Related Topics

[USB Host and Functions](#)

NU_USB_USER_Connect

A class driver calls this function when a user suitable for this device is detected by it in the class driver's `Initialize_Interface` or `Initialize_Device` functions.

Usage

```
STATUS NU_USB_USER_Connect (NU_USB_USER *cb,  
                           NU_USB_DRV *class_driver,  
                           VOID *handle)
```

Arguments

- `cb`
Pointer to the user driver control block.
- `class_driver`
Pointer to the class driver control block.
- `handle`
Context information.

Return Values

- `NU_SUCCESS`
Indicates successful completion of the service.

Description

This function invokes the `Connect` function pointer installed in the `NU_USB_USER_DISPATCH` of the user driver. The class driver generates a handle, an opaque ID, to uniquely identify this connection.

Related Topics

[Nucleus USB User Services](#)

NU_USB_USER_Disconnect

This function invokes the Disconnect function pointer installed in the NU_USB_USER_DISPATCH of the user driver. [NU_USB_USER_Connect](#) is called by a class driver for the user driver associated with the device from the class driver's Disconnect function.

Usage

```
STATUS NU_USB_USER_Disconnect (NU_USB_USER *cb,
                               NU_USB_DRV *class_driver,
                               VOID *handle)
```

Arguments

- **cb**
User control block.
- **class_driver**
Driver control block calling the callback.
- **handle**
Context information.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB User Services](#)

NU_USB_USER_Get_Protocol

This function returns the protocol this user is created to serve.

Usage

```
STATUS NU_USB_USER_Get_Protocol (NU_USB_USER *cb,  
                                UINT8        *protocol_out)
```

Arguments

- **cb**
Pointer to the control block of NU_USB_USER or its derivative.
- **protocol_out**
Pointer to a variable to hold the value of the protocol code that's supported by this user.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB User Services](#)

NU_USB_USER_Get_Subclass

This function returns the subclass this user is created to serve.

Usage

```
STATUS NU_USB_USER_Get_Subclass (NU_USB_USER *cb,  
                                UINT8        *subclass_out)
```

Arguments

- **cb**
Pointer to the control block of NU_USB_USER or its derivative.
- **subclass_out**
Pointer to a variable to hold the value of the protocol code that's supported by this user.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB User Services](#)

Nucleus USB Host Services

This section provides a detailed reference of the following Nucleus USB Host Services:

- [NU_USBH_Create](#)
- [NU_USBH_Delete](#)
- [NU_USBH_Validate_Drvr_Object](#)
- [NU_USBH_Validate_HW_Object](#)
- [NU_USBH_Validate_Stack_Object](#)
- [NU_USBH_Validate_User_Object](#)

Related Topics

[USB Host and Functions](#)

NU_USBH_Create

This function creates and initializes the Nucleus USB host environment. This must be the first API invoked before any other Nucleus USB host services can be used.

Usage

```
STATUS NU_USBH_Create (NU_USBH *singleton_cb)
```

Arguments

- `singleton_cb`
Pointer to the NU_USBH control block.

Return Values

- `NU_SUCCESS`
Indicates successful completion of the service.

Related Topics

[Nucleus USB Host Services](#)

NU_USBH_Delete

This function destroys the Nucleus USB host environment. Any further calls to any of the Nucleus USB host APIs leads to undefined behavior.

Usage

```
STATUS NU_USBH_Delete (VOID)
```

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Host Services](#)

NU_USBH_Validate_Drvr_Object

This function checks if the `class_driver` is a valid control block created in the Nucleus USB host environment.

Usage

```
STATUS NU_USBH_Validate_Drvr_Object (NU_USBH_DVR *class_driver)
```

Arguments

- `class_driver`
Pointer to the driver control block to be validated.

Return Values

- `NU_SUCCESS`
Indicates that the class driver is valid.
- `NU_USB_INVLD_ARG`
Indicates that the class driver is invalid.

Related Topics

[Nucleus USB Host Services](#)

NU_USBH_Validate_HW_Object

This function checks if the `hw_driver` is a valid control block created in the Nucleus USB host environment.

Usage

```
STATUS NU_USBH_Validate_Hw_Object (NU_USBH_HW *hw_driver)
```

Arguments

- `hw_driver`
Pointer to the hardware control block to be validated.

Return Values

- `NU_SUCCESS`
Indicates that the hardware is valid.
- `NU_USB_INVLD_ARG`
Indicates that the hardware is invalid.

Related Topics

[Nucleus USB Host Services](#)

NU_USBH_Validate_Stack_Object

This function checks if the stack is a valid control block created in the Nucleus USB host environment.

Usage

```
STATUS NU_USBH_Validate_Stack_Object (NU_USBH_STACK *stack)
```

Arguments

- `stack`
Pointer to the stack control block to be validated.

Return Values

- `NU_SUCCESS`
Indicates that the stack is valid.
- `NU_USB_INVLD_ARG`
Indicates that the stack is invalid.

Related Topics

[Nucleus USB Host Services](#)

NU_USBH_Validate_User_Object

This function checks if the user is a valid control block created in the Nucleus USB host environment.

Usage

```
STATUS NU_USBH_Validate_User_Object (NU_USBH_USER *user)
```

Arguments

- **user**
Pointer to the user control block to be validated.

Return Values

- **NU_SUCCESS**
Indicates that the user is valid.
- **NU_USB_INVLD_ARG**
Indicates that the user is invalid.

Related Topics

[Nucleus USB Host Services](#)

Nucleus USB Host Control IRP Services

This section provides a detailed reference of the following Nucleus USB Host Control IRP Services:

- [NU_USBH_CTRL_IRP_Create](#)
- [NU_USBH_CTRL_IRP_Get_bRequest](#)
- [NU_USBH_CTRL_IRP_Get_bmRequestType](#)
- [NU_USBH_CTRL_IRP_Get_Direction](#)
- [NU_USBH_CTRL_IRP_Get_Setup_Pkt](#)
- [NU_USBH_CTRL_IRP_Get_wIndex](#)
- [NU_USBH_CTRL_IRP_Get_wLength](#)
- [NU_USBH_CTRL_IRP_Get_wValue](#)
- [NU_USBH_CTRL_IRP_Set_bRequest](#)
- [NU_USBH_CTRL_IRP_Set_bmRequestType](#)
- [NU_USBH_CTRL_IRP_Set_wIndex](#)
- [NU_USBH_CTRL_IRP_Set_wLength](#)
- [NU_USBH_CTRL_IRP_Set_wValue](#)

Related Topics

[USB Host and Functions](#)

NU_USBH_CTRL_IRP_Create

This function initializes the control IRP control block with supplied values.

Usage

```
STATUS NU_USBH_CTRL_IRP_Create (NU_USBH_CTRL_IRP    *cb,  
                                UINT8                *data,  
                                NU_USBH_IRP_CALLBACK callback,  
                                VOID                 *context,  
                                UINT8                bmRequestType,  
                                UINT8                bRequest,  
                                UINT16               wValue,  
                                UINT16               wIndex,  
                                UINT16               wLength)
```

Arguments

- **cb**
User supplier pointer to the memory of the control IRP control block.
- **data**
Pointer to the data buffer for in/out data transfers during data phase.
- **callback**
The function pointer that is invoked by the stack, once the IRP transfer attempt is completed by the stack.

Note



This callback function should avoid making calls that require the NU_SUSPEND option.

- **context**
A cookie that is passed to the callback function when it is invoked by the stack. The caller can store information in this pointer that can help identify the IRP when the callback is invoked.
- **bmRequestType**
The field of the setup packet defined by the USB standard.
- **bRequest**
The field of the setup packet defined by the USB standard.
- **wValue**
The field of the setup packet defined by the USB standard.
- **wIndex**
The field of the setup packet defined by the USB standard.
- **wLength**
The field of the setup packet defined by the USB standard.

Return Values

- **NU_SUCCESS**

Indicates successful completion of the service.

Description

NU_USBH_CTRL_IRP is a specialization of NU_USB_IRP. It fills up the set up packet fields for the setup phase and the data fields for the data phase of the control transfer. All fields should be in little endian byte ordering. To delete a control IRP, use [NU_USB_IRP_Delete](#).

Related Topics

[Nucleus USB Host Control IRP Services](#)

NU_USBH_CTRL_IRP_Get_bRequest

This function returns the bRequest field of the setup packet associated with the control IRP.

Usage

```
STATUS NU_USBH_CTRL_IRP_Get_bRequest (NU_USBH_CTRL_IRP *cb,  
                                       UINT8 *bRequest_out)
```

Arguments

- **cb**
Pointer to the control IRP control block.
- **bRequest_out**
Pointer to the memory location to hold the value of the bRequest field of the setup packet

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Host Control IRP Services](#)

NU_USBH_CTRL_IRP_Get_bmRequestType

This function returns the bmRequestType field of the setup packet associated with the control IRP.

Usage

```
STATUS NU_USBH_CTRL_IRP_Get_bmRequestType (  
    NU_USBH_CTRL_IRP *cb,  
    UINT8             *bmRequestType_out)
```

Arguments

- **cb**
Pointer to the control IRP control block.
- **bmRequestType_out**
Pointer to the memory location to hold the value of the bmRequestType field of the setup packet.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Host Control IRP Services](#)

NU_USBH_CTRL_IRP_Get_Direction

This function returns the direction of the transfer during the data phase of this control IRP, as gathered from the bmRequestType field.

Usage

```
STATUS NU_USBH_CTRL_IRP_Get_Direction (NU_USBH_CTRL_IRP *cb,  
                                       UINT8 *direction_out)
```

Arguments

- **cb**
Pointer to the control IRP control block.
- **direction_out**
Pointer to the variable to hold the value of the direction USB_DIR_IN or USB_DIR_OUT.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Host Control IRP Services](#)

NU_USBH_CTRL_IRP_Get_Setup_Pkt

This function returns pointer to the setup packet associated with the control IRP.

Usage

```
STATUS NU_USBH_CTRL_IRP_Get_Setup_Pkt (NU_USBH_CTRL_IRP *cb,  
                                       NU_USB_SETUP_PKT **pkt_out)
```

Arguments

- **cb**
Pointer to the control IRP control block.
- **pkt_out**
Pointer to the memory location to hold the pointer to the setup packet associated with the control IRP.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Host Control IRP Services](#)

NU_USBH_CTRL_IRP_Get_wIndex

This function returns the wIndex field of the setup packet associated with the control IRP.

Usage

```
STATUS NU_USBH_CTRL_IRP_Get_wIndex (NU_USBH_CTRL_IRP *cb,  
                                     UINT16          *wIndex_out)
```

Arguments

- **cb**
Pointer to the control IRP control block.
- **wIndex_out**
Pointer to the memory location to hold the value of the wIndex field of the setup packet.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Host Control IRP Services](#)

NU_USBH_CTRL_IRP_Get_wLength

This function returns the wLength field of the setup packet associated with the control IRP.

Usage

```
STATUS NU_USBH_CTRL_IRP_Get_wLength (NU_USBH_CTRL_IRP *cb,  
                                     UINT16 *wLength_out)
```

Arguments

- **cb**
Pointer to the control IRP control block.
- **wLength_out**
Pointer to the memory location to hold the value of the wLength field of the setup packet.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Host Control IRP Services](#)

NU_USBH_CTRL_IRP_Get_wValue

This function returns the wValue field of the setup packet associated with the control IRP.

Usage

```
STATUS NU_USBH_CTRL_IRP_Get_wValue (NU_USBH_CTRL_IRP *cb,  
                                     UINT16          *wValue_out)
```

Arguments

- **cb**
Pointer to the control IRP control block.
- **wValue_out**
Pointer to the memory location to hold the value of the wValue field of the setup packet.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Host Control IRP Services](#)

NU_USBH_CTRL_IRP_Set_bRequest

This function sets the bRequest field of the setup packet associated with the control IRP.

Usage

```
STATUS NU_USBH_CTRL_IRP_Set_bRequest (NU_USBH_CTRL_IRP *cb,  
                                       UINT8          bRequest)
```

Arguments

- **cb**
Pointer to the control IRP control block.
- **bRequest**
bRequest value of the setup packet associated with the control IRP.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Host Control IRP Services](#)

NU_USBH_CTRL_IRP_Set_bmRequestType

This function sets the bmRequestType field of the setup packet associated with the control IRP.

Usage

```
STATUS NU_USBH_CTRL_IRP_Set_bmRequestType (  
                                         NU_USBH_CTRL_IRP *cb,  
                                         UINT8          bmRequestType)
```

Arguments

- **cb**
Pointer to the control IRP control block.
- **bmRequestType**
bmRequestType value of the setup packet associated with the control IRP.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Host Control IRP Services](#)

NU_USBH_CTRL_IRP_Set_wIndex

This function sets the wIndex field of the setup packet associated with the control IRP.

Usage

```
STATUS NU_USBH_CTRL_IRP_Set_wIndex (NU_USBH_CTRL_IRP *cb,  
                                     UINT16 wIndex)
```

Arguments

- **cb**
Pointer to the control IRP control block.
- **wIndex**
wIndex value of the setup packet associated with the control IRP.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Host Control IRP Services](#)

NU_USBH_CTRL_IRP_Set_wLength

This function sets the wLength field of the setup packet associated with the control IRP.

Usage

```
STATUS NU_USBH_CTRL_IRP_Set_wLength (NU_USBH_CTRL_IRP *cb,  
                                     UINT16          wLength)
```

Arguments

- **cb**
Pointer to the control IRP control block.
- **wLength**
wLength value of the setup packet associated with the control IRP.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Host Control IRP Services](#)

NU_USBH_CTRL_IRP_Set_wValue

This function sets the wValue field of the setup packet associated with the control IRP.

Usage

```
STATUS NU_USBH_CTRL_IRP_Set_wValue (NU_USBH_CTRL_IRP *cb,  
                                     UINT16          wValue)
```

Arguments

- **cb**
Pointer to the control IRP control block.
- **wValue**
wValue value of the setup packet associated with the control IRP.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Related Topics

[Nucleus USB Host Control IRP Services](#)

Nucleus USB Host Class Driver Services

This section provides a detailed reference of the following Nucleus USB Host Class Driver Services:

- [_NU_USBH_DRV_CREATE](#)
- [_NU_USBH_DRV_DELETE](#)

Related Topics

[USB Host and Functions](#)

_NU_USBH_DRV Create

This function initializes the USB Host class driver.

Usage

```
STATUS _NU_USBH_DRV Create (NU_USBH_DRV *cb,
                           CHAR          *name,
                           UINT32       match_flag,
                           UINT16       idVendor,
                           UINT16       idProduct,
                           UINT16       bcdDeviceLow,
                           UINT16       bcdDeviceHigh,
                           UINT8        bInterfaceClass,
                           UINT8        bInterfaceSubClass,
                           UINT8        bInterfaceProtocol,
                           const VOID   *dispatch)
```

Arguments

- **cb**
 Pointer to the user-supplied class driver control block. NOTE: All subsequent requests made to NU_USBH_DRV/NU_USB_DRV require this pointer.

- **name**
 Pointer to a 7-character name for the driver. The name must be null-terminated.

- **match_flag**
 bit wise OR any of the following:

USB_MATCH_VNDR_ID

The driver can serve those devices whose device descriptor's idVendor matches the idVendor of this driver.

USB_MATCH_PRDCT_ID

The driver can serve those devices whose device descriptor's idVendor and idProduct match that of this driver. if USB_MATCH_PRDCT_ID is in match flag, USB_MATCH_VNDR_ID must necessarily be part of the match_flag.

USB_MATCH_REL_NUM

The driver can serve those devices whose device descriptor's idVendor, idProduct match that of this driver and devices bcdDevice is in the range of bcdDeviceLow and bcdDevice high of the driver. If USB_MATCH_REL_NUM is in match flag, USB_MATCH_PRDCT_ID must necessarily be part of the match_flag.

USB_MATCH_CLASS

The driver can serve those device interfaces whose interface descriptor's bInterfaceClass matches that of this driver.

USB_MATCH_SUB_CLASS

The driver can serve those device interfaces whose interface descriptor's bInterfaceClass and bInterfaceSubClass matches that of this driver. If

USB_MATCH_SUB_CLASS is in match flag, USB_MATCH_CLASS must necessarily be part of the match flag.

USB_MATCH_PROTOCOL

The driver can serve those device interfaces whose interface descriptor's bInterfaceClass, bInterfaceSubClass and bInterfaceProtocol matches that of this driver. If USB_MATCH_SUB_CLASS is in match_flag, USB_MATCH_CLASS must necessarily be part of the match_flag.

- **idVendor**
Vendor ID of the devices that this driver can serve. Irrelevant, if USB_MATCH_VNDR_ID is not set in the match_flag.
- **idProduct**
Product ID of the devices that this driver can serve. Irrelevant, if USB_MATCH_PRDCT_ID is not set in the match_flag.
- **bcdDeviceLow**
Release numbers of the device that this driver can serve. The lowest number of this release number range. Irrelevant if USB_MATCH_REL_NUM is not set in the match flag.
- **bcdDeviceHigh**
Release numbers of the device that this driver can serve. The highest number of this release number range. Irrelevant if USB_MATCH_REL_NUM is not set in the match flag.
- **bInterfaceClass**
Interface class code of the device interfaces that this driver can serve. Irrelevant, if USB_MATCH_CLASS is not set.
- **bInterfaceSubClass**
Interface sub class code of the device interfaces that this driver can serve. Irrelevant, if USB_MATCH_SUB_CLASS is not set.
- **bInterfaceProtocol**
Interface protocol code of the device interfaces that this driver can serve. Irrelevant, if USB_MATCH_PROTOCOL is not set.
- **dispatch**
Dispatch table of the class driver.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_USB_INVLD_ARG**
Indicates that the match_flag is not well formed.

- **NU_USB_NOT_PRESENT**

Indicates that the maximum number of control blocks that could be created in the sub system has exceeded.

Description

The Arguments describe the match spec of the class driver. The valid values of the match spec are indicated by the match_flag. This function must be called to create a derived component from the NU_USBH_DRV.

Related Topics

[Nucleus USB Host Class Driver Services](#)

_NU_USBH_DRV_R_Delete

This function releases the resources, if any, that were allocated from the `_NU_USBH_DRV_R_Create` function. This function must be called from the delete function of the component that extends the `NU_USBH_DRV_R` component.

Usage

```
STATUS_NU_USBH_DRV_R_Delete (VOID *cb)
```

Arguments

- `cb`
Pointer to the host class driver control block that has to be deleted.

Return Values

- `NU_SUCCESS`
Indicates successful completion of the service.

Related Topics

[Nucleus USB Host Class Driver Services](#)

Nucleus USB Host Hardware Driver Services

This section provides a detailed reference of the following Nucleus USB Host Hardware Driver Services:

- [_NU_USBH_HW_Create](#)
- [_NU_USBH_HW_Delete](#)

Related Topics

[USB Host and Functions](#)

_NU_USBH_HW_Create

This function initializes the NU_USBH_HW control block.

Usage

```
STATUS _NU_USBH_HW_Create (NU_USBH_HW      *cb,  
                           NU_USB_HWENV    *hw_env,  
                           CHAR             *name,  
                           NU_MEMORY_POOL  *pool,  
                           UINT8           num_companion_hw,  
                           UINT8           speed,  
                           VOID            *base_address,  
                           INT             vector_number,  
                           const VOID     *dispatch)
```

Arguments

- **cb**
Pointer to the NU_USBH_HW control block.
- **hw_env**
Pointer to the control block of the associated NU_USB_HWENV component.
- **name**
Pointer to a 7-character name for the hardware driver. The name must be null-terminated.
- **pool**
Pool pointer from which the hardware should meet all its memory requirements.
- **num_companion_hw**
Number of companion controllers the hardware has. USB 2.0 controllers, typically those based on EHCI will have companion controllers to deal with full and low speeds. A USB 1.1 controller will not generally have companion controllers so while creating such hardware, this parameter can be 0.
- **speed**
Speed of the host controller's root hub. This can be set to one of the following: USB_SPEED_HIGH, USB_SPEED_LOW, USB_SPEED_FULL.
- **base_address**
The base address of the register set of the controller.
- **vector_number**
The vector number for the interrupt generated by the hardware.
- **dispatch**
Pointer to the dispatch table containing function pointers of the derivative of NU_USBH_HW that actually implements NU_USBH_HW services.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_USB_NOT_PRESENT**
Indicates that the maximum number of control blocks that could be created in the sub system has exceeded.

Description

NU_USBH_HW is a derivative of NU_USB_HW. This function invokes the NU_USB_HW create function to initialize the NU_USB_HW portion of the control block.

Related Topics

[Nucleus USB Host Hardware Driver Services](#)

_NU_USBH_HW_Delete

This function releases any resources that were allocated by the `_NU_USBH_HW_Create` function. This function must be called from the delete routine of the component that derives from `NU_USBH_HW`.

Usage

```
STATUS _NU_USBH_HW_Delete (VOID *cb)
```

Arguments

- `cb`
Pointer to the control block of derivative of the `NU_USBH_HW`.

Return Values

- `NU_SUCCESS`
Indicates successful completion.
- `NU_NOT_PRESENT`
Indicates that the control block to be deleted is invalid.

Related Topics

[Nucleus USB Host Hardware Driver Services](#)

Nucleus USB Host STACK Services

This section provides a detailed reference of the following Nucleus USB Host STACK Services:

- [NU_USBH_STACK_Create](#)
- [NU_USBH_STACK_Get_Config_Info](#)
- [NU_USBH_STACK_Get_Devices](#)
- [NU_USBH_STACK_LTM_Disable](#)
- [NU_USBH_STACK_LTM_Enable](#)
- [NU_USBH_STACK_Resume_Device](#)
- [NU_USBH_STACK_Suspend_Device](#)
- [NU_USBH_STACK_Switch_Config](#)
- [NU_USBH_STACK_U1_Disable](#)
- [NU_USBH_STACK_U1_Enable](#)
- [NU_USBH_STACK_U2_Disable](#)
- [NU_USBH_STACK_U2_Enable](#)

Related Topics

[USB Host and Functions](#)

NU_USBH_STACK_Create

This function creates a Nucleus USB host stack and creates all the resources the host stack component needs.

Usage

```
STATUS NU_USBH_STACK_Create (NU_USBH_STACK *cb,  
                             CHAR           *name,  
                             NU_MEMORY_POOL *pool,  
                             VOID           *stack_task_stack_address,  
                             UNSIGNED       stack_task_stack_size,  
                             OPTION        stack_task_priority,  
                             VOID           *hub_task_stack_address,  
                             UNSIGNED       hub_task_stack_size,  
                             OPTION        hub_task_priority,  
                             VOID           *histr_stack_address,  
                             UNSIGNED       histr_stack_size,  
                             OPTION        histr_priority)
```

Arguments

- **cb**
Pointer to the host stack control block.
- **name**
Pointer to a 7-character name for the driver. The name must be null-terminated.
- **pool**
A pointer to the memory pool the stack uses to allocate memory for its use.
- **stack_task_stack_address**
Pointer to the host stack task's stack area.
- **stack_task_stack_size**
Host stack task's stack size. Should be at least 4K bytes.
- **stack_task_priority**
Host stack task's priority. Must be the best priority in the system.
- **hub_task_stack_address**
Pointer to the stack area for the hub task.
- **hub_task_stack_size**
Hub task's stack size. Should be at least 4K bytes.
- **hub_task_priority**
Priority of the hub task. Should be the best priority in the system.
- **histr_stack_address**
Stack address for the USB host stack HISR.

- `hirs_stack_size`
Stack size for the USB host stack HIRS.
- `hirs_priority`
Priority of the USB host stack HIRS. HIRS priority should also be the best.

Return Values

- `NU_SUCCESS`
Successful request.
- `NU_INVALID_MEMORY`
Stack pointer is NULL.
- `NU_INVALID_SIZE`
Stack size is too small.
- `NU_INVALID_PRIORITY`
Invalid task priority or invalid HIRS priority.

Description

The pool size depends on the number of devices connected. It must be a minimum of 2KB with additional 12KB for each device connected to the host.

Related Topics

[Nucleus USB Host STACK Services](#)

NU_USBH_STACK_Get_Config_Info

This function returns the number of configurations in a device and their respective IDs to the caller.

Usage

```
STATUS NU_USBH_STACK_Get_Config_Info (NU_USB_DEVICE *device,  
                                     UINT8          *num_config,  
                                     UINT8          *config_values)
```

Arguments

- **device**
Pointer to NU_USBH_DEVICE control block.
- **num_config**
Output argument containing the number of configurations in a device when the function returns.
- **config_values**
Array of NU_USB_MAX_CONFIGURATIONS number of elements. This array contains the actual configuration IDs when the function returns.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Host STACK Services](#)

NU_USBH_STACK_Get_Devices

This function returns the first non-root hub device attached on the bus. Starting from this device, the caller can traverse the whole tree.

Usage

```
NU_USB_DEVICE *NU_USBH_STACK_Get_Devices (NU_USBH_HW *hw)
```

Arguments

- **hw**
Pointer to NU_USBH_HW control block.

Return Values

- **NU_USB_DEVICE**
Pointer to NU_USB_DEVICE control block which points to first no-root hub device of the USB tree.
- **NU_NULL**
Either there is no device attached with USB host or there was some error in execution.

Related Topics

[Nucleus USB Host STACK Services](#)

NU_USBH_STACK_LTM_Disable

This function disables the LTM generation feature of the device.

Usage

```
STATUS NU_USBH_STACK_LTM_Disable (NU_USB_STACK *cb,  
                                  NU_USB_DEVICE *device)
```

Arguments

- **cb**
Pointer to NU_USB_STACK control block.
- **device**
Pointer to NU_USBH_DEVICE control block.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.
- **NU_USB_INVLD_DEVICE**
USB device is invalid.

Related Topics

[Nucleus USB Host STACK Services](#)

NU_USBH_STACK_LTM_Enable

This function enables the LTM generation feature of the device.

Usage

```
STATUS NU_USBH_STACK_LTM_Enable (NU_USB_STACK *cb,  
                                NU_USB_DEVICE *device)
```

Arguments

- **cb**
Pointer to NU_USB_STACK control block.
- **device**
Pointer to NU_USBH_DEVICE control block.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.
- **NU_USB_INVLD_DEVICE**
USB device is invalid.

Related Topics

[Nucleus USB Host STACK Services](#)

NU_USBH_STACK_Resume_Device

This function resumes USB traffic on a given USB port.

Usage

```
STATUS NU_USBH_STACK_Resume_Device (NU_USBH_STACK *cb,  
                                     NU_USB_DEVICE *device)
```

Arguments

- **cb**
Pointer to NU_USBH_STACK control block.
- **device**
Pointer to NU_USB_DEVICE control block.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Host STACK Services](#)

NU_USBH_STACK_Suspend_Device

This function suspends the USB traffic on the bus on a given device. Callers must make sure that there are no active transfers on the USB, otherwise this may lead to data loss.

Usage

```
STATUS NU_USBH_STACK_Suspend_Device (NU_USBH_STACK *cb,  
                                     NU_USB_DEVICE *device)
```

Arguments

- **cb**
Pointer to NU_USBH_STACK control block.
- **device**
Pointer to NU_USB_DEVICE control block.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Host STACK Services](#)

NU_USBH_STACK_Switch_Config

This function is called by the user to switch configuration at run time.

Usage

```
STATUS NU_USBH_STACK_Switch_Config (NU_USBH_STACK *stack,  
                                   NU_USB_DEVICE *device,  
                                   UINT8          config_num)
```

Arguments

- **stack**
Pointer to NU_USBH_STACK control block.
- **device**
Pointer to NU_USBH_DEVICE control block.
- **config_num**
Number of configurations to be set as active by caller.

Return Values

- **NU_SUCCESS**
Operation completed successfully, configuration was switched to the requested configuration number.
- **NU_INVLD_ARG**
One or more input arguments are invalid.
- **NU_UNAVAILABLE**
The requested configuration is not available.

Related Topics

[Nucleus USB Host STACK Services](#)

NU_USBH_STACK_U1_Disable

This function sends a clear feature (U1_DISABLE) request to the SuperSpeed device. The U1_DISABLE feature selector disables the device from the U1 transition initiation.

Usage

```
STATUS NU_USBH_STACK_U1_Disable (NU_USB_STACK *cb,  
                                NU_USB_DEVICE *device)
```

Arguments

- **cb**
Pointer to NU_USB_STACK control block.
- **device**
Pointer to NU_USBH_DEVICE control block.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.
- **NU_USB_INVLD_DEVICE**
USB device is invalid.

Related Topics

[Nucleus USB Host STACK Services](#)

NU_USBH_STACK_U1_Enable

This function sends a set feature (U1_ENABLE) request to the SuperSpeed device. The U1_ENABLE feature selector enables the device to initiate a U1 transition on the link when the entry conditions for U1 are met.

Usage

```
STATUS NU_USBH_STACK_U1_Enable (NU_USB_STACK *cb,  
                                NU_USB_DEVICE *device)
```

Arguments

- **cb**
Pointer to NU_USB_STACK control block.
- **device**
Pointer to NU_USBH_DEVICE control block.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.
- **NU_USB_INVLD_DEVICE**
USB device is invalid.

Related Topics

[Nucleus USB Host STACK Services](#)

NU_USBH_STACK_U2_Disable

This function sends a clear feature (U2_DISABLE) request to the SuperSpeed device. The U2_DISABLE feature selector disables the device to initiate a U2 transition.

Usage

```
STATUS NU_USBH_STACK_U2_Disable (NU_USB_STACK *cb,  
                                NU_USB_DEVICE *device)
```

Arguments

- **cb**
Pointer to NU_USB_STACK control block.
- **device**
Pointer to NU_USBH_DEVICE control block.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.
- **NU_USB_INVLD_DEVICE**
USB device is invalid.

Related Topics

[Nucleus USB Host STACK Services](#)

NU_USBH_STACK_U2_Enable

This function sends a set feature (U2_ENABLE) request to the SuperSpeed device. The U2_ENABLE feature selector enables the device to initiate a U2 transition on the link when entry conditions for U2 are met.

Usage

```
STATUS NU_USBH_STACK_U2_Enable (NU_USB_STACK *cb,  
                                NU_USB_DEVICE *device)
```

Arguments

- **cb**
Pointer to NU_USB_STACK control block.
- **device**
Pointer to NU_USBH_DEVICE control block.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.
- **NU_USB_INVLD_DEVICE**
USB device is invalid.

Related Topics

[Nucleus USB Host STACK Services](#)

Nucleus USB Host User Driver Services

This section provides a detailed reference of the following Nucleus USB Host User Driver Services:

- [NU_USBH_USER_Close_Device](#)
- [NU_USBH_USER_Get_Drvr](#)
- [NU_USBH_USER_Open_Device](#)
- [NU_USBH_USER_Remove_Device](#)
- [NU_USBH_USER_Wait](#)
- [_NU_USBH_USER_Create](#)
- [_NU_USBH_USER_Delete](#)

Related Topics

[USB Host and Functions](#)

NU_USBH_USER_Close_Device

This function should be called from the user level threads/applications after they have finished using the device.

Usage

```
STATUS NU_USBH_USER_Close_Device (NU_USBH_USER *cb,  
                                VOID          *handle)
```

Arguments

- **cb**
Pointer to the host user driver control block.
- **handle**
Handle/cookie for the concerned device.

Return Values

- **NU_SUCCESS**
Indicates successful completion.
- **NU_USB_INVLD_ARG**
Indicates that some pointer became stale before the call was completed.
- **NU_NOT_PRESENT**
Indicates that the device has been safely removed.

Description

If the application calling this API happens to be the last thread that had kept this device open, and if there is any thread waiting for safe removal, the waiting thread is unblocked by this call and the device can be safely unplugged.

Related Topics

[Nucleus USB Host User Driver Services](#)

NU_USBH_USER_Get_Drvr

This function returns a pointer to the Nucleus USB class driver associated (in `drv_out`) with the given device.

Usage

```
STATUS NU_USBH_USER_Get_Drvr (NU_USBH_USER *cb,
                              VOID          *handle,
                              NU_USB_DRV   **drv_out)
```

Arguments

- `cb`
Pointer to the user control block.
- `handle`
Handle/cookie for the concerned device
- `drv_out`
Location where the class driver control block pointer can be stored.

Return Values

- `NU_SUCCESS`
Indicates successful completion.
- `NU_USB_INVLD_ARG`
Indicates that no device with the specified handle could be found.

Related Topics

[Nucleus USB Host User Driver Services](#)

NU_USBH_USER_Open_Device

Every application that requires access to the USB device's functionality must invoke this API to make the device ready for use.

Usage

```
STATUS NU_USBH_USER_Open_Device (NU_USBH_USER *cb,  
                                VOID *handle)
```

Arguments

- **cb**
Pointer to the host user driver control block.
- **handle**
Handle/cookie for the concerned device.

Return Values

- **NU_SUCCESS**
Indicates successful completion.
- **NU_USB_INVLD_ARG**
Indicates that some argument became stale before call was completed.
- **NU_NOT_PRESENT**
Indicates device has been safely removed.

Related Topics

[Nucleus USB Host User Driver Services](#)

NU_USBH_USER_Remove_Device

This function allows a thread to know when all threads are done with using the device, so that it can be safely unplugged.

Usage

```
STATUS NU_USBH_USER_Remove_Device (NU_USBH_USER *cb,
                                   VOID          *handle,
                                   UNSIGNED       suspend)
```

Arguments

- **cb**
Pointer to the host user driver control block.
- **handle**
Handle of the device to be removed safely.
- **suspend**
Suspension option. This can be set to one of the following values: NU_SUSPEND, NU_NO_SUSPEND, or timeout value.

Return Values

- **NU_SUCCESS**
Indicates successful completion.
- **NU_USB_INVLD_ARG**
Some pointer became stale before call completion.

Description

Such safe removal prevents any catastrophes that may happen if the device is unplugged while transfers to the device are in progress. The thread calling this API may choose to wait until all applications execute a 'close' on the device.

Related Topics

[Nucleus USB Host User Driver Services](#)

NU_USBH_USER_Wait

This function provides services for application threads to wait for a particular device to be connected.

Usage

```
STATUS NU_USBH_USER_Wait (NU_USBH_USER *cb,  
                          UNSIGNED      suspend,  
                          VOID          **handle_out)
```

Arguments

- **cb**
Pointer to the user control block.
- **suspend**
Suspension option.
- **handle_out**
Pointer to the memory location to hold pointer to the device handle.

Return Values

- **NU_SUCCESS**
Indicates successful completion.
- **NU_TIMEOUT**
Indicates timeout on suspension.
- **NU_NOT_PRESENT**
Indicates that event flags are not present.
- **NU_USB_INTERNAL_ERROR**
Indicates an internal error in USB subsystem

Description

The thread goes into a state specified by the suspension option if the device is not yet connected. This service returns a device handle as output when the call is successful. This handle is required for invoking open, close, or remove services on this device. If a thread invokes this API again, it is suspended (per the suspend option) until a new device connect event of this device category event occurs. Thus, with the help of this API an application thread can walk through all connected devices of this user category.

Related Topics

[Nucleus USB Host User Driver Services](#)

_NU_USBH_USER_Create

This function initializes the NU_USBH_USER control block and enables the services of the NU_USBH_USER component. The component that specializes from the NU_USBH_USER must call this function in its create routine with its dispatch table.

Usage

```
STATUS _NU_USBH_USER_Create (NU_USBH_USER *cb,  
                             CHAR *name,  
                             NU_MEMORY_POOL *pool,  
                             UINT8 bInterfaceSubclass,  
                             UINT8 bInterfaceProtocol,  
                             const VOID *dispatch)
```

Arguments

- **cb**
Pointer to the user control block.
- **name**
Pointer to a 7-character name for the driver. The name must be null-terminated.
- **pool**
Pointer to the memory pool to be used by the user driver.
- **bInterfaceSubclass**
Subclass this user is serving (0xFF, if the user driver deals only with protocol handling).
- **bInterfaceProtocol**
Protocol this user is serving (0xFF, if the user driver deals only with subclass handling).
- **dispatch**
Pointer to the dispatch table.

Return Values

- **NU_SUCCESS**
Indicates successful completion.
- **NU_INVALID_SEMAPHORE**
Indicates that the control block is invalid.
- **NU_INVALID_GROUP**
Indicates that the control block is invalid.

Related Topics

[Nucleus USB Host User Driver Services](#)

_NU_USBH_USER_Delete

This function releases the resources, if any, allocated in [_NU_USBH_USER_Create](#). This function must be called from the delete function of the component that derives NU_USBH_USER.

Usage

```
STATUS _NU_USBH_USER_Delete (VOID *cb)
```

Arguments

- **cb**
Pointer to the control block of NU_USBH_USER or its derivative.

Return Values

- **NU_SUCCESS**
Indicates successful completion.
- **NU_INVALID_POINTER**
Indicates control block is invalid.

Related Topics

[Nucleus USB Host User Driver Services](#)

Nucleus USB Function Class Driver Services

This section provides a detailed reference of the following Nucleus USB Function Class Driver Services:

- [_NU_USBF_DRVR_Create](#)
- [_NU_USBF_DRVR_Delete](#)

Related Topics

[USB Host and Functions](#)

_NU_USBF_DRV Create

This function initializes the USB function stack class driver.

Usage

```
STATUS _NU_USBF_DRV Create (NU_USBF_DRV *cb,  
                           CHAR          *name,  
                           UINT32       match_flag,  
                           UINT16       idVendor,  
                           UINT16       idProduct,  
                           UINT16       bcdDeviceLow,  
                           UINT16       bcdDeviceHigh,  
                           UINT8        bInterfaceClass,  
                           UINT8        bInterfaceSubClass,  
                           UINT8        bInterfaceProtocol,  
                           const VOID   *dispatch)
```

Arguments

- **cb**
Pointer to the user-supplied driver control block.

Note



All subsequent requests made to NU_USBF_DRV/NU_USB_DRV require this pointer.

- **name**
Pointer to a 7-character name for the driver. The name must be null-terminated.
- **match_flag**

You can OR of one or more of the following:

USB_MATCH_VNDR_ID

The driver can serve those devices whose device descriptor's idVendor matches the idVendor of this driver.

USB_MATCH_PRDCT_ID

The driver can serve those devices whose device descriptor's idVendor and idProduct match that of this driver if USB_MATCH_PRDCT_ID is in match flag, USB_MATCH_VNDR_ID must necessarily be part of the match_flag.

USB_MATCH_REL_NUM

The driver can serve those devices whose device descriptor's idVendor, idProduct match that of this driver and devices bcdDevice is in the range of bcdDeviceLow and bcdDevice high of the driver. If USB_MATCH_REL_NUM is in match flag, USB_MATCH_PRDCT_ID must necessarily be part of the match_flag.

USB_MATCH_CLASS

The driver can serve those device interfaces whose interface descriptor's bInterface Class matches that of this driver.

USB_MATCH_SUB_CLASS

The driver can serve those device interfaces whose interface descriptor's bInterfaceClass and bInterfaceSubClass matches that of this driver. If USB_MATCH_SUB_CLASS is in match flag, USB_MATCH_CLASS must necessarily be part of the match flag.

USB_MATCH_PROTOCOL

The driver can serve those device interfaces whose interface descriptor's bInterfaceClass, bInterfaceSubClass and bInterface Protocol matches that of this driver. If USB_MATCH_SUB_CLASS is in match_flag, USB_MATCH_CLASS must necessarily be part of the match_flag.

- **idVendor**
Vendor ID of the devices that this driver can serve. Irrelevant, if USB_MATCH_VNDR_ID is not set in the match_flag.
- **idProduct**
Product ID of the devices that this driver can serve. Irrelevant, if USB_MATCH_PRDCT_ID is not set in the match_flag.
- **bcdDeviceLow**
Release numbers of the device that this driver can serve. The lowest number of this release number range. Irrelevant if USB_MATCH_REL_NUM is not set in the match flag.
- **bcdDeviceHigh**
Release numbers of the device that this driver can serve. The highest number of this release number range. Irrelevant if USB_MATCH_REL_NUM is not set in the match flag.
- **bInterfaceClass**
Interface class code of the device interfaces that this driver can serve. Irrelevant, if USB_MATCH_CLASS is not set.
- **bInterfaceSubClass**
Interface sub class code of the device interfaces that this driver can serve. Irrelevant, if USB_MATCH_SUB_CLASS is not set.
- **bInterfaceProtocol**
Interface protocol code of the device interfaces that this driver can serve. Irrelevant, if USB_MATCH_PROTOCOL is not set.
- **dispatch**
Dispatch table of the class driver.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

- **NU_USB_INVLD_ARG**
Indicates that the match_flag is not well formed.
- **NU_USB_NOT_PRESENT**
Indicates that the maximum number of control blocks that could be created in the sub system has exceeded.

Description

The parameters describe the credentials of the devices the class driver services. For a given class driver, all fields may not be valid. The valid values are indicated by the match_flag parameter.

Related Topics

[Nucleus USB Function Class Driver Services](#)

_NU_USB_F_DRVR_Delete

This function releases all of the resources acquired by `_NU_USB_F_DRVR_Create`.

Usage

```
STATUS _NU_USB_F_DRVR_Delete (VOID *drv)
```

Arguments

- `drv`
Class driver to be deleted.

Return Values

- `NU_SUCCESS`
Indicates that the class driver has been uninitialized successfully.
- `NU_USB_INVLD_ARG`
Indicates that the supplied parameter is incorrect.

Related Topics

[Nucleus USB Function Class Driver Services](#)

USB Function Device Configuration Services

This section provides a detailed reference of the following Nucleus USB Function Device Configuration services:

- [USBF_DEVCFG_Activate_Device](#)
- [USBF_DEVCFG_Add_Config_String](#)
- [USBF_DEVCFG_Add_Function](#)
- [USBF_DEVCFG_Add_Intf_String](#)
- [USBF_DEVCFG_Bind_Function](#)
- [USBF_DEVCFG_Create_Config](#)
- [USBF_DEVCFG_Deactivate_Device](#)
- [USBF_DEVCFG_Delete_Config](#)
- [USBF_DEVCFG_Delete_Function](#)
- [USBF_DEVCFG_Disable_Function](#)
- [USBF_DEVCFG_Enable_Function](#)
- [USBF_DEVCFG_Unbind_Function](#)

Related Topics

[USB Host and Functions](#)

USBF_DEVCFG_Activate_Device

Activates the USB device by enabling the DP pull-up so that it is ready to be connected with the USB host and communicate with it.

Usage

```
STATUS USBF_DEVCFG_Activate_Device (NU_USB_DEVICE *device)
```

Arguments

- device
Pointer to NU_USB_DEVICE control block.

Return Values

- NU_SUCCESS
Operation completed successfully.
- NU_USB_INVLD_ARG
One or more input arguments are invalid.

Related Topics

[USB Function Device Configuration Services](#)

USBF_DEVCFG_Add_Config_String

This function adds the configuration string descriptor to a particular configuration descriptor.

Usage

```
STATUS USBF_DEVCFG_Add_Config_String (UINT8      config_index,  
                                     NU_USB_DEVICE *device,  
                                     CHAR          *string)
```

Arguments

- **config_index**
Index number of configuration.
- **device**
Pointer to NU_USB_DEVICE control block.
- **string**
Character array containing an ASCII string.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[USB Function Device Configuration Services](#)

USBF_DEVCFG_Add_Function

This function registers descriptors (interface and class specific) from the device configuration information base with the USB device configuration repository.

Usage

```
STATUS USBF_DEVCFG_Add_Function (UINT8  config_index,
                                UINT8  *fs_raw_descriptor,
                                UINT16  fs_raw_desc_len,
                                UINT8  *hs_raw_descriptor,
                                UINT16  hs_raw_desc_len,

                                #if ( CFG_NU_OS_CONN_USB_COM_STACK_SS_ENABLE == NU_TRUE )
                                UINT8  *ss_raw_descriptor,
                                UINT16  ss_raw_desc_len,
                                #endif
                                USBF_FUNCTION **func_out)
```

Arguments

- **config_index**
Configuration number, to which this interface descriptor belongs.
- **fs_raw_descriptor**
Array containing full speed raw descriptor.
- **fs_raw_desc_len**
Length of full speed raw descriptor
- **hs_raw_descriptor**
Array containing high speed raw descriptor.
- **hs_raw_desc_len**
Length of high speed raw descriptor
- **ss_raw_descriptor**
Array containing super speed raw descriptor.
- **ss_raw_desc_len**
Length of super speed raw descriptor
- **func_out**
Pointer to the USBF_FUNCTION control block returned to caller. The caller will use this pointer as handle for any subsequent operations.

Return Values

- **NU_SUCCESS**
Operation completed successfully.

- NU_USB_INVLD_ARG

One or more input arguments are invalid.

Description

Class drivers call this function during its creation to register the interface.

Descriptors are only populated in NU_USB_DEVICE when the application tries to use a particular class.

The signature of the function depends upon the current configuration of the stack. If USB stack is configured for super speed then the caller must supply super speed descriptors as well.

Related Topics

[USB Function Device Configuration Services](#)

USBF_DEVCFG_Add_Intf_String

This function adds the interface string descriptor to a particular interface descriptor.

Usage

```
STATUS USBF_DEVCFG_Add_Intf_String (USBF_FUNCTION *usbfunc,  
                                     NU_USB_DEVICE *device,  
                                     CHAR            *string)
```

Arguments

- **usbfunc**
Pointer to the USBF_FUNCTION control block.
- **device**
Pointer to NU_USB_DEVICE control block.
- **string**
Character array containing an ASCII string.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[USB Function Device Configuration Services](#)

USBF_DEVCFG_Bind_Function

This function binds a pre-registered function with a NU_USB_DEVICE instance.

Usage

```
STATUS USBF_DEVCFG_Bind_Function (USBF_FUNCTION *cb)
```

Arguments

- **cb**
Pointer to USBF_FUNCTION control block.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Description

Binding includes adding associated interface and class-specific descriptors in a specific configuration, aligning interface numbers and acquiring endpoints for each alternate setting.

Once registered, a USB function is disabled by default. It is the responsibility of the caller to enable it first before performing a bind operation on it.

Related Topics

[USB Function Device Configuration Services](#)

USBF_DEVCFG_Create_Config

This function creates a new configuration in the NU_USB_DEVICE control block and returns its index as an output argument.

Usage

```
STATUS USBF_DEVCFG_Create_Config (NU_USB_DEVICE *device,
                                  UINT8          config_index_out)
```

Arguments

- **device**
Pointer to NU_USB_DEVICE control block.
- **config_index_out**
Pointer to the index of the newly created configuration when the function returns.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.
- **NU_NOT_PRESENT**
No empty slot is present for creating a new configuration.

Related Topics

[USB Function Device Configuration Services](#)

USBF_DEVCFG_Deactivate_Device

This function deactivates the USB device by disabling the DP pull-up so that it is no longer ready to be connected with the USB host.

Usage

```
STATUS USBF_DEVCFG_Deactivate_Device (NU_USB_DEVICE *device)
```

Arguments

- **device**
Pointer to NU_USB_DEVICE control block.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[USB Function Device Configuration Services](#)

USBF_DEVCFG_Delete_Config

This function deletes a previously created configuration in the NU_USB_DEVICE control block.

Usage

```
STATUS USBF_DEVCFG_Delete_Config (NU_USB_DEVICE *device,
                                  UINT8          config_index)
```

Arguments

- **device**
Pointer to NU_USB_DEVICE control block.
- **config_index**
Pointer to the configuration index to be deleted.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[USB Function Device Configuration Services](#)

USBF_DEVCFG_Delete_Function

This function deletes a previously registered interface descriptor of a class from the USB function device configuration repository.

Usage

```
STATUS USBF_DEVCFG_Delete_Function (USBF_FUNCTION *cb)
```

Arguments

- **cb**
Pointer to USBF_FUNCTION control block.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[USB Function Device Configuration Services](#)

USBF_DEVCFG_Disable_Function

This API disables a USB function from binding with a NU_USB_DEVICE instance so that the associate interface and class-specific descriptors are not included in the configuration descriptor.

Usage

```
STATUS USBF_DEVCFG_Disable_Function (USBF_FUNCTION *cb)
```

Arguments

- **cb**
Pointer to USBF_FUNCTION control block.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[USB Function Device Configuration Services](#)

USBF_DEVCFG_Enable_Function

This API enables a USB function to bind with a NU_USB_DEVICE instance so that the associate interface and class specific descriptors are included in the configuration descriptor.

Usage

```
STATUS USBF_DEVCFG_Enable_Function (USBF_FUNCTION *cb)
```

Arguments

- **cb**
Pointer to USBF_FUNCTION control block.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[USB Function Device Configuration Services](#)

USBF_DEVCFG_Unbind_Function

This functions unbinds a bound function with a NU_USB_DEVICE instance. Once unbound a USB function is disabled and cannot be bound again unless it is enabled again.

Usage

```
STATUS USBF_DEVCFG_Unbind_Function (USBF_FUNCTION *cb)
```

Arguments

- **cb**
Pointer to USBF_FUNCTION control block.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[USB Function Device Configuration Services](#)

Nucleus USB Function Hardware Driver Services

This section provides a detailed reference of the following Nucleus USB Function Hardware Driver Services:

- [_NU_USBF_HW_Create](#)
- [_NU_USBF_HW_Delete](#)

Related Topics

[USB Host and Functions](#)

_NU_USB_F_HW_Create

This function initializes and allocates resources required by the base function controller component.

Usage

```
STATUS _NU_USB_F_HW_Create (NU_USB_F_HW *cb,  
                           CHAR          *name,  
                           UINT32       capability,  
                           UINT8        speed,  
                           VOID         *base_address,  
                           UINT8        num_irq,  
                           INT          *irq,  
                           const VOID *dispatch)
```

Arguments

- **cb**
Pointer to the control block of the function controller.
- **name**
Name of the control block.
- **capability**
Hardware capabilities.
- **speed**
USB speeds supported.
- **base_address**
Base address in the memory map.
- **num_irq**
Number of IRQs supported by the function controller.
- **irq**
List of supported IRQs.
- **dispatch**
Dispatch table of the function controller.

Return Values

- **NU_SUCCESS**
Indicates that the component has been initialized successfully.
- **NU_NOT_PRESENT**
Indicates no more USB objects could be created because of a configuration problem.

Related Topics

[Nucleus USB Function Hardware Driver Services](#)

_NU_USB_F_H_W_Delete

This function releases all of the resources acquired by the [_NU_USB_F_H_W_Create](#) function.

Usage

```
STATUS _NU_USB_F_H_W_Delete (VOID *hw)
```

Arguments

- **hw**
Pointer to control block of the function controller.

Return Values

- **NU_SUCCESS**
Indicates that the component is successfully deleted.

Related Topics

[Nucleus USB Function Hardware Driver Services](#)

Nucleus USB Function STACK Services

This section provides a detailed reference of the following Nucleus USB Function STACK Services:

- [NU_USBF_STACK_Attach_Device](#)
- [NU_USBF_STACK_Create](#)
- [NU_USBF_STACK_Detach_Device](#)
- [NU_USBF_STACK_New_Transfer](#)
- [NU_USBF_STACK_Notify](#)
- [NU_USBF_STACK_Speed_Change](#)

Related Topics

[USB Host and Functions](#)

NU_USBFS_STACK_Attach_Device

This function registers a USB device with the stack.

Usage

```
STATUS NU_USBFS_STACK_Attach_Device (NU_USBFS_STACK *cb,  
                                     NU_USB_DEVICE *usb_device)
```

Arguments

- **cb**
Pointer to the stack control block.
- **usb_device**
Device to be registered.

Return Values

- **NU_SUCCESS**
If the stack could be initialized successfully, otherwise an error code is returned by the called functions.

Description

Prior to making this call, [NU_USB_DEVICE_Set_Hw](#) should be invoked to set the hardware controller associated with the device. This function:

- enlists the device in the stack's device list.
- parses the raw-configuration descriptors of the device.
- finds out the capabilities of the associated hardware controller and does the necessary initialization for those control transfers that are handled by the hardware independently.
- enables the interrupts on the associated hardware controller.

Related Topics

[Nucleus USB Function STACK Services](#)

NU_USBFS_STACK_Create

This function initializes the stack control block and the framework for driving the hardware controllers and class drivers.

Usage

```
STATUS NU_USBFS_STACK_Create (NU_USBFS_STACK *cb,  
                             CHAR             *name)
```

Arguments

- **cb**
Stack control block to be initialized.
- **name**
Pointer to a 7-character name for the USB device stack. The name must be null-terminated.

Return Values

- **NU_SUCCESS**
If the stack could be initialized successfully, otherwise an error code is returned by the called functions.

Related Topics

[Nucleus USB Function STACK Services](#)

NU_USBFS_STACK_Detach_Device

This function deregisters a USB device with the stack.

Usage

```
STATUS NU_USBFS_STACK_Detach_Device (NU_USBFS_STACK *cb,  
                                     NU_USB_DEVICE *usb_device)
```

Arguments

- `cb`
Pointer to the stack control block.
- `usb_device`
Device to be registered.

Return Values

- `NU_SUCCESS`
Stack has successfully de-listed the requested device.

Description

This call should be made for a device that is already attached to the stack through the [NU_USBFS_STACK_Attach_Device](#). This function resets all the device related information in stack's device list.

Related Topics

[Nucleus USB Function STACK Services](#)

NU_USBFS_STACK_New_Transfer

This function identifies the recipient class driver of a device and reports an unscheduled transfer on an endpoint.

Usage

```
STATUS NU_USBFS_STACK_New_Transfer (NU_USB_STACK *cb,  
                                   NU_USBFS_HW   *fc,  
                                   UINT8         bEndpointAddress)
```

Arguments

- **cb**
Pointer to NU_USB_STACK control block.
- **fc**
Pointer to NU_USBFS_HW control block.
- **bEndpointAddress**
Endpoint which receives the token but no transfer is scheduled on it.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_NOT_PRESENT**
Associated NU_USB_DEVICE control block was not found.
- **NU_USB_INVLD_ARG**
One or more input arguments invalid.

Description

The USB function controller driver calls this function when a token (usually OUT) is present on an endpoint but the USB function stack has not scheduled any transfer for it.

This API only notifies the class driver of an unscheduled transfer; the class driver is responsible for determining what further action to take.

Related Topics

[Nucleus USB Function STACK Services](#)

NU_USBFS_STACK_Notify

This function identifies all registered class drivers and passes this notification to every registered class driver. The USB function controller driver calls this function when it receives a bus event like RESET, SUSPEND and RESUME.

Usage

```
STATUS NU_USBFS_STACK_Notify (NU_USBFS_STACK *cb,
                             NU_USBFS_HW    *fc,
                             UINT32         event)
```

Arguments

- **cb**
Pointer to NU_USBFS_STACK control block.
- **fc**
Pointer to NU_USBFS_HW control block.
- **event**
Bus event to be reported.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_NOT_PRESENT**
Associated NU_USB_DEVICE control block was not found.
- **NU_USB_INVLD_ARG**
One or more input arguments are invalid.

Related Topics

[Nucleus USB Function STACK Services](#)

NU_USBFS_STACK_Speed_Change

This function notifies the USB function controller driver of speed at which the USB device is enumerated.

Usage

```
STATUS NU_USBFS_STACK_Speed_Change (NU_USB_STACK *cb,  
                                     NU_USBFS_HW *fc,  
                                     UINT32 speed,  
  
                                     #if CFG_NU_OS_CONN_USB_COM_STACK_SS_ENABLE  
                                     UINT8 maxp0  
                                     #endif  
)
```

Arguments

- **cb**
Pointer to NU_USB_STACK control block.
- **fc**
Pointer to NU_USBFS_HW control block.
- **speed**
Speed at which this USB host has enumerated this device.
- **maxp0**
Maximum packet size of the default control endpoint according to the reported speed. This parameter is only visible if the USB stack is configured for Super Speed operations.

Return Values

- **NU_SUCCESS**
Operation completed successfully.
- **NU_USB_INVLD_ARG**
One or more input arguments invalid.

Description

USB is a backwards-compatible bus. A high speed device can operate either with Full speed or High speed hosts. USB devices may need to change certain behavior according to the speed on which they are enumerated.

The USB function controller driver calls this API to notify the speed on which this USB device is enumerated. Usage of this API is different when the stack is configured for Super Speed USB operation. This is because, in that case, function hardware also needs to report the maximum packets size of default control endpoint according to the reported speed.

Related Topics

[Nucleus USB Function STACK Services](#)

Nucleus USB Function Services

This section provides a detailed reference of the following Nucleus USB Function Services:

- [NU_USBF_Create](#)
- [NU_USBF_Delete](#)
- [NU_USBF_Validate_Drvr_Object](#)
- [NU_USBF_Validate_Hw_Object](#)
- [NU_USBF_Validate_Stack_Object](#)
- [NU_USBF_Validate_User_Object](#)

Related Topics

[USB Host and Functions](#)

NU_USBF_Create

This function initializes the USB device software environment and enables the services of all other Nucleus USB function software components. This API must therefore be the first API of the Nucleus USB function software invoked in the application before invoking the APIs of any other Nucleus USB function software.

Usage

```
STATUS NU_USBF_Create (NU_USBF *singleton_cb)
```

Arguments

- **singleton_cb**
Pointer to the control block of NU_USBF.

Return Values

- **NU_SUCCESS**
Indicates that the initialization was successful.

Related Topics

[Nucleus USB Function Services](#)

NU_USB_Delete

This function uninitializes and tears down the Nucleus USB function software environment. Any further access to any of the USB function software services leads to undefined behavior.

Usage

```
STATUS NU_USB_Delete (VOID)
```

Return Values

- **NU_SUCCESS**
Indicates that the singleton was uninitialized successfully.

Related Topics

[Nucleus USB Function Services](#)

NU_USB_F_Validate_Drvr_Object

This function checks if the supplied class driver object is valid and known to the USB function software.

Usage

```
STATUS NU_USB_F_Validate_Drvr_Object (NU_USB_F_DVR *driver)
```

Arguments

- driver
Class driver object to be validated.

Return Values

- NU_SUCCESS
Given object is valid.
- NU_USB_INVLD_ARG
Object could not be found in the subsystem.

Related Topics

[Nucleus USB Function Services](#)

NU_USB_F_Validate_Hw_Object

This function checks if the supplied hardware object is valid and known to the USB function software.

Usage

```
STATUS NU_USB_F_Validate_Hw_Object (NU_USB_F_HW *hw)
```

Arguments

- hw
Controller object to be validated.

Return Values

- NU_SUCCESS
Given object is valid.
- NU_USB_INVLD_ARG
Object could not be found in the subsystem.

Related Topics

[Nucleus USB Function Services](#)

NU_USBFS_Verify_Stack_Object

This function checks if the supplied stack object is valid and known to the USB function software.

Usage

```
STATUS NU_USBFS_Verify_Stack_Object (NU_USBFS_STACK *stack)
```

Arguments

- `stack`
Stack object to be validated.

Return Values

- `NU_SUCCESS`
Given object is valid.
- `NU_USB_INVLD_ARG`
Object could not be found in the subsystem.

Related Topics

[Nucleus USB Function Services](#)

NU_USB_F_Validate_User_Object

This function checks if the supplied user object is valid and known to the USB Function software.

Usage

```
STATUS NU_USB_F_Validate_User_Object (NU_USB_F_USER *user)
```

Arguments

- **user**
User object to be validated.

Return Values

- **NU_SUCCESS**
Given object is valid.
- **NU_USB_INVLD_ARG**
Object could not be found in the subsystem.

Related Topics

[Nucleus USB Function Services](#)

Nucleus USB Function User Driver Services

This section provides a detailed reference of the following Nucleus USB Function User Driver Services:

- [NU_USBF_USER_New_Command](#)
- [NU_USBF_USER_New_Transfer](#)
- [NU_USBF_USER_Notify](#)
- [NU_USBF_USER_Transfer_Complete](#)
- [_NU_USBF_USER_Create](#)
- [_NU_USBF_USER_Delete](#)

Related Topics

[USB Host and Functions](#)

NU_USBF_USER_New_Command

This function processes a new command from the host. The class driver should call this function for all commands from the host whose processing has been delegated to its user driver.

Usage

```
STATUS NU_USBF_USER_New_Command (NU_USBF_USER *cb,  
                                NU_USBF_DRV *drv,  
                                VOID *handle,  
                                UINT8 *command,  
                                UINT16 cmd_len,  
                                UINT8 **data_out,  
                                UINT32 *data_len_out)
```

Arguments

- **cb**
User control block for which the command is meant.
- **drv**
Class driver invoking this function.
- **handle**
Identifies for the logical function to which the command is directed.
- **command**
Memory location where the command block is stored.
- **cmd_len**
Length of the command block.
- **data_out**
Memory location where the data pointer for the transfer is to be stored.
- **data_len_out**
Memory location where the length of data to be transferred, in bytes, must be filled.

Return Values

- **NU_SUCCESS**
Indicates that the command has been processed successfully.
- **NU_USB_NOT_SUPPORTED**
Indicates that the command is unsupported.
- **NU_USB_INVLD_ARG**
Indicates a malformed command block.

Description

This user driver function processes the command identified by the command parameter. The length of the command block, in bytes, is expected in the cmd_len parameter.

If there is any data transfer in response to a command, then the location corresponding to the data to be transferred, either to/from host is filled in the data_out parameter. The length of the data to be transferred is filled in the location pointed to by the data_len_out parameter.

If there is no data transfer associated with the command, then the location pointed to by the data is filled NU_NULL.

For unknown and unsupported commands, this function returns the appropriate error status.

Related Topics

[Nucleus USB Function User Driver Services](#)

NU_USBF_USER_New_Transfer

The class driver with a user driver that handles commands received from host should call this function New_Transfer notification if it has no outstanding data with which to reply.

Usage

```
STATUS NU_USBF_USER_New_Transfer (NU_USBF_USER *cb,  
                                  NU_USBF_DRV *drv,  
                                  VOID *handle,  
                                  UINT8 **data_out,  
                                  UINT32 *data_len_out)
```

Arguments

- **cb**
User control block.
- **drv**
Class driver invoking this function.
- **handle**
Identifies for the logical function to which the transfer is directed.
- **data_out**
Memory location where the data pointer for the transfer is to be stored
- **data_len_out**
Memory location where the length of data to be transferred, in bytes, must be filled.

Return Values

- **NU_SUCCESS**
Indicates that the function has executed successfully. The class driver will have data provided by the user driver that can be forwarded to the host.
- **NU_USB_INVLD_ARG**
Indicates an unexpected transfer request from the host.
- **NU_USB_NOT_SUPPORTED**
Indicates that the new transfer requests are not supported by the user.

Related Topics

[Nucleus USB Function User Driver Services](#)

NU_USB_USER_Notify

This function carries out class specific processing for the USB event.

Usage

```
STATUS NU_USB_USER_Notify (NU_USB_USER *cb,  
                           NU_USB_DRIVER *drv,  
                           VOID          *handle,  
                           UINT32        event)
```

Arguments

- **cb**
Pointer to a user control block for which the event is meant.
- **drv**
Pointer to a class driver invoking this function.
- **handle**
Pointer to a the logical function to which the notification is directed.
- **event**
The USB event that has occurred. USBF_EVENT_SUSPEND or USBF_EVENT_RESET.

Return Values

- **NU_SUCCESS**
Indicates that the event has been processed successfully.
- **NU_USB_INVLD_ARG**
Indicates an unexpected event.
- **NU_USB_NOT_SUPPORTED**
Indicates that the event notifications are not supported by the user.

Description

The class driver may call this API to notify the USB event to its user driver. This user driver function processes a USB event. Examples of such USB events include suspend and reset.

Related Topics

[Nucleus USB Function User Driver Services](#)

NU_USBFS_USER_Transfer_Complete

This function notifies the user driver that the transfer of the previously submitted data has completed.

Usage

```
STATUS NU_USBFS_USER_Transfer_Complete (NU_USBFS_USER *cb,  
                                         NU_USBFS_DVR *dvr,  
                                         VOID *handle,  
                                         UINT8 *completed_data,  
                                         UINT32 completed_data_len,  
                                         UINT8 **data_out,  
                                         UINT32 *data_len_out)
```

Arguments

- **cb**
Pointer to a user control block.
- **dvr**
Pointer to the class driver invoking this function.
- **handle**
Identifies for the logical function to which the notification is directed.
- **completed_data**
Memory location to/from where the data has been transferred.
- **completed_data_len**
Length of data transferred, in bytes.
- **data_out**
Memory location where the data pointer for the transfer is to be stored, if pending.
- **data_len_out**
Memory location where the length of data to be transferred, in bytes, is to be filled.

Return Values

- **NU_SUCCESS**
Indicates that the function has executed successfully.
- **NU_USB_INVLD_ARG**
Indicates an unexpected transfer completion.
- **NU_USB_NOT_SUPPORTED**
Indicates that the transfer completion notifications are not supported by the user.

Description

The class driver must call this function from its IRP callback function if it has a user driver that provides data for the IRP.

The completed data transfer is described in the `completed_data` and the `completed_data_out` parameters. The `completed_data` contains the pointer to the memory location to/from which `completed_data_out` bytes have been transferred from/to the host.

If there is still data to be transferred to the previous command, then the location corresponding to the data to be transferred, either to/from host is filled in the `data_out` parameter. The length of the data to be transferred is filled in the location pointed to by the `data_len_out` parameter.

If there is no data transfer associated with the command, then, the location pointed to by the `data` is filled in with `NULL`.

Related Topics

[Nucleus USB Function User Driver Services](#)

_NU_USBFS_USER_Create

This function initializes the user driver component.

Usage

```
STATUS _NU_USBFS_USER_Create (NU_USBFS_USER *cb,  
                              CHAR           *name,  
                              UINT8         subclass,  
                              UINT8         protocol,  
                              const VOID    *dispatch)
```

Arguments

- **cb**
User control block.
- **name**
Name of the user.
- **subclass**
USB defined subclass code supported by the user.
- **protocol**
USB defined protocol code supported by the user.
- **dispatch**
User dispatch table.

Return Values

- **NU_SUCCESS**
Indicates that the initialization has been successfully processed.
- **NU_NOT_PRESENT**
Indicates no more USB objects could be created because of a configuration problem.
- **NU_USB_INVLD_ARG**
Indicates an incorrect parameter to this function.

Description

The subclass and protocol codes are defined by the USB. These indicate the capabilities of the device, in a family of devices. An example would be a SCSI device in a family of mass storage devices. If the protocol code is not applicable for a specific user, it can be initialized to zero (0). Subclass code, however cannot be zero.

Related Topics

[Nucleus USB Function User Driver Services](#)

_NU_USB_USER_Delete

This function uninitializes the user.

Usage

```
STATUS _NU_USB_USER_Delete (VOID *user)
```

Arguments

- **user**
User to be uninitialized.

Return Values

- **NU_SUCCESS**
Indicates that the user has been uninitialized successfully.

Related Topics

[Nucleus USB Function User Driver Services](#)

Dispatch Table Reference

This section describes the dispatch tables for the Nucleus USB software components. It describes the functions that can be installed in the dispatch table. It also provides the default implementation references for each of the entries in the dispatch table.

This section can be skipped entirely, if you do not plan to develop your own class drivers, user drivers, or hardware drivers.

NU_USB

Dispatch Table

The dispatch table for the NU_USB component is listed as follows:

```
typedef struct usb_dispatch
{
    STATUS (*Delete) (VOID *cb);
    STATUS (*Get_Name) (NU_USB *cb, CHAR *name_out);
    STATUS (*Get_Object_Id) (NU_USB *cb, UINT32 *id_out);
}
NU_USB_DISPATCH;
```

Delete

Every component that derives from NU_USB must implement this function to release all resources acquired by it during the creation of the component.

Usage

```
STATUS (*Delete) (VOID *cb)
```

Arguments

- **cb**
Pointer to the control block. Can be any derivative of NU_USB.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_NOT_PRESENT**
Indicates that the control block to be deleted is invalid.

Description

In its implementation, each component must call the `_<Component_Name>_Delete` API of the component from which it derives.

For example, `NU_USBH_OHCI` derives from `NU_USBH_HW`, so `_NU_USBH_OHCI_Delete` must call [_NU_USBH_HW_Delete](#). `NU_USBH_HW` derives from `NU_USB_HW`, so `_NU_USBH_HW_Delete` calls `_NU_USB_HW_Delete`.

`_NU_USB_Delete` is the delete implementation of the `NU_USB` component.

Get_Name

This function must return the name (NULL terminated 7-character string) assigned to the component during its creation. The default implementation for the function is `_NU_USB_Get_Name`. This function obtains the name string from the component's control block.

Usage

```
STATUS (*Get_Name) (NU_USB *cb,  
                   CHAR *name_out)
```

Arguments

- `cb`
Pointer to `NU_USB` or its derivatives control block.
- `name_out`
Pointer to the array to hold the NULL terminated 7 character string.

Return Values

- `NU_SUCCESS`
Indicates successful completion of the service.

Get_Object_Id

This function must return the object ID assigned during the creation of the component. The default implementation for the function is `_NU_USB_Get_Object_Id`. This function obtains the ID from the component's control block.

Usage

```
STATUS (*Get_Object_Id) (NU_USB *cb,  
                        UINT32 *id_out)
```

Arguments

- `cb`
Pointer to the `NU_USB` or its derivatives control block.
- `id_out`
Pointer to the variable to hold the object ID.

Return Values

- `NU_SUCCESS`
Indicates successful completion of the service.

NU_USB_DRV

Dispatch Table

```
typedef struct usb_driver_dispatch
{
    NU_USB_DISPATCH dispatch;

    STATUS (*Examine_Intf) (NU_USB_DRV *cb, NU_USB_INTF_DESC *intf);

    STATUS (*Examine_Device) (NU_USB_DRV *cb,
                             NU_USB_DEVICE_DESC *device);

    STATUS (*Get_Score) (NU_USB_DRV *cb,
                        UINT8 *score_out);

    STATUS (*Initialize_Device) (NU_USB_DRV *cb,
                                NU_USB_STACK *stack,
                                NU_USB_DEVICE *device);

    STATUS (*Initialize_Interface) (NU_USB_DRV *cb,
                                    NU_USB_STACK *stack,
                                    NU_USB_DEVICE *device,
                                    NU_USB_INTF *intf);

    STATUS (*Disconnect) (NU_USB_DRV *cb,
                         NU_USB_STACK *stack,
                         NU_USB_DEVICE *device);
}
NU_USB_DRV_DISPATCH;
```

Initialize_Device

This is a device class driver function that finds all of the necessary interfaces and pipes, chooses a configuration, and then claims the interface.

Usage

```
STATUS (*Initialize_Device) (NU_USB_DRV_R    *cb,  
                             NU_USB_STACK    *stack,  
                             NU_USB_DEVICE    *device)
```

Arguments

- **cb**
Pointer to the driver control block.
- **stack**
Pointer to the stack control block that has invoked this function.
- **device**
Pointer to the device control block, that needs to be initialized.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Description

If the device class driver has supported user drivers, you will need to notify the user drivers of this new device by invoking the user driver's Connect function.

This function is invoked by the stack once `NU_USB_DRV_R_Examine_Device` returns `NU_SUCCESS`. Only a USB Device class driver needs to implement this function. An interface class driver can have this dispatch table entry as `NU_NULL`.

No default implementation exists for this function. Every specialization of `NU_USB_DRV_R` that is a USB device class driver has to implement this for itself.

Initialize_Interface

This driver function finds all the necessary pipes, chooses an alternate setting for the interface, and claims the interface.

Usage

```
STATUS (*Initialize_Interface) (NU_USB_DRV_R    *cb,  
                               NU_USB_STACK    *stack,  
                               NU_USB_DEVICE    *device,  
                               NU_USB_INTF      *intf)
```

Arguments

- **cb**
Pointer to the driver control block.
- **stack**
Pointer to the stack control block that has invoked this function.
- **device**
Pointer to the device control block, that needs to be initialized.
- **intf**
Pointer to the interface control block, that needs to be initialized.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Description

If the driver has supporting user drivers, you will need to notify the user drivers of this new interface by invoking user driver's Connect function.

This function is invoked by the stack once `NU_USB_DRV_R_Examine_Intf` returns `NU_SUCCESS`. Only a USB interface class driver needs to implement this function. A USB device class driver can have this dispatch table entry as `NU_NULL`.

No default implementation exists for this function. Every specialization of `NU_USB_DRV_R` that is a USB interface class driver has to implement this for itself.

Examine_Device

This function examines the device descriptor of a device to see the suitability this driver.

Usage

```
STATUS (*Examine_Device) (NU_USB_DRV_R      *cb,  
                          NU_USB_DEVICE_DESC *device)
```

Arguments

- **cb**
Pointer to the driver control block.
- **device**
Pointer to the device descriptor that needs to be matched against.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.
- **NU_USB_REJECTED**
Indicates that the driver did not find a match and the device is rejected by the driver.

Description

A driver, whose match spec has the vendor id/product id/release number fields, will have this function defined in the dispatch table.

On the host software, this function is invoked by the stack in the process of finding a best matching driver for the device on the function software. This is invoked by the stack whenever the SET_CONFIGURATION command is received from the host.

A default implementation of this function is provided by `_NU_USB_DRV_R_Examine_Device`. This default implementation matches the driver's match spec against the vendor ID/product id/release number fields found in the device descriptor. If a match is found it returns `NU_SUCCESS`, otherwise else it returns `NU_USB_REJECTED`.

You can use this function to define a different way of selecting devices for the driver if the default behavior described is not suitable.

Examine_Intf

This function examines an interface descriptor to see the suitability of its driver.

Usage

```
STATUS (*Examine_Intf) (NU_USB_DRV_R      *cb,  
                        NU_USB_INTF_DESC *intf)
```

Arguments

- **cb**
Pointer to the driver control block.
- **intf**
Pointer to the interface descriptor that needs to be matched against the driver's match spec.

Return Values

- **NU_SUCCESS**
Indicates that the driver found a match and the device is preliminarily accepted by the driver.
- **NU_USB_REJECTED**
Indicates that the driver did not find a match and the device is rejected by the driver.

Description

A driver, whose match spec has an interface class/subclass/protocol field, will have this function defined in its dispatch table.

On the host software, this function is invoked by the stack in the process of finding the best matching driver for the device. On the function software, this is invoked by the stack whenever the SET_CONFIGURATION command is received from the host.

A default implementation of this function is provided by `_NU_USB_DRV_R_Examine_Intf`. This default implementation matches the driver's match spec against the class/subclass/protocol number fields found in the device descriptor. If a match is found it returns `NU_SUCCESS`, otherwise it returns `NU_USB_REJECTED`.

You can use this function to define a different way of selecting interfaces for your driver, if the default behaviour described is not suitable for your case.

Get_Score

This function returns the score assigned to the driver during create.

Usage

```
STATUS (*Get_Score) (NU_USB_DRV_R *cb,  
                    UINT8         *score_out)
```

Arguments

- **cb**
Pointer to the driver control block.
- **score_out**
Pointer to the variable to hold the value of the score.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Description

This score is the sorting key for the drivers registered with the stack. When the stack is in process of finding a best matching driver for the newly found interface/device, it attempts each driver in the descending order of their scores until it finds a driver that returns **NU_SUCCESS** for **NU_USB_Examine_Device** or **NU_USB_Examine_Intf** call.

The stack uses this function to know the score of the driver. The default implementation of this function is provided by **_NU_USB_DRV_R_Get_Score**. The default implementation assigns credit to each field that is set in the match spec of the driver. And the score of the driver is sum total of the credits for each specified value in the match spec. The value in the brackets is the credit assigned to the field by this default implementation.

- **idVendor** (10)
- **idProduct** (4)
- **bcdDeviceLow** and **bcdDeviceHigh** (3)
- **bInterfaceClass** (3)
- **bInterfaceSubClass** (2)
- **bInterfaceProtocol** (1)

Thus, a match spec with **idVendor** and **idProduct** specified would have a score of 14, while a match spec with **bInterfaceClass**, **bInterfaceSubClass**, and **bInterface protocol** specified would have a score of 6. The default implementation hence guarantees a higher score for those based

on vendor/product/release number than those based on class/subclass/protocol. Thus the more the fields that are set in the match spec the higher the score.

Implement this function if different behavior than that described above is desired.

NU_USB_USER

Dispatch Table

The following is the dispatch table for the NU_USB_USER component.

```
typedef struct usb_user_dispatch
{
    NU_USB_DISPATCH usb_dispatch;

    STATUS (*Connect) (NU_USB_USER *user,
                       NU_USB_DRV *class_driver,
                       VOID *handle);

    STATUS (*Disconnect) (NU_USB_USER *user,
                          NU_USB_DRV *class_driver,
                          VOID *handle);
}
NU_USB_USER_DISPATCH;
```


Connect

This function establishes a connection to a device so communication with that device can begin.

Usage

```
STATUS (*Connect) (NU_USB_USER *cb,  
                  NU_USB_DRV *class_driver,  
                  VOID *handle)
```

Arguments

- **cb**
Pointer to the user driver control block.
- **class_driver**
Pointer to the class driver control block.
- **handle**
Context information that identifies the newly connected device. The handle is an opaque ID. User drivers are required to use this handle in all APIs of the class driver that may be used to communicate with the device.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Description

A class driver calls this function when a device matching the capabilities of the user is detected. This is invoked during the driver matching process.

This function may implement its specific initialization steps or may contact the device to get further details or to set it up for future transactions. There is no default implementation of this function.

Disconnect

This function disconnects a device from the bus.

Usage

```
STATUS (*Disconnect) (NU_USB_USER *cb,  
                     NU_USB_DRV *class_driver,  
                     VOID *handle)
```

Arguments

- **cb**
Pointer to the user driver control block.
- **class_driver**
Pointer to the class driver control block.
- **handle**
Context information.

Return Values

- **NU_SUCCESS**
Indicates successful completion of the service.

Description

A class driver calls this function when a device previously owned by the user driver is disconnected from the bus. The user driver may use this callback to perform any cleanup operations required. There is no default implementation provided.

NU_USB_USER

Dispatch Table

The following is the dispatch table for the NU_USB component.

```
typedef struct nu_usb_user_dispatch
{
    NU_USB_USER_DISPATCH dispatch;

    STATUS (*New_Command) (NU_USB_USER *cb,
                           NU_USB_DRVR *drv,
                           VOID *handle,
                           UINT8 *command,
                           UINT16 cmd_len,
                           UINT8 **data,
                           UINT32 *data_len);

    STATUS (*New_Transfer) (NU_USB_USER *cb,
                           NU_USB_DRVR *drv,
                           VOID *handle,
                           UINT8 **data,
                           UINT32 *data_len);

    STATUS (*Transfer_Complete) (NU_USB_USER *cb,
                                 NU_USB_DRVR *drv,
                                 VOID *handle,
                                 UINT8 *completed_data,
                                 UINT32 completed_data_len,
                                 UINT8 **data,
                                 UINT32 *data_len);

    STATUS (*Notify) (NU_USB_USER *cb,
                     NU_USB_DRVR *drv,
                     VOID *handle,
                     UINT32 event);
}
NU_USB_USER_DISPATCH;
```

New_Command

Class drivers invoke this function when they receive a command from the host meant for processing by the user driver. The user drivers must process the command.

Usage

```
STATUS (*New_Command) (NU_USBF_USER *cb,  
                      NU_USBF_DRV *drv,  
                      VOID *handle,  
                      UINT8 *command,  
                      UINT16 cmd_len,  
                      UINT8 **data_out,  
                      UINT32 *data_len_out)
```

Arguments

- **cb**
User control block for which the command is meant.
- **drv**
Class driver invoking this function.
- **handle**
Identifies for the logical function to which the command is directed.
- **command**
Memory location where the command block is stored.
- **cmd_len**
Length of the command block.
- **data_out**
Memory location where the data pointer for the transfer is to be stored.
- **data_len_out**
Memory location where the length of data to be transferred, in bytes, must be filled.

Return Values

- **NU_SUCCESS**
Indicates that the command was processed successfully.
- **NU_USB_NOT_SUPPORTED**
Indicates that the command is unsupported.
- **NU_USB_INVLD_ARG**
Indicates a malformed command block.

Description

These commands are class/subclass specific. This function must process the command identified by the command parameter. The length of the command block, in bytes is expected in the cmd_len parameter.

If there is any data transfer in response to a command, the location corresponding to the data to be transferred, either to/from the host is filled in the data_out parameter. The length of the data to be transferred is filled in the location pointed to by the data_len_out parameter.

If there is no data transfer associated with the command, the location pointed to by the data is filled in with NU_NULL. For unknown and unsupported commands, this function returns the appropriate error status.

New_Transfer

Class drivers invoke this function when they receive in/out tokens from the host and they do not have any data/buffer for transfers.

Usage

```
STATUS (*New_Transfer) (NU_USBF_USER *cb,  
                        NU_USBF_DRVVR *drvvr,  
                        VOID *handle,  
                        UINT8 **data_out,  
                        UINT32 *data_len_out)
```

Arguments

- **cb**
User control block.
- **drvvr**
Class driver invoking this function.
- **handle**
Identifies for the logical function to which the transfer is directed.
- **data_out**
Memory location where the data pointer for the transfer is to be stored
- **data_len_out**
Memory location where the length of data to be transferred, in bytes, must be filled.

Return Values

- **NU_SUCCESS**
Indicates that the function was executed successfully.
- **NU_USB_INVLD_ARG**
Indicates an unexpected transfer request from the host.
- **NU_USB_NOT_SUPPORTED**
Indicates that the new transfer requests are not supported by the user.

Description

User drivers may process this callback as follows:

- If there is any data transfer in response to a previously received command, then the location corresponding to the data to be transferred, either to/from host is filled in the **data_out** parameter. The length of the data to be transferred is filled in the location pointed to by the **data_len_out** parameter.
- If there is no data transfer required, this function must return **NU_USB_INVLD_ARG**. No default implementation is provided.

Notify

Class drivers invoke this function to notify the user driver of a USB event. Examples of such USB events include suspend and reset. No default implementation is provided.

Usage

```
STATUS (*Notify) (NU_USBF_USER *cb,  
                  NU_USBF_DRVVR *drvvr,  
                  VOID *handle,  
                  UINT32 event)
```

Arguments

- **cb**
User control block for which the event is meant.
- **drvvr**
Class driver invoking this function.
- **handle**
Identifies for the logical function to which the notification is directed.
- **event**
The USB event that has occurred. The event may be `USB_EVENT_RESET` or `USB_EVENT_SUSPEND`

Return Values

- **NU_SUCCESS**
Indicates that the event was processed successfully.
- **NU_USB_INVLD_ARG**
Indicates an unexpected event.
- **NU_USB_NOT_SUPPORTED**
Indicates that the event notifications are not supported by the user.

Transfer_Complete

Class drivers invoke this function when a transfer, previously submitted by the user driver is complete.

Usage

```
STATUS (*Transfer_Complete) (NU_USBF_USER *cb,  
                             NU_USBF_DRV  *drv,  
                             VOID          *handle,  
                             UINT8         *completed_data,  
                             UINT32        completed_data_len,  
                             UINT8         **data_out,  
                             UINT32        *data_len_out)
```

Arguments

- **cb**
User control block.
- **drv**
Class driver invoking this function.
- **handle**
Identifies for the logical function to which the notification is directed.
- **completed_data**
Memory location to/from where the data has been transferred.
- **completed_data_len**
Length of data transferred, in bytes.
- **data_out**
Memory location where the data pointer for the transfer is to be stored, if pending.
- **data_len_out**
Memory location where the length of data to be transferred, in bytes, is to be filled.

Return Values

- **NU_SUCCESS**
Indicates that the function was executed successfully.
- **NU_USB_INVLD_ARG**
Indicates an unexpected transfer completion.
- **NU_USB_NOT_SUPPORTED**
Indicates that the transfer completion notifications are not supported by the user.

Description

The completed data transfer is described in the `completed_data` and the `completed_data_out` parameters. The `completed_data` contains the pointer to the memory location to/from which `completed_data_out` bytes have been transferred from/to host.

If there is still data to be transferred to the previous command, the location corresponding to the data to be transferred, either to/from host is filled in the `data_out` parameter. The length of the data to be transferred is filled in the location pointed to by the `data_len_out` parameter.

If there is no data transfer associated with the command, the location pointed to by the data is filled in with NULL. No default implementation is provided.

Nucleus USB Function Stack Initialization

Nucleus USB function stack initialization starts when system initialization calls the entry function of Nucleus USB function stack.

Initially, USB function stack reads configurable parameters from registry. These parameters are described in .metadata file of Nucleus USB function stack and affect the behavior of USB function device. [Table 12-1](#) shows the configurable parameters are read from registry during initialization.

Table 12-1. Configurable Parameters Read from Registry

Parameter Name	Description
'ncontroller'	Number of USB hardware controllers present in system.
'PID'	Product ID of device
'VID'	Vendor ID of device
'manuf_string'	Manufacturer string
'product_string'	Product string
'serial_number'	Serial number string

After reading configuration parameters, USB function stack looks for USB function hardware devices registered with device manager with USBFHW class GUID.

Upon successfully opening a USB function hardware device, USB function stack initializes the hardware and add to stacks hardware list by calling [NU_USB_STACK_Add_Hw](#) function.

When a hardware device is successfully initialized, USB function stack initializes necessary parameters of device descriptor like *bcdUSB*, maximum packet size of default control endpoint and device qualifier descriptor etc. and creates a default configuration of device.

USB Function Stack Interface with Hardware Driver

Nucleus USB function stack interacts with USB function hardware driver through device manager. All USB function hardware drivers are registered with device manager with a USBFHW class GUID.

During initialization USB function stack looks and opens all devices registered with device manager using USBFHW class GUID.

USB function stack does not use read/write interfaces of device manager instead all communication is carried through IOCLs.

Nucleus USB Host Stack Initialization

Nucleus USB host stack initialization starts when system initialization calls the entry function of Nucleus USB host stack.

During initialization USB host stack looks for USB host hardware devices registered with device manager with USBHWW class GUID.

Upon successfully opening a USB host hardware device, USB function stack initializes the hardware and add to stacks hardware list by calling the [NU_USB_STACK_Add_Hw](#) function.

Nucleus USB Host Stack Interface with Hardware driver

Nucleus USB host stack interacts with USB host hardware driver through device manager. All USB host hardware drivers are registered with device manager with a USBHWW class GUID.

During initialization USB host stack looks and open all devices registered with device manager using USBHWW class GUID.

USB host stack does not use read/write interfaces of device manager instead all communication is carried through IOCLs.

Note



By default Nucleus USB stack is not enabled for Super Speed (USB 3.0) capability. In order to enable Super Speed capability in the Nucleus USB stack, set the default value of “ss_enable” option to “true” in the .metadata file of the Nucleus USB Stack Common Component located at: *nucleus/os/connectivity/usb/common/stack*.

Nucleus USB Function and Host IOCTLs

This section describes the following Nucleus USB Function and Host IOCTLs:

- `NU_USB_IOCTL_INITIALIZE`
- `NU_USB_IOCTL_IO_REQUEST`
- `NU_USB_IOCTL_OPEN_PIPE`
- `NU_USB_IOCTL_CLOSE_PIPE`
- `NU_USB_IOCTL_MODIFY_PIPE`
- `NU_USB_IOCTL_FLUSH_PIPE`
- `NU_USB_IOCTL_EXECUTE_ISR`
- `NU_USB_IOCTL_ENABLE_INT`
- `NU_USB_IOCTL_DISABLE_INT`
- `NU_USB_IOCTL_GET_ROLE`
- `NU_USB_IOCTL_START_SESSION`
- `NU_USB_IOCTL_END_SESSION`
- `NU_USB_IOCTL_NOTIF_ROLE_SWITCH`
- `NU_USB_IOCTL_GET_SPEED`
- `NU_USB_IOCTL_GET_HW_CB`
- `NU_USB_IOCTL_IS_CURR_AVAILABLE`
- `NU_USB_IOCTL_RELEASE_POWER`
- `NU_USB_IOCTL_REQ_POWER_DOWN`
- `NU_USB_IOCTL_OPEN_SS_PIPE`
- `NU_USB_IOCTL_MODIFY_SS_PIPE`
- `NU_USB_IOCTL_UPDATE_PWR_MODE`
- `NU_USB_IOCTL_UPDATE_BELT_VAL`
- `NU_USBF_IOCTL_GET_CAPABILITY`
- `NU_USBF_IOCTL_SET_ADDRESS`
- `NU_USBF_IOCTL_GET_DEV_STATUS`
- `NU_USBF_IOCTL_GET_ENDP_STATUS`

- `NU_USBF_IOCTL_STALL_ENDP`
- `NU_USBF_IOCTL_UNSTALL_ENDP`
- `NU_USBF_IOCTL_START_HNP`
- `NU_USBF_IOCTL_ACQUIRE_ENDP`
- `NU_USBF_IOCTL_RELEASE_ENDP`
- `NU_USBF_IOCTL_ENABLE_PULLUP`
- `NU_USBF_IOCTL_DISABLE_PULLUP`
- `NU_USBF_IOCTL_GET_EP0_MAXP`
- `NU_USBF_IOCTL_SEND_FUNCWAKENOTIF`
- `NU_USBF_IOCTL_SET_FEATURE_U0/U1_ENABLE`
- `NU_USBF_IOCTL_SET_LTM_ENABLE`
- `NU_USBF_IOCTL_CHECK_LTM_CAPABLE`
- `NU_USBF_IOCTL_GET_SUPPORTED_SPEEDS`
- `NU_USBF_IOCTL_GET_U1/U2DEVEXITLAT`
- `NU_USBH_IOCTL_UPDATE_DEVICE`
- `NU_USBH_IOCTL_INIT_DEVICE`
- `NU_USBH_IOCTL_DEINIT_DEVICE`
- `NU_USBH_IOCTL_UNSTALL_PIPE`
- `NU_USBH_IOCTL_DISABLE_PIPE`
- `NU_USBH_IOCTL_RESET_BANDWIDTH`

Related Topics

[USB Host and Functions](#)

NU_USB_IOCTL_INITIALIZE

This IOCTL initializes the USB hardware controller (either function or host) and makes it ready for communication on USB wire.

Called From

`NU_USB_HW_Initialize`

IOCTL Parameter

- Pointer to `NU_USB_STACK` control block

Include File

os/include/connectivity/nu_connectivity.h

Parameter

[IN] Pointer to USB stack control block

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USB_IOCTL_IO_REQUEST

Scope: USB stacks

This IOCTL is executed by USB stacks to send an IO Request Packet (IRP) to USB hardware driver.

Called From

NU_USB_HW_Submit_IRP

IOCTL Parameter

```
typedef struct _usb_irp_info
{
    NU_USB_IRP *irp;
    UINT8      endp_addr;
    UINT8      func_addr;
}USB_IRP_INFO;
```

Include File

os/include/connectivity/nu_connectivity.h

Parameter

[IN] IO Request Packet information

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USB_IOCTL_OPEN_PIPE

Opens a pipe for communication over USB.

Called From

NU_USB_HW_Open_Pipe

IOCTL Parameter

```
typedef struct _usb_ep_info
{
    UINT32 interval;
    UINT32 load;
    UINT16 max_packet_size;
    UINT16 endp_sts;
    UINT8  function_addr;
    UINT8  endp_addr;
    UINT8  endp_attrib;
    UINT8  speed;
    UINT8  config_num;
    UINT8  intf_num;
    UINT8  alt_sttg;

    #if ( CFG_NU_OS_CONN_USB_COM_STACK_SS_ENABLE == NU_TRUE )
        UINT8  max_burst;
        UINT8  ep_comp_attrib;
        UINT8  dummy_1;
        UINT8  dummy_2;
        UINT8  dummy_3;
        UINT16 bytes_per_interval;
        UINT16 dummy_4;
    #else
        UINT8  dummy_5;
    #endif
}USB_EP_INFO;
```

Include File

os/include/connectivity/nu_connectivity.h

Parameters

[IN] Information about the endpoint to be opened

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USB_IOCTL_CLOSE_PIPE

Closes an already opened pipe.

Called From

NU_USB_HW_Close_Pipe

IOCTL Parameter

```
typedef struct _usb_ep_info
{
    UINT32 interval;
    UINT32 load;
    UINT16 max_packet_size;
    UINT16 endp_sts;
    UINT8  function_addr;
    UINT8  endp_addr;
    UINT8  endp_attrib;
    UINT8  speed;
    UINT8  config_num;
    UINT8  intf_num;
    UINT8  alt_sttg;

    #if ( CFG_NU_OS_CONN_USB_COM_STACK_SS_ENABLE == NU_TRUE )
        UINT8  max_burst;
        UINT8  ep_comp_attrib;
        UINT8  dummy_1;
        UINT8  dummy_2;
        UINT8  dummy_3;
        UINT16 bytes_per_interval;
        UINT16 dummy_4;
    #else
        UINT8  dummy_5;
    #endif
}USB_EP_INFO;
```

Include File

os/include/connectivity/nu_connectivity.h

Parameters

[IN] Information about the endpoint to be closed

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USB_IOCTL_MODIFY_PIPE

Modifies an already opened pipe according to new parameters.

Called From

NU_USB_HW_Modify_Pipe

IOCTL Parameter

```
typedef struct _usb_ep_info
{
    UINT32 interval;
    UINT32 load;
    UINT16 max_packet_size;
    UINT16 endp_sts;
    UINT8  function_addr;
    UINT8  endp_addr;
    UINT8  endp_attrib;
    UINT8  speed;
    UINT8  config_num;
    UINT8  intf_num;
    UINT8  alt_sttg;

    #if ( CFG_NU_OS_CONN_USB_COM_STACK_SS_ENABLE == NU_TRUE )
        UINT8  max_burst;
        UINT8  ep_comp_attrib;
        UINT8  dummy_1;
        UINT8  dummy_2;
        UINT8  dummy_3;
        UINT16 bytes_per_interval;
        UINT16 dummy_4;
    #else
        UINT8  dummy_5;
    #endif
}USB_EP_INFO;
```

Include File

os/include/connectivity/nu_connectivity.h

Parameters

[IN] Information about the endpoint to be modified

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USB_IOCTL_FLUSH_PIPE

Flushes data from an open endpoint hardware FIFO and retire any submitted IRPs.

Called From

NU_USB_HW_Flush_Pipe

IOCTL Parameter

```
typedef struct _usb_ep_info
{
    UINT32 interval;
    UINT32 load;
    UINT16 max_packet_size;
    UINT16 endp_sts;
    UINT8  function_addr;
    UINT8  endp_addr;
    UINT8  endp_attrib;
    UINT8  speed;
    UINT8  config_num;
    UINT8  intf_num;
    UINT8  alt_sttg;

    #if ( CFG_NU_OS_CONN_USB_COM_STACK_SS_ENABLE == NU_TRUE )
        UINT8  max_burst;
        UINT8  ep_comp_attrib;
        UINT8  dummy_1;
        UINT8  dummy_2;
        UINT8  dummy_3;
        UINT16 bytes_per_interval;
        UINT16 dummy_4;
    #else
        UINT8  dummy_5;
    #endif
}USB_EP_INFO;
```

Include File

os/include/connectivity/nu_connectivity.h

Parameters

[IN] Information about the endpoint to be flushed

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USB_IOCTL_EXECUTE_ISR

Requests hardware driver to execute its ISR function to handle an interrupt.

Called From

NU_USB_HW_ISR

IOCTL Parameter

None

Include File

os/include/connectivity/nu_connectivity.h

Parameters

[IN] Don't care

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USB_IOCTL_ENABLE_INT

Enables hardware interrupt of USB hardware controller.

Called From

NU_USB_HW_Enable_Interrupts

IOCTL Parameter

None

Include File

os/include/connectivity/nu_connectivity.h

Parameters

[IN] Don't care

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USB_IOCTL_DISABLE_INT

Disables hardware interrupts of USB hardware controller.

Called From

NU_USB_HW_Disable_Interrupts

IOCTL Parameter

None

Include File

os/include/connectivity/nu_connectivity.h

Parameters

[IN] Don't care

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USB_IOCTL_GET_ROLE

Gets the current role of USB hardware (function or host). This is only required for dual role hardware devices.

Called From

NU_USB_HW_Get_Role

IOCTL Parameter

```
typedef struct _usb_port_info
{
    UINT8    port_id;
    UINT8    hw_role_out;
    UINT16   delay;
}USB_PORT_INFO;
```

Include File

os/include/connectivity/nu_connectivity.h

Parameters

[IN] Port information

[OUT] Current role of hardware (function or host).

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USB_IOCTL_START_SESSION

Requests the USB hardware to start session. This IOCTL is only required for dual role hardware devices.

Called From

NU_USB_HW_Start_Session

IOCTL Parameter

```
typedef struct _usb_port_info
{
    UINT8    port_id;
    UINT8    hw_role_out;
    UINT16   delay;
}USB_PORT_INFO;
```

Include File

os/include/connectivity/nu_connectivity.h

Parameters

[IN] Port information

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USB_IOCTL_END_SESSION

Requests the hardware to end the session. This IOCTL is only required for dual role hardware devices.

Called From

NU_USB_HW_End_Session

IOCTL Parameter

```
typedef struct _usb_port_info
{
    UINT8    port_id;
    UINT8    hw_role_out;
    UINT16   delay;
}USB_PORT_INFO;
```

Include File

os/include/connectivity/nu_connectivity.h

Parameters

[IN] Port information

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USB_IOCTL_NOTIF_ROLE_SWITCH

This IOCTL registers a role switch notification callback function.

Called From

`NU_USB_HW_Notify_Role_Switch`

IOCTL Parameter

Pointer to `NU_USB_HW_ROLESWITCH_CALLBACK`

Include File

os/include/connectivity/nu_connectivity.h

Parameters

[IN] Pointer to role switch notification callback function

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USB_IOCTL_GET_SPEED

This IOCTL gets the maximum speed at which this USB hardware can operate.

Called From

NU_USB_HW_Get_Speed

IOCTL Parameter

Pointer to UINT8

Include File

os/include/connectivity/nu_connectivity.h

Parameters

[IN] Don't care

[OUT] Maximum supported speed of USB hardware

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USB_IOCTL_GET_HW_CB

This IOCTL gets a pointer to USB hardware driver control block.

Called From

USBF_HW_Init_Task

IOCTL Parameter

Pointer to NU_USBF_HW control block

Include File

os/include/connectivity/nu_connectivity.h

Parameters

[IN] Don't care

[OUT] Pointer to NU_USBF_HW control block

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USB_IOCTL_IS_CURR_AVAILABLE

This IOCTL checks if the USB host controller has enough current available to support a newly connected device.

Called from

USB_Set_Config

IOCTL Parameters

```
typedef struct _usb_device_current_info
{
    NU_USB_DEVICE *device;
    UINT8         cfg;
    BOOLEAN        is_current_available;
}USB_DEV_CURRENT_INFO;
```

Include File

os/include/connectivity/nu_usb_hw_imp.h

Parameters

[IN] Pointer to USB_DEV_CURRENT_INFO containing the device control block and configuration number

[OUT] Boolean variable, set to NU_TRUE if current is available for the particular configuration otherwise NU_FALSE.

Description

If the newly connected device is self powered then this IOCTL always returns true otherwise it checks the configuration descriptor of the device to find the current requirement.

As the amount of current requirement of the device depends on the active configuration, this IOCTL is called every time when the configuration of a device is switched.

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USB_IOCTL_RELEASE_POWER

This IOCTL is used to release any power that may have been acquired by the device during the enumeration process.

Called from

USB_Unset_Config

IOCTL Parameters

```
typedef struct _usb_device_current_info
{
    NU_USB_DEVICE *device;
    UINT8         cfg;
    BOOLEAN       is_current_available;
}USB_DEV_CURRENT_INFO;
```

Include file

os/include/connectivity/nu_usb_hw_imp.h

Parameters

[IN] Pointer to USB_DEV_CURRENT_INFO containing the device control block

[OUT] Don't Care.

Description

Self-powered devices do not depend on power from the USB host controller, but the bus-powered devices get power from the USB host controller. As the amount of power depends on the active configuration, this IOCTL is called every time when the configuration of a device is switched.

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USB_IOCTL_REQ_POWER_DOWN

This IOCTL sets the USB hardware to low power mode.

Called from

NU_USB_HW_Request_Power_Down_Mode

IOCTL Parameters

None

Include File

os/include/connectivity/nu_usb_hw_imp.h

Parameters

[IN] Don't care

[OUT] Don't care

Description

Upon receiving this IOCTL, the USB hardware driver decides whether it can move to a lower power state or not. In the end, the decision about moving to a low power state is taken by the USB hardware driver.

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USB_IOCTL_OPEN_SS_PIPE

This IOCTL opens a USB super speed pipe for communication.

Called from

NU_USB_HW_Open_SS_Pipe

IOCTL Parameters

```
typedef struct _usb_ep_info
{
    UINT32 interval;
    UINT32 load;
    UINT16 max_packet_size;
    UINT16 endp_sts;
    UINT8  function_addr;
    UINT8  endp_addr;
    UINT8  endp_attrib;
    UINT8  speed;
    UINT8  config_num;
    UINT8  intf_num;
    UINT8  alt_sttg;

    #if ( CFG_NU_OS_CONN_USB_COM_STACK_SS_ENABLE == NU_TRUE )
        UINT8  max_burst;
        UINT8  ep_comp_attrib;
        UINT8  dummy_1;
        UINT8  dummy_2;
        UINT8  dummy_3;
        UINT16 bytes_per_interval;
        UINT16 dummy_4;
    #else
        UINT8  dummy_5;
    #endif
}USB_EP_INFO;
```

Include File

os/include/connectivity/nu_usb_hw_imp.h

Parameters

[IN] Information about the associated endpoint

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USB_IOCTL_MODIFY_SS_PIPE

This IOCTL modifies an already opened USB super speed pipe.

Called from

NU_USB_HW_Modify_SS_Pipe

IOCTL Parameters

```
typedef struct _usb_ep_info
{
    UINT32 interval;
    UINT32 load;
    UINT16 max_packet_size;
    UINT16 endp_sts;
    UINT8  function_addr;
    UINT8  endp_addr;
    UINT8  endp_attrib;
    UINT8  speed;
    UINT8  config_num;
    UINT8  intf_num;
    UINT8  alt_sttg;

    #if ( CFG_NU_OS_CONN_USB_COM_STACK_SS_ENABLE == NU_TRUE )
        UINT8  max_burst;
        UINT8  ep_comp_attrib;
        UINT8  dummy_1;
        UINT8  dummy_2;
        UINT8  dummy_3;
        UINT16 bytes_per_interval;
        UINT16 dummy_4;
    #else
        UINT8  dummy_5;
    #endif
}USB_EP_INFO;
```

Include File

os/include/connectivity/nu_usb_hw_imp.h

Parameters

[IN] Information about the associated endpoint

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USB_IOCTL_UPDATE_PWR_MODE

This IOCTL requests the USB hardware to scale down power according to the current link state.

Called from

`NU_USB_HW_Update_Power_Mode`

IOCTL Parameters

Pointer to UINT8 variable containing the requested power mode

Include File

os/include/connectivity/nu_usb_hw_imp.h

Parameters

[IN] Requested power mode

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USB_IOCTL_UPDATE_BELT_VAL

This IOCTL requests the USB hardware to update the BELT (Best Effort Latency Tolerance) value according to the current link state.

Called from

NU_USB_HW_Update_BELT_Value

IOCTL Parameters

Pointer to UINT16 containing the BELT value to be updated.

Include File

os/include/connectivity/nu_usb_hw_imp.h

Parameters

[IN] Best effort latency tolerance value to be set in hardware

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USBFI_IOCTL_GET_CAPABILITY

This IOCTL gets capability of USB function hardware regarding self handling of standard USB requests like SET_ADDRESS, SET_CONFIGURATION and so on.

Called From

NU_USBFI_HW_Get_Capability

IOCTL Parameter

Pointer to UINT32

Include File

os/include/connectivity/nu_connectivity.h

Parameters

[IN] Don't care

[OUT] Capability settings of USB function hardware

Parameters

[IN] Best effort latency tolerance value to be set in hardware

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USB_IOCTL_SET_ADDRESS

This IOCTL sets the current address of USB function hardware.

Called From

`NU_USB_HW_Set_Address`

IOCTL Parameter

Pointer to UINT8

Include File

os/include/connectivity/nu_connectivity.h

Parameters

[IN] Address of USB function hardware

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USBFI_IOCTL_GET_DEV_STATUS

This IOCTL gets the device status.

Called From

NU_USBFI_HW_Get_Status

IOCTL Parameter

Pointer to UINT16

Include File

os/include/connectivity/nu_connectivity.h

Parameters

[IN] Don't care

[OUT] Current devices status

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USBFI_IOCTL_GET_ENDP_STATUS

This IOCTL gets current status of endpoint.

Called From

NU_USBFI_HW_Get_Endpoint_Status

IOCTL Parameter

```
typedef struct _usb_ep_info
{
    UINT32 interval;
    UINT32 load;
    UINT16 max_packet_size;
    UINT16 endp_sts;
    UINT8  function_addr;
    UINT8  endp_addr;
    UINT8  endp_attrib;
    UINT8  speed;
    UINT8  config_num;
    UINT8  intf_num;
    UINT8  alt_sttg;

    #if ( CFG_NU_OS_CONN_USB_COM_STACK_SS_ENABLE == NU_TRUE )
        UINT8  max_burst;
        UINT8  ep_comp_attrib;
        UINT8  dummy_1;
        UINT8  dummy_2;
        UINT8  dummy_3;
        UINT16 bytes_per_interval;
        UINT16 dummy_4;
    #else
        UINT8  dummy_5;
    #endif
}USB_EP_INFO;
```

Include File

os/include/connectivity/nu_connectivity.h

Parameters

[IN] Don't care

[OUT] Current endpoint status

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USBFI_IOCTL_STALL_ENDP

This IOCTL stalls the specified endpoint.

Called From

NU_USBFI_HW_Stall_Endpoint

IOCTL Parameter

```
typedef struct _usb_ep_info
{
    UINT32 interval;
    UINT32 load;
    UINT16 max_packet_size;
    UINT16 endp_sts;
    UINT8  function_addr;
    UINT8  endp_addr;
    UINT8  endp_attrib;
    UINT8  speed;
    UINT8  config_num;
    UINT8  intf_num;
    UINT8  alt_sttg;

    #if ( CFG_NU_OS_CONN_USB_COM_STACK_SS_ENABLE == NU_TRUE )
        UINT8  max_burst;
        UINT8  ep_comp_attrib;
        UINT8  dummy_1;
        UINT8  dummy_2;
        UINT8  dummy_3;
        UINT16 bytes_per_interval;
        UINT16 dummy_4;
    #else
        UINT8  dummy_5;
    #endif
}USB_EP_INFO;
```

Include File

os/include/connectivity/nu_connectivity.h

Parameters

[IN] Information of the endpoint to be stalled

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USBFI_IOCTL_UNSTALL_ENDP

This IOCTL un-stalls an already stalled endpoint.

Called From

NU_USBFI_HW_Unstall_Endpoint

IOCTL Parameter

```
typedef struct _usb_ep_info
{
    UINT32 interval;
    UINT32 load;
    UINT16 max_packet_size;
    UINT16 endp_sts;
    UINT8  function_addr;
    UINT8  endp_addr;
    UINT8  endp_attrib;
    UINT8  speed;
    UINT8  config_num;
    UINT8  intf_num;
    UINT8  alt_sttg;

    #if ( CFG_NU_OS_CONN_USB_COM_STACK_SS_ENABLE == NU_TRUE )
        UINT8  max_burst;
        UINT8  ep_comp_attrib;
        UINT8  dummy_1;
        UINT8  dummy_2;
        UINT8  dummy_3;
        UINT16 bytes_per_interval;
        UINT16 dummy_4;
    #else
        UINT8  dummy_5;
    #endif
}USB_EP_INFO;
```

Include File

os/include/connectivity/nu_connectivity.h

Parameters

[IN] Information of the endpoint to be un-stalled.

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USBFI_IOCTL_START_HNP

This IOCTL requests the USB function hardware to start Host Negotiation Protocol (HNP) signaling on USB.

Called From

NU_USBFI_HW_Start_HNP

IOCTL Parameter

```
typedef struct _usb_port_info
{
    UINT8   port_id;
    UINT8   hw_role_out;
    UINT16  delay;
}USB_PORT_INFO;
```

Include File

os/include/connectivity/nu_connectivity.h

Parameters

[IN] Port information

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USBFI_IOCTL_ACQUIRE_ENDP

This IOCTL requests the USB function hardware to reserve an appropriate endpoint according to the specified endpoint information.

Called From

NU_USBFI_HW_Acquire_Endp

IOCTL Parameter

```
typedef struct _usb_ep_info
{
    UINT32 interval;
    UINT32 load;
    UINT16 max_packet_size;
    UINT16 endp_sts;
    UINT8  function_addr;
    UINT8  endp_addr;
    UINT8  endp_attrib;
    UINT8  speed;
    UINT8  config_num;
    UINT8  intf_num;
    UINT8  alt_sttg;

    #if ( CFG_NU_OS_CONN_USB_COM_STACK_SS_ENABLE == NU_TRUE )
        UINT8  max_burst;
        UINT8  ep_comp_attrib;
        UINT8  dummy_1;
        UINT8  dummy_2;
        UINT8  dummy_3;
        UINT16 bytes_per_interval;
        UINT16 dummy_4;
    #else
        UINT8  dummy_5;
    #endif
}USB_EP_INFO;
```

Include File

os/include/connectivity/nu_connectivity.h

Parameters

[IN] Endpoint information

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USBFI_IOCTL_RELEASE_ENDP

This IOCTL releases an already acquired endpoint.

Called From

NU_USBFI_HW_Release_Endp

IOCTL Parameter

```
typedef struct _usb_ep_info
{
    UINT32 interval;
    UINT32 load;
    UINT16 max_packet_size;
    UINT16 endp_sts;
    UINT8  function_addr;
    UINT8  endp_addr;
    UINT8  endp_attrib;
    UINT8  speed;
    UINT8  config_num;
    UINT8  intf_num;
    UINT8  alt_sttg;

    #if ( CFG_NU_OS_CONN_USB_COM_STACK_SS_ENABLE == NU_TRUE )
        UINT8  max_burst;
        UINT8  ep_comp_attrib;
        UINT8  dummy_1;
        UINT8  dummy_2;
        UINT8  dummy_3;
        UINT16 bytes_per_interval;
        UINT16 dummy_4;
    #else
        UINT8  dummy_5;
    #endif
}USB_EP_INFO;
```

Include File

os/include/connectivity/nu_connectivity.h

Parameters

[IN] Information about the endpoint

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USBFI_IOCTL_ENABLE_PULLUP

This IOCTL enables the pull-up on D+ line of USB.

Called From

NU_USBFI_HW_Enable_Pullup

IOCTL Parameter

None

Include File

os/include/connectivity/nu_connectivity.h

Parameters

[IN] Don't care

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USBFI_IOCTL_DISABLE_PULLUP

This IOCTL disables the pull-up on D+ line of USB.

Called From

NU_USBFI_HW_Disable_Pullup

IOCTL Parameter

None

Include File

os/include/connectivity/nu_connectivity.h

Parameters

[IN] Don't care

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USBFI_IOCTL_GET_EP0_MAXP

Scope: USB function stack

This IOCTL gets the maximum packet size of the default control endpoint for a particular speed. If the controller driver does not support this IOCTL, then the default maximum packet size of the control endpoint is 8 bytes, and for high speed it is 64 bytes.

Called from

NU_USBFI_HW_Get_EP0_Maxp

IOCTL Parameters

```
typedef struct _usb_ep_info
{
    UINT32 interval;
    UINT32 load;
    UINT16 max_packet_size;
    UINT16 endp_sts;
    UINT8  function_addr;
    UINT8  endp_addr;
    UINT8  endp_attrib;
    UINT8  speed;
    UINT8  config_num;
    UINT8  intf_num;
    UINT8  alt_sttg;

    #if ( CFG_NU_OS_CONN_USB_COM_STACK_SS_ENABLE == NU_TRUE )
        UINT8  max_burst;
        UINT8  ep_comp_attrib;
        UINT8  dummy_1;
        UINT8  dummy_2;
        UINT8  dummy_3;
        UINT16 bytes_per_interval;
        UINT16 dummy_4;
    #else
        UINT8  dummy_5;
    #endif
}USB_EP_INFO;
```

Include File

os/include/connectivity/nu_usbfi_hw_imp.h

Parameters

[IN] Required information for default control endpoint

[OUT] Maximum packet size of default control endpoint for specified speed

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USBFI_IOCTL_SEND_FUNCWAKENOTIF

This IOCTL sends a function wake-up notification to the USB host.

Called from

`NU_USBFI_HW_Send_FuncWakeNotif`

IOCTL Parameters

UINT8 Pointer to the variable containing the interface number of devices to be notified for wake-up.

Include File

os/include/connectivity/nu_usbfi_hw_imp.h

Parameters

[IN] Interface number of device function to be notified for wake-up.

[OUT] Don't care.

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USBF_IOCTL_SET_FEATURE_U0/U1_ENABLE

This IOCTL enables or disables the U0/U1 link entry.

Called from:

NU_USBF_HW_Set_Ux_Enable

IOCTL Parameters

Pointer to a BOOLEAN variable containing instructions for enabling/disabling the U0/U1 link entry.

Include File

os/include/connectivity/nu_usb_hw_imp.h

Parameters

[IN] enable/disable flags for U0/U1 entry.

[OUT] Don't care.

Description

The USB host can prevent a super speed device from entering the U0/U1 link state by sending a SET/CLEAR feature command. These IOCTLs are executed by the USB function stack upon receiving the SET/CLEAR feature command for U0/U1 link state transition.

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USBFI_IOCTL_SET_LTM_ENABLE

This IOCTL enables or disables Latency Tolerance Messages (LTM) generation.

Called from

NU_USBFI_HW_Set_LTM_Enable

IOCTL Parameters

Pointer to a BOOLEAN variable containing instructions for enabling/disabling LTM generation.

Include File

os/include/connectivity/nu_usbfi_hw_imp.h

Parameters

[IN] enable/disable flags for LTM generation.

[OUT] Don't care.

Description

The USB host can prevent a super speed device from generating Latency Tolerance Messages (LTM) by sending a SET/CLEAR LTM feature generation. These IOCTLs are executed by the USB function stack upon receiving a SET/CLEAR feature command for LTM generation.

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USBFI_IOCTL_CHECK_LTM_CAPABLE

Scope: USB function stack

This IOCTL checks if the USB hardware is capable of generating Latency Tolerance Messages (LTM) or not.

Called from

NU_USBFI_HW_Is_LTM_Capable

IOCTL Parameters

Pointer to the BOOLEAN variable containing LTM generation capability when the function returns.

Include File

os/include/connectivity/nu_usbfi_hw_imp.h

Parameters

[IN] Don't care.

[OUT] LTM generation capability.

NU_TRUE: Hardware is capable of generating LTM.

NU_FALSE: Hardware is not capable of generating LTM.

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USBF_IOCTL_GET_SUPPORTED_SPEEDS

This IOCTL gets the supported speeds of the USB super speed hardware.

Called from

`NU_USBF_HW_Get_Supported_Speeds`

IOCTL Parameters

Pointer to UINT16 containing a bitmap for supported speeds when the function returns.

Include File

`os/include/connectivity/nu_usbf_hw_imp.h`

Parameters

[IN] Don't care.

[OUT] Bitmap containing supported speeds by the USB hardware. The encoding of this bitmap is:

- b0: Device supports operations at low speed.
- b1: Device supports operations at full speed.
- b2: Device supports operations at high speed.
- b3: Device supports operations at super speed.
- b4-b15: Unused.

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USBFI_IOCTL_GET_U1/U2DEVEXITLAT

This IOCTL gets the worst case latency of the USB hardware to transition from state U1/U2 to state U0.

Called from

NU_USBFI_HW_Get_U1/U2DevExitLat

IOCTL Parameters

Pointer to UINT8 containing U1/U2 exit latency of the device when function returns.

Include File

os/include/connectivity/nu_usbfi_hw_imp.h

Parameters

[IN] Don't care.

[OUT] U1/U2 exit latency of the device.

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USBH_IOCTL_UPDATE_DEVICE

This IOCTL updates the parameters of the device as they are retrieved from the device descriptors.

Called from

```
USBH_Enumerate_Device  
USBH_HUB_Initialize_Intf
```

IOCTL Parameters

Pointer to USB_DEV_SS_PARAMS

```
typedef struct _usb_device_ss_parameters  
{  
    NU_USB_DEVICE *device;  
    UINT8          packet_size;  
    UINT32         sel;  
    BOOLEAN        is_hub;  
    UINT8          tt_time;  
    UINT8          num_ports;  
}USB_DEV_SS_PARAMS;
```

Include File

```
os/include/connectivity/nu_usbh_hw_imp.h
```

Parameters

[IN] Pointer to USB_DEV_SS_PARAMS structures.

[OUT] Don't care.

Description

The received parameters include System Exit Latency (SEL), Maximum Packet Size, TT time, number of ports and hub device.

The following functions are mapped on this IOCTL:

- NU_USBH_HW_Update_Max_Packet_Size
- NU_USBH_HW_Update_Hub_Device
- NU_USBH_HW_Update_SEL

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USBH_IOCTL_INIT_DEVICE

This IOCTL creates and initializes the data structures that are required by the host controller driver when the device is connected.

Called from

USB_HUB_Port_Status_Change

IOCTL Parameters

None

Include File

os/include/connectivity/nu_usbh_hw_imp.h

Parameters

[IN] Don't care.

[OUT] Don't care.

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USBH_IOCTL_DEINIT_DEVICE

This IOCTL releases the resources acquired by the device once it is disconnected.

Called from

NU_USBH_HW_Dinit_Device

IOCTL Parameters

Pointer to UINT8 containing the address of function device.

Include File

os/include/connectivity/nu_usbh_hw_imp.h

Parameters

[IN] USB Function address.

[OUT] Don't care

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USBH_IOCTL_UNSTALL_PIPE

This IOCTL removes a stall condition on an endpoint.

Called from

NU_USBH_HW_Unstall_Pipe

IOCTL Parameters

```
typedef struct _usb_ep_info
{
    UINT32 interval;
    UINT32 load;
    UINT16 max_packet_size;
    UINT16 endp_sts;
    UINT8  function_addr;
    UINT8  endp_addr;
    UINT8  endp_attrib;
    UINT8  speed;
    UINT8  config_num;
    UINT8  intf_num;
    UINT8  alt_sttg;

    #if ( CFG_NU_OS_CONN_USB_COM_STACK_SS_ENABLE == NU_TRUE )
        UINT8  max_burst;
        UINT8  ep_comp_attrib;
        UINT8  dummy_1;
        UINT8  dummy_2;
        UINT8  dummy_3;
        UINT16 bytes_per_interval;
        UINT16 dummy_4;
    #else
        UINT8  dummy_5;
    #endif
}USB_EP_INFO;
```

Include File

os/include/connectivity/nu_usbh_hw_imp.h

Parameters

[IN] Pointer to USB_EP_INFO control block.

[OUT] Don't care.

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USBH_IOCTL_DISABLE_PIPE

This IOCTL stops the endpoint from processing the IRP. After completion of this IOCTL the endpoint is in a stopped state.

Called from

NU_USBH_HW_Disable_Pipe

IOCTL Parameters

```
typedef struct _usb_ep_info
{
    UINT32 interval;
    UINT32 load;
    UINT16 max_packet_size;
    UINT16 endp_sts;
    UINT8  function_addr;
    UINT8  endp_addr;
    UINT8  endp_attrib;
    UINT8  speed;
    UINT8  config_num;
    UINT8  intf_num;
    UINT8  alt_sttg;

    #if ( CFG_NU_OS_CONN_USB_COM_STACK_SS_ENABLE == NU_TRUE )
        UINT8  max_burst;
        UINT8  ep_comp_attrib;
        UINT8  dummy_1;
        UINT8  dummy_2;
        UINT8  dummy_3;
        UINT16 bytes_per_interval;
        UINT16 dummy_4;
    #else
        UINT8  dummy_5;
    #endif
}USB_EP_INFO;
```

Include File

os/include/connectivity/nu_usbh_hw_imp.h

Parameters

[IN] Pointer to USB_EP_INFO control block.

[OUT] Don't care.

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

NU_USBH_IOCTL_RESET_BANDWIDTH

This IOCTL releases the BW acquired by the device in case of error.

Called from

NU_USBH_HW_Allocate_Bandwidth

IOCTL Parameters

Pointer to UINT8 variable containing the address of the USB function device.

Include File

os/include/connectivity/nu_usbh_hw_imp.h

Parameters

[IN] USB function address.

[OUT] Don't Care.

Related Topics

[Nucleus USB Function and Host IOCTLs](#)

Embedded Software and Hardware License Agreement

The latest version of the Embedded Software and Hardware License Agreement is available on-line at:
www.mentor.com/eshla

IMPORTANT INFORMATION

USE OF ALL PRODUCTS IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE PRODUCTS. USE OF PRODUCTS INDICATES CUSTOMER'S COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.

EMBEDDED SOFTWARE AND HARDWARE LICENSE AGREEMENT ("Agreement")

This is a legal agreement concerning the use of Products (as defined in Section 1) between the company acquiring the Products ("Customer"), and the Mentor Graphics entity that issued the corresponding quotation or, if no quotation was issued, the applicable local Mentor Graphics entity ("Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by Customer and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If Customer does not agree to these terms and conditions, promptly return or, in the case of Products received electronically, certify destruction of Products and all accompanying items within five days after receipt of such Products and receive a full refund of any license fee paid.

1. **Definitions.** As used in this Agreement and any applicable quotation, supplement, attachment and/or addendum ("Addenda"), these terms shall have the following meanings:
 - 1.1. "Customer's Product" means Customer's end-user product identified by a unique SKU (including any Related SKUs) in an applicable Addenda that is developed, manufactured, branded and shipped solely by Customer or an authorized manufacturer or subcontractor on behalf of Customer to end-users or consumers;
 - 1.2. "Developer" means a unique user, as identified by a unique user identification number, with access to Embedded Software at an authorized Development Location. A unique user is an individual who works directly with the embedded software in source code form, or creates, modifies or compiles software that ultimately links to the Embedded Software in Object Code form and is embedded into Customer's Product at the point of manufacture;
 - 1.3. "Development Location" means the location where Products may be used as authorized in the applicable Addenda;
 - 1.4. "Development Tools" means the software that may be used by Customer for building, editing, compiling, debugging or prototyping Customer's Product;
 - 1.5. "Embedded Software" means Software that is embeddable;
 - 1.6. "End-User" means Customer's customer;
 - 1.7. "Executable Code" means a compiled program translated into a machine-readable format that can be loaded into memory and run by a certain processor;
 - 1.8. "Hardware" means a physically tangible electro-mechanical system or sub-system and associated documentation;
 - 1.9. "Linkable Object Code" or "Object Code" means linkable code resulting from the translation, processing, or compiling of Source Code by a computer into machine-readable format;
 - 1.10. "Mentor Embedded Linux" or "MEL" means Mentor Graphics' tools, source code, and recipes for building Linux systems;
 - 1.11. "Open Source Software" or "OSS" means software subject to an open source license which requires as a condition for redistribution of such software, including modifications thereto, that the: (i) redistribution be in source code form or be made available in source code form; (ii) redistributed software be licensed to allow the making of derivative works; or (iii) redistribution be at no charge;
 - 1.12. "Processor" means the specific microprocessor to be used with Software and implemented in Customer's Product;
 - 1.13. "Products" means Software, Term-Licensed Products and/or Hardware;
 - 1.14. "Proprietary Components" means the components of the Products that are owned and/or licensed by Mentor Graphics and are not subject to an Open Source Software license, as more fully set forth in the product documentation provided with the Products;

- 1.15. “Redistributable Components” means those components that are intended to be incorporated or linked into Customer’s Linkable Object Code developed with the Software, as more fully set forth in the documentation provided with the Products;
- 1.16. “Related SKU” means two or more Customer Products identified by logically-related SKUs, where there is no difference or change in the electrical hardware or software content between such Customer Products;
- 1.17. “Software” means software programs, Embedded Software and/or Development Tools, including any updates, modifications, revisions, copies, documentation and design data that are licensed under this Agreement;
- 1.18. “Source Code” means software in a form in which the program logic is readily understandable by a human being;
- 1.19. “Sourcery CodeBench Software” means Mentor Graphics’ Development Tool for C/C++ embedded application development;
- 1.20. “Sourcery VSIPL++” is Software providing C++ classes and functions for writing embedded signal processing applications designed to run on one or more processors;
- 1.21. “Stock Keeping Unit” or “SKU” is a unique number or code used to identify each distinct product, item or service available for purchase;
- 1.22. “Subsidiary” means any corporation more than 50% owned by Customer, excluding Mentor Graphics competitors. Customer agrees to fulfill the obligations of such Subsidiary in the event of default. To the extent Mentor Graphics authorizes any Subsidiary’s use of Products under this Agreement, Customer agrees to ensure such Subsidiary’s compliance with the terms of this Agreement and will be liable for any breach by a Subsidiary; and
- 1.23. “Term-Licensed Products” means Products licensed to Customer for a limited time period (“Term”).

2. Orders, Fees and Payment.

- 2.1. To the extent Customer (or if agreed by Mentor Graphics, Customer’s appointed third party buying agent) places and Mentor Graphics accepts purchase orders pursuant to this Agreement (“Order(s)”), each Order will constitute a contract between Customer and Mentor Graphics, which shall be governed solely and exclusively by the terms and conditions of this Agreement and any applicable Addenda, whether or not these documents are referenced on the Order. Any additional or conflicting terms and conditions appearing on an Order will not be effective unless agreed in writing by an authorized representative of Customer and Mentor Graphics.
- 2.2. Amounts invoiced will be paid, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. All invoices will be sent electronically to Customer on the date stated on the invoice unless otherwise specified in an Addendum. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Prices do not include freight, insurance, customs duties, taxes or other similar charges, which Mentor Graphics will state separately in the applicable invoice(s). Unless timely provided with a valid certificate of exemption or other evidence that items are not taxable, Mentor Graphics will invoice Customer for all applicable taxes including, but not limited to, VAT, GST, sales tax, consumption tax and service tax. Customer will make all payments free and clear of, and without reduction for, any withholding or other taxes; any such taxes imposed on payments by Customer hereunder will be Customer’s sole responsibility. If Customer appoints a third party to place purchase orders and/or make payments on Customer’s behalf, Customer shall be liable for payment under Orders placed by such third party in the event of default.
- 2.3. All Products are delivered FCA factory (Incoterms 2010), freight prepaid and invoiced to Customer, except Software delivered electronically, which shall be deemed delivered when made available to Customer for download. Mentor Graphics’ delivery of Software by electronic means is subject to Customer’s provision of both a primary and an alternate e-mail address.

3. Grant of License.

- 3.1. The Products installed, downloaded, or otherwise acquired by Customer under this Agreement constitute or contain copyrighted, trade secret, proprietary and confidential information of Mentor Graphics or its licensors, who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Mentor Graphics grants to Customer, subject to payment of applicable license fees, a nontransferable, nonexclusive license to use Software as described in the applicable Addenda. The limited licenses granted under the applicable Addenda shall continue until the expiration date of Term-Licensed Products or termination in accordance with Section 12 below, whichever occurs first. **Mentor Graphics does NOT grant Customer any right to (a) sublicense or (b) use Software beyond the scope of this Section without first signing a separate agreement or Addenda with Mentor Graphics for such purpose.**
- 3.2. License Type. The license type shall be identified in the applicable Addenda.
- 3.2.1. Development License: During the Term, if any, Customer may modify, compile, assemble and convert the applicable Embedded Software Source Code into Linkable Object Code and/or Executable Code form by the number of Developers specified, for the Processor(s), Customer’s Product(s) and at the Development Location(s) identified in the applicable Addenda.

- 3.2.2. End-User Product License: During the Term, if any, and unless otherwise specified in the applicable Addenda, Customer may incorporate or embed an Executable Code version of the Embedded Software into the specified number of copies of Customer's Product(s), using the Processor Unit(s), and at the Development Location(s) identified in the applicable Addenda. Customer may manufacture, brand and distribute such Customer's Product(s) worldwide to its End-Users.
- 3.2.3. Internal Tool License: During the Term, if any, Customer may use the Development Tools solely: (a) for internal business purposes and (b) on the specified number of computer work stations and sites. Development Tools are licensed on a per-seat or floating basis, as specified in the applicable Addenda, and shall not be distributed to others or delivered in Customer's Product(s) unless specifically authorized in an applicable Addenda.
- 3.2.4. Sourcery CodeBench Professional Edition License: During the Term specified in the applicable Addenda, Customer may (a) install and use the Proprietary Components of the Software (i) if the license is a node-locked license, by a single user who uses the Software on up to two machines provided that only one copy of the Software is in use at any one time, or (ii) if the license is a floating license, by the authorized number of concurrent users on one or more machines provided that only the authorized number of copies of the Software are in use at any one time, and (b) distribute the Redistributable Components of the Software in Executable Code form only and only as part of Customer's Object Code developed with the Software that provides substantially different functionality than the Redistributable Component(s) alone.
- 3.2.5. Sourcery CodeBench Standard Edition License: During the Term specified in the applicable Addenda, Customer may (a) install and use the Proprietary Components of the Software by a single user who uses the Software on up to two machines provided that only one copy of the Software is in use at any one time, and (b) distribute the Redistributable Component(s) of the Software in Executable Code form only and only as part of Customer's Object Code developed with the Software that provides substantially different functionality than the Redistributable Component(s) alone.
- 3.2.6. Sourcery CodeBench Personal Edition License: During the Term specified in the applicable Addenda, Customer may (a) install and use the Proprietary Components of the Software by a single user who uses the Software on one machine, and (b) distribute the Redistributable Component(s) of the Software in Executable Code form only and only as part of Customer Object Code developed with the Software that provides substantially different functionality than the Redistributable Component(s) alone.
- 3.2.7. Sourcery CodeBench Academic Edition License: During the Term specified in the applicable Addenda, Customer may (a) install and use the Proprietary Components of the Software for non-commercial, academic purposes only by a single user who uses the Software on one machine, and (b) distribute the Redistributable Component(s) of the Software in Executable Code form only and only as part of Customer Object Code developed with the Software that provides substantially different functionality than the Redistributable Component(s) alone.
- 3.3. Mentor Graphics may from time to time, in its sole discretion, lend Products to Customer. For each loan, Mentor Graphics will identify in writing the quantity and description of Software loaned, the authorized location and the Term of the loan. Mentor Graphics will grant to Customer a temporary license to use the loaned Software solely for Customer's internal evaluation in a non-production environment. Customer shall return to Mentor Graphics or delete and destroy loaned Software on or before the expiration of the loan Term. Customer will sign a certification of such deletion or destruction if requested by Mentor Graphics.

4. Beta Code.

- 4.1. Portions or all of certain Products may contain code for experimental testing and evaluation ("Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to Customer a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. This grant and Customer's use of the Beta Code shall not be construed as marketing or offering to sell a license to the Beta Code, which Mentor Graphics may choose not to release commercially in any form.
- 4.2. If Mentor Graphics authorizes Customer to use the Beta Code, Customer agrees to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. Customer will contact Mentor Graphics periodically during Customer's use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of Customer's evaluation and testing, Customer will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.
- 4.3. Customer agrees to maintain Beta Code in confidence and shall restrict access to the Beta Code, including the methods and concepts utilized therein, solely to those employees and Customer location(s) authorized by Mentor Graphics to perform beta testing. Customer agrees that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on Customer's feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this Subsection 4.3 shall survive termination of this Agreement.

5. Restrictions on Use.

- 5.1. Customer may copy Software only as reasonably necessary to support the authorized use, including archival and backup purposes. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. Except where embedded in Executable Code form in Customer's Product, Customer shall maintain a record of the number and location of all copies of Software, including copies merged with other software and products, and shall make those records available to Mentor Graphics upon request. Customer shall not make Products available in any form to any person other than Customer's employees, authorized manufacturers or authorized contractors, excluding Mentor Graphics competitors, whose job performance requires access and who are under obligations of confidentiality. Customer shall take appropriate action to protect the confidentiality of Products and ensure that any person permitted access does not disclose or use Products except as permitted by this Agreement. Customer shall give Mentor Graphics immediate written notice of any unauthorized disclosure or use of the Products as soon as Customer learns or becomes aware of such unauthorized disclosure or use.
- 5.2. Customer acknowledges that the Products provided hereunder may contain Source Code which is proprietary and its confidentiality is of the highest importance and value to Mentor Graphics. Customer acknowledges that Mentor Graphics may be seriously harmed if such Source Code is disclosed in violation of this Agreement. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, Customer shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive any Source Code from Products that are not provided in Source Code form. Except as embedded in Executable Code in Customer's Product and distributed in the ordinary course of business, in no event shall Customer provide Products to Mentor Graphics competitors. Log files, data files, rule files and script files generated by or for the Software (collectively "Files") constitute and/or include confidential information of Mentor Graphics. Customer may share Files with third parties, excluding Mentor Graphics competitors, provided that the confidentiality of such Files is protected by written agreement at least as well as Customer protects other information of a similar nature or importance, but in any case with at least reasonable care. Under no circumstances shall Customer use Products or allow their use for the purpose of developing, enhancing or marketing any product that is in any way competitive with Products, or disclose to any third party the results of, or information pertaining to, any benchmark.
- 5.3. Customer may not assign this Agreement or the rights and duties under it, or relocate, sublicense or otherwise transfer the Products, whether by operation of law or otherwise ("Attempted Transfer"), without Mentor Graphics' prior written consent, which shall not be unreasonably withheld, and payment of Mentor Graphics' then-current applicable relocation and/or transfer fees. Any Attempted Transfer without Mentor Graphics' prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics' option, result in the immediate termination of the Agreement and/or the licenses granted under this Agreement. The terms of this Agreement, including without limitation the licensing and assignment provisions, shall be binding upon Customer's permitted successors in interest and assigns.
- 5.4. Notwithstanding any provision in an OSS license agreement applicable to a component of the Sourcery CodeBench Software that permits the redistribution of such component to a third party in Source Code or binary form, Customer may not use any Mentor Graphics trademark, whether registered or unregistered, in connection with such distribution, and may not recompile the Open Source Software components with the --with-pkgversion or --with-bugurl configuration options that embed Mentor Graphics' trademarks in the resulting binary.
- 5.5. The provisions of this Section 5 shall survive the termination of this Agreement.

6. Support Services.

- 6.1. Except as described in Sections 6.2, 6.3 and 6.4 below, and unless otherwise specified in any applicable Addenda to this Agreement, to the extent Customer purchases support services, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased in accordance with Mentor Graphics' then-current End-User Software Support Terms located at <http://supportnet.mentor.com/about/legal/>.
- 6.2. To the extent Customer purchases support services for Sourcery CodeBench Software, support will be provided solely in accordance with the provisions of this Section 6.2. Mentor Graphics shall provide updates and technical support to Customer as described herein only on the condition that Customer uses the Executable Code form of the Sourcery CodeBench Software for internal use only and/or distributes the Redistributable Components in Executable Code form only (except as provided in a separate redistribution agreement with Mentor Graphics or as required by the applicable Open Source license). Any other distribution by Customer of the Sourcery CodeBench Software (or any component thereof) in any form, including distribution permitted by the applicable Open Source license, shall automatically terminate any remaining support term. Subject to the foregoing and the payment of support fees, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased in accordance with Mentor Graphics' then-current Sourcery CodeBench Software Support Terms located at <http://www.mentor.com/codebench-support-legal>.
- 6.3. To the extent Customer purchases support services for Sourcery VSIPL++, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased solely in accordance with Mentor Graphics' then-current Sourcery VSIPL++ Support Terms located at <http://www.mentor.com/vsipl-support-legal>.
- 6.4. To the extent Customer purchases support services for Mentor Embedded Linux, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased

solely in accordance with Mentor Graphics' then-current Mentor Embedded Linux Support Terms located at <http://www.mentor.com/mel-support-legal>.

7. **Third Party and Open Source Software.** Products may contain Open Source Software or code distributed under a proprietary third party license agreement. Please see applicable Products documentation, including but not limited to license notice files, header files or source code for further details. Please see the applicable Open Source Software license(s) for additional rights and obligations governing your use and distribution of Open Source Software. Customer agrees that it shall not subject any Product provided by Mentor Graphics under this Agreement to any Open Source Software license that does not otherwise apply to such Product. In the event of conflict between the terms of this Agreement, any Addenda and an applicable OSS or proprietary third party agreement, the OSS or proprietary third party agreement will control solely with respect to the OSS or proprietary third party software component. The provisions of this Section 7 shall survive the termination of this Agreement.
8. **Limited Warranty.**
 - 8.1. Mentor Graphics warrants that during the warranty period its standard, generally supported Products, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual and/or specification. Mentor Graphics does not warrant that Products will meet Customer's requirements or that operation of Products will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. Customer must notify Mentor Graphics in writing of any nonconformity within the warranty period. For the avoidance of doubt, this warranty applies only to the initial shipment of Products under an Order and does not renew or reset, for example, with the delivery of (a) Software updates or (b) authorization codes. This warranty shall not be valid if Products have been subject to misuse, unauthorized modification or improper installation. MENTOR GRAPHICS' ENTIRE LIABILITY AND CUSTOMER'S EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF THE PRODUCTS TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF THE PRODUCTS THAT DO NOT MEET THIS LIMITED WARRANTY, PROVIDED CUSTOMER HAS OTHERWISE COMPLIED WITH THIS AGREEMENT. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; OR (B) PRODUCTS PROVIDED AT NO CHARGE, WHICH ARE PROVIDED "AS IS" UNLESS OTHERWISE AGREED IN WRITING.
 - 8.2. THE WARRANTIES SET FORTH IN THIS SECTION 8 ARE EXCLUSIVE TO CUSTOMER AND DO NOT APPLY TO ANY END-USER. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, WITH RESPECT TO PRODUCTS OR OTHER MATERIAL PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.
9. **LIMITATION OF LIABILITY.** EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, AND EXCEPT FOR EITHER PARTY'S BREACH OF ITS CONFIDENTIALITY OBLIGATIONS, CUSTOMER'S BREACH OF LICENSING TERMS OR CUSTOMER'S OBLIGATIONS UNDER SECTION 10, IN NO EVENT SHALL: (A) EITHER PARTY OR ITS RESPECTIVE LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF SUCH PARTY OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES; AND (B) EITHER PARTY OR ITS RESPECTIVE LICENSORS' LIABILITY UNDER THIS AGREEMENT, INCLUDING, FOR THE AVOIDANCE OF DOUBT, LIABILITY FOR ATTORNEYS' FEES OR COSTS, EXCEED THE GREATER OF THE FEES PAID OR OWING TO MENTOR GRAPHICS FOR THE PRODUCT OR SERVICE GIVING RISE TO THE CLAIM OR \$500,000 (FIVE HUNDRED THOUSAND U.S. DOLLARS). NOTWITHSTANDING THE FOREGOING, IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 9 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.
10. **Hazardous Applications.**
 - 10.1. Customer agrees that Mentor Graphics has no control over Customer's testing or the specific applications and use that Customer will make of Products. Mentor Graphics Products are not specifically designed for use in the operation of nuclear facilities, aircraft navigation or communications systems, air traffic control, life support systems, medical devices or other applications in which the failure of Mentor Graphics Products could lead to death, personal injury, or severe physical or environmental damage ("Hazardous Applications").
 - 10.2. CUSTOMER ACKNOWLEDGES IT IS SOLELY RESPONSIBLE FOR TESTING PRODUCTS USED IN HAZARDOUS APPLICATIONS AND SHALL BE SOLELY LIABLE FOR ANY DAMAGES RESULTING FROM SUCH USE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS SHALL BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF PRODUCTS IN ANY HAZARDOUS APPLICATIONS.
 - 10.3. CUSTOMER AGREES TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE OR LIABILITY, INCLUDING REASONABLE ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH THE USE OF PRODUCTS AS DESCRIBED IN SECTION 10.1.
 - 10.4. THE PROVISIONS OF THIS SECTION 10 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

11. Infringement.

- 11.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against Customer in the United States, Canada, Japan, or member state of the European Union which alleges that any standard, generally supported Product acquired by Customer hereunder infringes a patent or copyright or misappropriates a trade secret in such jurisdiction. Mentor Graphics will pay any costs and damages finally awarded against Customer that are attributable to the action. Customer understands and agrees that as conditions to Mentor Graphics' obligations under this section Customer must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.
 - 11.2. If a claim is made under Subsection 11.1 Mentor Graphics may, at its option and expense, and in addition to its obligations under Section 11.1, either (a) replace or modify the Product so that it becomes noninfringing; or (b) procure for Customer the right to continue using the Product. If Mentor Graphics determines that neither of those alternatives is financially practical or otherwise reasonably available, Mentor Graphics may require the return of the Product and refund to Customer any purchase price or license fee(s) paid.
 - 11.3. Mentor Graphics has no liability to Customer if the claim is based upon: (a) the combination of the Product with any product not furnished by Mentor Graphics, where the Product itself is not infringing; (b) the modification of the Product other than by Mentor Graphics or as directed by Mentor Graphics, where the unmodified Product would not infringe; (c) the use of the infringing Product when Mentor Graphics has provided Customer with a current unaltered release of a non-infringing Product of substantially similar functionality in accordance with Subsection 11.2(a); (d) the use of the Product as part of an infringing process; (e) a product that Customer makes, uses, or sells, where the Product itself is not infringing; (f) any Product provided at no charge; (g) any software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; (h) Open Source Software, except to the extent that the infringement is directly caused by Mentor Graphics' modifications to such Open Source Software; or (i) infringement by Customer that is deemed willful. In the case of (i), Customer shall reimburse Mentor Graphics for its reasonable attorneys' fees and other costs related to the action.
 - 11.4. THIS SECTION 11 IS SUBJECT TO SECTION 9 ABOVE AND STATES: (A) THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS AND (B) CUSTOMER'S SOLE AND EXCLUSIVE REMEDY, WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY PRODUCT PROVIDED UNDER THIS AGREEMENT.
12. **Termination and Effect of Termination.** If a Software license was provided for limited term use, such license will automatically terminate at the end of the authorized Term.
- 12.1. Termination for Breach. This Agreement shall remain in effect until terminated in accordance with its terms. Mentor Graphics may terminate this Agreement and/or any licenses granted under this Agreement, and Customer will immediately discontinue use and distribution of Products, if Customer (a) commits any material breach of any provision of this Agreement and fails to cure such breach upon 30-days prior written notice; or (b) becomes insolvent, files a bankruptcy petition, institutes proceedings for liquidation or winding up or enters into an agreement to assign its assets for the benefit of creditors. Termination of this Agreement or any license granted hereunder will not affect Customer's obligation to pay for Products shipped or licenses granted prior to the termination, which amounts shall be payable immediately upon the date of termination. For the avoidance of doubt, nothing in this Section 12 shall be construed to prevent Mentor Graphics from seeking immediate injunctive relief in the event of any threatened or actual breach of Customer's obligations hereunder.
 - 12.2. Effect of Termination. Upon termination of this Agreement, the rights and obligations of the parties shall cease except as expressly set forth in this Agreement. Upon termination or expiration of the Term, Customer will discontinue use and/or distribution of Products, and shall return Hardware and either return to Mentor Graphics or destroy Software in Customer's possession, including all copies and documentation, and certify in writing to Mentor Graphics within ten business days of the termination date that Customer no longer possesses any of the affected Products or copies of Software in any form, except to the extent an Open Source Software license conflicts with this Section 12.2 and permits Customer's continued use of any Open Source Software portion or component of a Product. Upon termination for Customer's breach, an End-User may continue its use and/or distribution of Customer's Product so long as: (a) the End-User was licensed according to the terms of this Agreement, if applicable to such End-User, and (b) such End-User is not in breach of its agreement, if applicable, nor a party to Customer's breach.
13. **Export.** The Products provided hereunder are subject to regulation by local laws and United States government agencies, which prohibit export or diversion of certain products, information about the products, and direct or indirect products thereof, to certain countries and certain persons. Customer agrees that it will not export Products in any manner without first obtaining all necessary approval from appropriate local and United States government agencies. Customer acknowledges that the regulation of product export is in continuous modification by local governments and/or the United States Congress and administrative agencies. Customer agrees to complete all documents and to meet all requirements arising out of such modifications.
14. **U.S. Government License Rights.** Software was developed entirely at private expense. All Software is commercial computer software within the meaning of the applicable acquisition regulations. Accordingly, pursuant to US FAR 48 CFR 12.212 and DFAR 48 CFR 227.7202, use, duplication and disclosure of the Software by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in this Agreement, except for provisions which are contrary to applicable mandatory federal laws.

15. **Third Party Beneficiary.** For any Products licensed under this Agreement and provided by Customer to End-Users, Mentor Graphics or the applicable licensor is a third party beneficiary of the agreement between Customer and End-User. Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, and other licensors may be third party beneficiaries of this Agreement with the right to enforce the obligations set forth herein.
16. **Review of License Usage.** Customer will monitor the access to and use of Software. With prior written notice, during Customer's normal business hours, and no more frequently than once per calendar year, Mentor Graphics may engage an internationally recognized accounting firm to review Customer's software monitoring system, records, accounts and sublicensing documents deemed relevant by the internationally recognized accounting firm to confirm Customer's compliance with the terms of this Agreement or U.S. or other local export laws. Such review may include FlexNet (or successor product) report log files that Customer shall capture and provide at Mentor Graphics' request. Customer shall make records available in electronic format and shall fully cooperate with data gathering to support the license review. Mentor Graphics shall bear the expense of any such review unless a material non-compliance is revealed. Mentor Graphics shall treat as confidential information all Customer information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement. Such license review shall be at Mentor Graphics' expense unless it reveals a material underpayment of fees of five percent or more, in which case Customer shall reimburse Mentor Graphics for the costs of such license review. Customer shall promptly pay any such fees. If the license review reveals that Customer has made an overpayment, Mentor Graphics has the option to either provide the Customer with a refund or credit the amount overpaid to Customer's next payment. The provisions of this Section 16 shall survive the termination of this Agreement.
17. **Controlling Law, Jurisdiction and Dispute Resolution.** This Agreement shall be governed by and construed under the laws of the State of California, USA, excluding choice of law rules. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of the state and federal courts of California, USA. Nothing in this section shall restrict Mentor Graphics' right to bring an action (including for example a motion for injunctive relief) against Customer or its Subsidiary in the jurisdiction where Customer's or its Subsidiary's place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.
18. **Severability.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.
19. **Miscellaneous.** This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements, including but not limited to any purchase order terms and conditions. This Agreement may only be modified in writing, signed by an authorized representative of each party. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.