



USB Library

Microchip Libraries for Applications (MLA)

Table of Contents

1 USB Library	15
1.1 Introduction	16
1.2 Legal Information	17
1.3 Release Notes	18
1.3.1 Revision History	18
1.3.1.1 v2.17	18
1.3.1.2 v2.16	18
1.3.1.3 v2.15	18
1.3.1.4 v2.14	19
1.3.1.5 v2.13	19
1.3.1.6 v2.12	20
1.3.1.7 v2.11	20
1.3.1.8 v2.10	20
1.3.1.9 v2.9j	20
1.3.1.10 v2.9i	21
1.3.1.11 v2.9h	21
1.3.1.12 v2.9g	22
1.3.1.13 v2.9f	22
1.3.1.14 v2.9e	23
1.3.1.15 v2.9d	23
1.3.1.16 v2.9c	24
1.3.1.17 v2.9b	24
1.3.1.18 v2.9a	25
1.3.1.19 v2.9	25
1.3.1.20 v2.8	26
1.3.1.21 v2.7a	27
1.3.1.22 v2.7	27
1.3.2 Support	29
1.3.3 Online Reference and Resources	29
1.3.4 Device (Slave) Demo Board Support and Limitations	30
1.3.5 Host Demo Board Support and Limitations	31
1.3.6 Operating System Support and Limitations	32
1.3.7 Tool Information	34
1.3.8 Library Migration	34
1.3.8.1 From v2.13 to v2.14+	34
1.3.8.2 From v2.12 to v2.13	34
1.3.8.3 From v2.11 to v2.12	35

1.3.8.4 From v2.10 to v2.11	35
1.3.8.5 From v2.9j to v2.10	35
1.3.8.6 From v2.9i to v2.9j	36
1.3.8.7 From v2.9h to v2.9i	36
1.3.8.8 From v2.9g to v2.9h	36
1.3.8.9 From v2.9f to v2.9g	36
1.3.8.10 From v2.9e to v2.9f	36
1.3.8.11 From v2.9d to v2.9e	37
1.3.8.12 From v2.9c to v2.9d	37
1.3.8.13 From v2.9b to v2.9c	37
1.3.8.14 From v2.9a to v2.9b	37
1.3.8.15 From v2.9 to v2.9a	37
1.3.8.16 From v2.8 to v2.9	37
1.3.8.17 From v2.7a to v2.8	37
1.3.8.18 From v2.7 to v2.7a	37
1.3.8.19 From v2.6a to v2.7	38
1.3.8.20 From v2.6 to v2.6a	38
1.3.8.21 From v2.5 to v2.6	38
1.4 Library Interface	40
1.4.1 Device/Peripheral	40
1.4.1.1 Device Stack	40
1.4.1.1.1 Functions	40
1.4.1.1.1.1 USB_APPLICATION_EVENT_HANDLER Function	43
1.4.1.1.1.2 USBCancelIO Function	44
1.4.1.1.1.3 USBCtrlEPAllowDataStage Function	44
1.4.1.1.1.4 USBCtrlEPAllowStatusStage Function	44
1.4.1.1.1.5 USBDeferINDataStage Function	45
1.4.1.1.1.6 USBDeferOUTDataStage Function	46
1.4.1.1.1.7 USBDeferStatusStage Function	47
1.4.1.1.1.8 USBDeviceAttach Function	48
1.4.1.1.1.9 USBDeviceDetach Function	48
1.4.1.1.1.10 USBDeviceInit Function	50
1.4.1.1.1.11 USBDeviceTasks Function	50
1.4.1.1.1.12 USBEnableEndpoint Function	51
1.4.1.1.1.13 USBEP0Receive Function	52
1.4.1.1.1.14 USBEP0SendRAMPtr Function	53
1.4.1.1.1.15 USBEP0SendROMPtr Function	53
1.4.1.1.1.16 USBEP0Transmit Function	54
1.4.1.1.1.17 USBGet1msTickCount Function	54
1.4.1.1.1.18 USBGetTicksSinceSuspendEnd Function	55
1.4.1.1.1.19 USBGetDeviceState Function	56
1.4.1.1.1.20 USBGetNextHandle Function	57

1.4.1.1.1.21 USBGetRemoteWakeupStatus Function	58
1.4.1.1.1.22 USBGetSuspendState Function	59
1.4.1.1.1.23 USBHandleBusy Function	60
1.4.1.1.1.24 USBHandleGetAddr Function	61
1.4.1.1.1.25 USBHandleGetLength Function	61
1.4.1.1.1.26 USBIncrement1msInternalTimers Function	62
1.4.1.1.1.27 USBIIDataStageDeferred Function	62
1.4.1.1.1.28 USBIsBusSuspended Function	63
1.4.1.1.1.29 USBIsDeviceSuspended Function	63
1.4.1.1.1.30 USBOUTDataStageDeferred Function	64
1.4.1.1.1.31 USBRxOnePacket Function	65
1.4.1.1.1.32 USBSoftDetach Function	65
1.4.1.1.1.33 USBStallEndpoint Function	66
1.4.1.1.1.34 USBTransferOnePacket Function	66
1.4.1.1.1.35 USBTxOnePacket Function	67
1.4.1.1.2 Data Types and Constants	68
1.4.1.1.2.1 USB_DEVICE_STACK_EVENTS Type	69
1.4.1.1.2.2 USB_DEVICE_STATE Type	69
1.4.1.1.2.3 DESC_CONFIG_uint32_t Macro	69
1.4.1.1.2.4 DESC_CONFIG_uint8_t Macro	69
1.4.1.1.2.5 DESC_CONFIG_WORD Macro	70
1.4.1.1.2.6 USB_EP0_BUSY Macro	70
1.4.1.1.2.7 USB_EP0_INCLUDE_ZERO Macro	70
1.4.1.1.2.8 USB_EP0_NO_DATA Macro	70
1.4.1.1.2.9 USB_EP0_NO_OPTIONS Macro	71
1.4.1.1.2.10 USB_EP0_RAM Macro	71
1.4.1.1.2.11 USB_EP0_ROM Macro	71
1.4.1.1.2.12 USB_HANDLE Macro	71
1.4.1.1.3 _USB_DEVICE_H Macro	71
1.4.1.1.4 usb_device.h	72
1.4.1.2 Audio Function Driver	75
1.4.1.2.1 Functions	75
1.4.1.2.1.1 USBCheckAudioRequest Function	76
1.4.1.2.2 usb_device_audio.h	76
1.4.1.3 CDC Function Driver	76
1.4.1.3.1 usb_device_cdc.h	77
1.4.1.3.2 Functions	78
1.4.1.3.2.1 CDCInitEP Function	80
1.4.1.3.2.2 CDCNotificationHandler Function	80
1.4.1.3.2.3 CDCTxService Function	81
1.4.1.3.2.4 getsUSBUSART Function	82
1.4.1.3.2.5 putsUSBUSART Function	82

1.4.1.3.2.6 putsUSBUSART Function	83
1.4.1.3.2.7 putUSBUSART Function	84
1.4.1.3.2.8 USBCDCEventHandler Function	84
1.4.1.3.2.9 USBCheckCDCRequest Function	85
1.4.1.3.2.10 CDCSetBaudRate Macro	85
1.4.1.3.2.11 CDCSetCharacterFormat Macro	86
1.4.1.3.2.12 CDCSetDataSize Macro	86
1.4.1.3.2.13 CDCSetLineCoding Macro	87
1.4.1.3.2.14 CDCSetParity Macro	87
1.4.1.3.2.15 mUSBUSARTIsTxTrfReady Macro	88
1.4.1.3.2.16 mUSBUSARTTxRam Macro	88
1.4.1.3.2.17 mUSBUSARTTxRom Macro	89
1.4.1.3.2.18 USBUSARTIsTxTrfReady Macro	89
1.4.1.3.3 CDC_H Macro	90
1.4.1.3.4 Data Types and Constants	90
1.4.1.3.4.1 NUM_STOP_BITS_1 Macro	91
1.4.1.3.4.2 NUM_STOP_BITS_1_5 Macro	91
1.4.1.3.4.3 NUM_STOP_BITS_2 Macro	91
1.4.1.3.4.4 PARITY_EVEN Macro	91
1.4.1.3.4.5 PARITY_MARK Macro	91
1.4.1.3.4.6 PARITY_NONE Macro	92
1.4.1.3.4.7 PARITY_ODD Macro	92
1.4.1.3.4.8 PARITY_SPACE Macro	92
1.4.1.4 HID Function Driver	92
1.4.1.4.1 Functions	92
1.4.1.4.1.1 HIDRxHandleBusy Macro	93
1.4.1.4.1.2 HIDRxPacket Macro	93
1.4.1.4.1.3 HIDTxHandleBusy Macro	94
1.4.1.4.1.4 HIDTxPacket Macro	95
1.4.1.4.2 Data Types and Constants	95
1.4.1.4.2.1 BOOT_INTF_SUBCLASS Macro	96
1.4.1.4.2.2 BOOT_PROTOCOL Macro	96
1.4.1.4.2.3 HID_PROTOCOL_KEYBOARD Macro	96
1.4.1.4.2.4 HID_PROTOCOL_MOUSE Macro	96
1.4.1.4.2.5 HID_PROTOCOL_NONE Macro	97
1.4.1.4.3 usb_device_hid.h	97
1.4.1.5 MSD Function Driver	97
1.4.1.5.1 Functions	97
1.4.1.5.1.1 MSDTasks Function	98
1.4.1.5.1.2 MSDTransferTerminated Function	98
1.4.1.5.1.3 USBCheckMSDRequest Function	98
1.4.1.5.1.4 USBMSDInit Function	99

1.4.1.5.2 Data Types and Constants	99
1.4.1.5.2.1 LUN_FUNCTIONS Structure	99
1.4.1.5.3 usb_device_msd.h	100
1.4.1.6 Vendor Class (Generic) Function Driver	101
1.4.1.6.1 Functions	101
1.4.1.6.1.1 USBCheckVendorRequest Function	101
1.4.1.6.1.2 USBGEN_H Macro	102
1.4.1.6.1.3 USBGenRead Macro	102
1.4.1.6.1.4 USBGenWrite Macro	103
1.4.1.6.2 usb_device_generic.h	103
1.4.2 Embedded Host API	104
1.4.2.1 Embedded Host Stack	104
1.4.2.1.1 Functions	104
1.4.2.1.1.1 USB_HOST_APP_DATA_EVENT_HANDLER Function	106
1.4.2.1.1.2 USB_HOST_APP_EVENT_HANDLER Function	106
1.4.2.1.1.3 USB_HostInterruptHandler Function	107
1.4.2.1.1.4 USBHostClearEndpointErrors Function	107
1.4.2.1.1.5 USBHostDeviceSpecificClientDriver Function	108
1.4.2.1.1.6 USBHostDeviceStatus Function	108
1.4.2.1.1.7 USBHostInit Function	109
1.4.2.1.1.8 USBHostIsochronousBuffersCreate Function	110
1.4.2.1.1.9 USBHostIsochronousBuffersDestroy Function	110
1.4.2.1.1.10 USBHostIsochronousBuffersReset Function	111
1.4.2.1.1.11 USBHostIssueDeviceRequest Function	111
1.4.2.1.1.12 USBHostRead Function	112
1.4.2.1.1.13 USBHostResetDevice Function	113
1.4.2.1.1.14 USBHostResumeDevice Function	113
1.4.2.1.1.15 USBHostSetDeviceConfiguration Function	114
1.4.2.1.1.16 USBHostSetNAKTimeout Function	115
1.4.2.1.1.17 USBHostShutdown Function	115
1.4.2.1.1.18 USBHostSuspendDevice Function	116
1.4.2.1.1.19 USBHostTasks Function	116
1.4.2.1.1.20 USBHostTerminateTransfer Function	117
1.4.2.1.1.21 USBHostTransferIsComplete Function	117
1.4.2.1.1.22 USBHostVbusEvent Function	118
1.4.2.1.1.23 USBHostWrite Function	119
1.4.2.1.1.24 USBHostGetCurrentConfigurationDescriptor Macro	119
1.4.2.1.1.25 USBHostGetDeviceDescriptor Macro	120
1.4.2.1.1.26 USBHostGetStringDescriptor Macro	120
1.4.2.1.1.27 USBHostReadIsochronous Macro	122
1.4.2.1.1.28 USBHostWriteIsochronous Macro	122
1.4.2.1.2 Data Types and Constants	123

1.4.2.1.2.1 CLIENT_DRIVER_TABLE Structure	124
1.4.2.1.2.2 HOST_TRANSFER_DATA Structure	125
1.4.2.1.2.3 TRANSFER_ATTRIBUTES Type	125
1.4.2.1.2.4 USB_CLIENT_EVENT_HANDLER Type	126
1.4.2.1.2.5 USB_CLIENT_INIT Type	126
1.4.2.1.2.6 USB_TPL Type	127
1.4.2.1.2.7 INIT_CL_SC_P Macro	127
1.4.2.1.2.8 INIT_VID_PID Macro	127
1.4.2.1.2.9 TPL_ALLOW_HNP Macro	127
1.4.2.1.2.10 TPL_CLASS_DRV Macro	128
1.4.2.1.2.11 TPL_EP0_ONLY_CUSTOM_DRIVER Macro	128
1.4.2.1.2.12 TPL_IGNORE_CLASS Macro	128
1.4.2.1.2.13 TPL_IGNORE_PID Macro	128
1.4.2.1.2.14 TPL_IGNORE_PROTOCOL Macro	128
1.4.2.1.2.15 TPL_IGNORE_SUBCLASS Macro	129
1.4.2.1.2.16 TPL_SET_CONFIG Macro	129
1.4.2.1.2.17 USB_HOST_APP_DATA_EVENT_HANDLER Macro	129
1.4.2.1.2.18 USB_HOST_APP_EVENT_HANDLER Macro	129
1.4.2.1.2.19 USB_NUM_BULK_NAKS Macro	129
1.4.2.1.2.20 USB_NUM_COMMAND_TRIES Macro	130
1.4.2.1.2.21 USB_NUM_CONTROL_NAKS Macro	130
1.4.2.1.2.22 USB_NUM_ENUMERATION_TRIES Macro	130
1.4.2.1.2.23 USB_NUM_INTERRUPT_NAKS Macro	130
1.4.2.1.3 usb_host.h	130
1.4.2.1.4 __USBHOST_H__ Macro	133
1.4.2.2 CDC Client Driver	133
1.4.2.2.1 Functions	135
1.4.2.2.1.1 USBHostCDC_Api_ACm_Request Function	136
1.4.2.2.1.2 USBHostCDC_Api_Get_IN_Data Function	136
1.4.2.2.1.3 USBHostCDC_Api_Send_OUT_Data Function	137
1.4.2.2.1.4 USBHostCDC_ApiDeviceDetect Function	137
1.4.2.2.1.5 USBHostCDC_ApiTransferIsComplete Function	137
1.4.2.2.1.6 USBHostCDCDeviceStatus Function	138
1.4.2.2.1.7 USBHostCDCEventHandler Function	138
1.4.2.2.1.8 USBHostCDCInitAddress Function	139
1.4.2.2.1.9 USBHostCDCInitialize Function	139
1.4.2.2.1.10 USBHostCDCResetDevice Function	140
1.4.2.2.1.11 USBHostCDCTasks Function	140
1.4.2.2.1.12 USBHostCDCTransfer Function	141
1.4.2.2.1.13 USBHostCDCTransferIsComplete Function	142
1.4.2.2.2 Data Types and Constants	142
1.4.2.2.2.1 COMM_INTERFACE_DETAILS Structure	144

1.4.2.2.2.2 DATA_INTERFACE_DETAILS Structure	145
1.4.2.2.2.3 USB_CDC_ACM_FN_DSC Structure	146
1.4.2.2.2.4 USB_CDC_CALL_MGT_FN_DSC Structure	146
1.4.2.2.2.5 USB_CDC_CONTROL_SIGNAL_BITMAP Union	146
1.4.2.2.2.6 USB_CDC_DEVICE_INFO Structure	147
1.4.2.2.2.7 USB_CDC_HEADER_FN_DSC Structure	148
1.4.2.2.2.8 USB_CDC_LINE_CODING Union	148
1.4.2.2.2.9 USB_CDC_UNION_FN_DSC Structure	149
1.4.2.2.2.10 DEVICE_CLASS_CDC Macro	149
1.4.2.2.2.11 EVENT_CDC_ATTACH Macro	149
1.4.2.2.2.12 EVENT_CDC_COMM_READ_DONE Macro	150
1.4.2.2.2.13 EVENT_CDC_COMM_WRITE_DONE Macro	150
1.4.2.2.2.14 EVENT_CDC_DATA_READ_DONE Macro	150
1.4.2.2.2.15 EVENT_CDC_DATA_WRITE_DONE Macro	150
1.4.2.2.2.16 EVENT_CDC_NAK_TIMEOUT Macro	151
1.4.2.2.2.17 EVENT_CDC_NONE Macro	151
1.4.2.2.2.18 EVENT_CDC_OFFSET Macro	151
1.4.2.2.2.19 EVENT_CDC_RESET Macro	151
1.4.2.2.2.20 USB_CDC_ABSTRACT_CONTROL_MODEL Macro	151
1.4.2.2.2.21 USB_CDC_ATM_NETWORKING_CONTROL_MODEL Macro	152
1.4.2.2.2.22 USB_CDC_CAPI_CONTROL_MODEL Macro	152
1.4.2.2.2.23 USB_CDC_CLASS_ERROR Macro	152
1.4.2.2.2.24 USB_CDC_COMM_INTF Macro	152
1.4.2.2.2.25 USB_CDC_COMMAND_FAILED Macro	152
1.4.2.2.2.26 USB_CDC_COMMAND_PASSED Macro	153
1.4.2.2.2.27 USB_CDC_CONTROL_LINE_LENGTH Macro	153
1.4.2.2.2.28 USB_CDC_CS_ENDPOINT Macro	153
1.4.2.2.2.29 USB_CDC_CS_INTERFACE Macro	153
1.4.2.2.2.30 USB_CDC_DATA_INTF Macro	153
1.4.2.2.2.31 USB_CDC_DEVICE_BUSY Macro	154
1.4.2.2.2.32 USB_CDC_DEVICE_DETACHED Macro	154
1.4.2.2.2.33 USB_CDC_DEVICE_HOLDING Macro	154
1.4.2.2.2.34 USB_CDC_DEVICE_MANAGEMENT Macro	154
1.4.2.2.2.35 USB_CDC_DEVICE_NOT_FOUND Macro	154
1.4.2.2.2.36 USB_CDC_DIRECT_LINE_CONTROL_MODEL Macro	155
1.4.2.2.2.37 USB_CDC_DSC_FN_ACM Macro	155
1.4.2.2.2.38 USB_CDC_DSC_FN_CALL_MGT Macro	155
1.4.2.2.2.39 USB_CDC_DSC_FN_COUNTRY_SELECTION Macro	155
1.4.2.2.2.40 USB_CDC_DSC_FN_DLM Macro	155
1.4.2.2.2.41 USB_CDC_DSC_FN_HEADER Macro	156
1.4.2.2.2.42 USB_CDC_DSC_FN_RPT_CAPABILITIES Macro	156
1.4.2.2.2.43 USB_CDC_DSC_FN_TEL_OP_MODES Macro	156

1.4.2.2.2.44 USB_CDC_DSC_FN_TELEPHONE_RINGER Macro	156
1.4.2.2.2.45 USB_CDC_DSC_FN_UNION Macro	156
1.4.2.2.2.46 USB_CDC_DSC_FN_USB_TERMINAL Macro	157
1.4.2.2.2.47 USB_CDC_ETHERNET_EMULATION_MODEL Macro	157
1.4.2.2.2.48 USB_CDC_ETHERNET_NETWORKING_CONTROL_MODEL Macro	157
1.4.2.2.2.49 USB_CDC_GET_COMM_FEATURE Macro	157
1.4.2.2.2.50 USB_CDC_GET_ENCAPSULATED_REQUEST Macro	157
1.4.2.2.2.51 USB_CDC_GET_LINE_CODING Macro	158
1.4.2.2.2.52 USB_CDC_ILLEGAL_REQUEST Macro	158
1.4.2.2.2.53 USB_CDC_INITIALIZING Macro	158
1.4.2.2.2.54 USB_CDC_INTERFACE_ERROR Macro	158
1.4.2.2.2.55 USB_CDC_LINE_CODING_LENGTH Macro	158
1.4.2.2.2.56 USB_CDC_MAX_PACKET_SIZE Macro	159
1.4.2.2.2.57 USB_CDC_MOBILE_DIRECT_LINE_MODEL Macro	159
1.4.2.2.2.58 USB_CDC_MULTI_CHANNEL_CONTROL_MODEL Macro	159
1.4.2.2.2.59 USB_CDC_NO_PROTOCOL Macro	159
1.4.2.2.2.60 USB_CDC_NO_REPORT_DESCRIPTOR Macro	159
1.4.2.2.2.61 USB_CDC_NORMAL_RUNNING Macro	160
1.4.2.2.2.62 USB_CDC_OBEX Macro	160
1.4.2.2.2.63 USB_CDC_PHASE_ERROR Macro	160
1.4.2.2.2.64 USB_CDC_REPORT_DESCRIPTOR_BAD Macro	160
1.4.2.2.2.65 USB_CDC_RESET_ERROR Macro	161
1.4.2.2.2.66 USB_CDC_RESETTING_DEVICE Macro	161
1.4.2.2.2.67 USB_CDC_SEND_BREAK Macro	161
1.4.2.2.2.68 USB_CDC_SEND_ENCAPSULATED_COMMAND Macro	161
1.4.2.2.2.69 USB_CDC_SET_COMM_FEATURE Macro	161
1.4.2.2.2.70 USB_CDC_SET_CONTROL_LINE_STATE Macro	162
1.4.2.2.2.71 USB_CDC_SET_LINE_CODING Macro	162
1.4.2.2.2.72 USB_CDC_TELEPHONE_CONTROL_MODEL Macro	162
1.4.2.2.2.73 USB_CDC_V25TER Macro	162
1.4.2.2.2.74 USB_CDC_WIRELESS_HANDSET_CONTROL_MODEL Macro	162
1.4.2.2.3 usb_host_cdc.h	163
1.4.2.2.4 usb_host_cdc_interface.h	165
1.4.2.2.5 _USB_HOST_CDC_H Macro	166
1.4.2.2.6 _USB_HOST_CDC_INTERFACE_H Macro	166
1.4.2.3 HID Client Driver	166
1.4.2.3.1 Functions	167
1.4.2.3.1.1 USBHostHID_ApiFindBit Function	168
1.4.2.3.1.2 USBHostHID_ApiFindValue Function	168
1.4.2.3.1.3 USBHostHID_ApiGetCurrentInterfaceNum Function	169
1.4.2.3.1.4 USBHostHID_ApilImportData Function	169
1.4.2.3.1.5 USBHostHID_HasUsage Function	170

1.4.2.3.1.6 USBHostHID_GetCurrentReportInfo Macro	170
1.4.2.3.1.7 USBHostHID_GetItemListPointers Macro	171
1.4.2.3.1.8 USBHostHIDDeviceDetect Function	171
1.4.2.3.1.9 USBHostHIDDeviceStatus Function	172
1.4.2.3.1.10 USBHostHIDEEventHandler Function	172
1.4.2.3.1.11 USBHostHIDInitialize Function	173
1.4.2.3.1.12 USBHostHIDRead Function	173
1.4.2.3.1.13 USBHostHIDReadIsComplete Function	174
1.4.2.3.1.14 USBHostHIDReadTerminate Function	174
1.4.2.3.1.15 USBHostHIDResetDevice Function	174
1.4.2.3.1.16 USBHostHIDTasks Function	175
1.4.2.3.1.17 USBHostHIDWrite Function	176
1.4.2.3.1.18 USBHostHIDWriteIsComplete Function	176
1.4.2.3.1.19 USBHostHIDWriteTerminate Function	177
1.4.2.3.2 Data Types and Constants	177
1.4.2.3.2.1 HID_COLLECTION Structure	180
1.4.2.3.2.2 HID_DATA_DETAILS Structure	180
1.4.2.3.2.3 HID_DESIGITEM Structure	181
1.4.2.3.2.4 HID_GLOBALS Structure	181
1.4.2.3.2.5 HID_ITEM_INFO Structure	182
1.4.2.3.2.6 HID_REPORT Structure	183
1.4.2.3.2.7 HID_REPORTITEM Structure	183
1.4.2.3.2.8 HID_STRINGITEM Structure	184
1.4.2.3.2.9 HID_TRANSFER_DATA Structure	184
1.4.2.3.2.10 HID_USAGEITEM Structure	185
1.4.2.3.2.11 HID_USER_DATA_SIZE Type	185
1.4.2.3.2.12 HIDReportTypeEnum Enumeration	185
1.4.2.3.2.13 USB_HID_DEVICE_ID Structure	186
1.4.2.3.2.14 USB_HID_DEVICE_RPT_INFO Structure	186
1.4.2.3.2.15 USB_HID_ITEM_LIST Structure	188
1.4.2.3.2.16 USB_HID_RPT_DESC_ERROR Enumeration	188
1.4.2.3.2.17 deviceRptInfo Variable	189
1.4.2.3.2.18 itemListPtrs Variable	190
1.4.2.3.2.19 DEVICE_CLASS_HID Macro	190
1.4.2.3.2.20 DSC_HID Macro	190
1.4.2.3.2.21 DSC_PHY Macro	190
1.4.2.3.2.22 EVENT_HID_ATTACH Macro	190
1.4.2.3.2.23 EVENT_HID_BAD_REPORT_DESCRIPTOR Macro	191
1.4.2.3.2.24 EVENT_HID_DETACH Macro	191
1.4.2.3.2.25 EVENT_HID_NONE Macro	191
1.4.2.3.2.26 EVENT_HID_OFFSET Macro	191
1.4.2.3.2.27 EVENT_HID_READ_DONE Macro	191

1.4.2.3.2.28 EVENT_HID_RESET Macro	192
1.4.2.3.2.29 EVENT_HID_RESET_ERROR Macro	192
1.4.2.3.2.30 EVENT_HID_RPT_DESC_PARSED Macro	192
1.4.2.3.2.31 EVENT_HID_WRITE_DONE Macro	192
1.4.2.3.2.32 USB_HID_CLASS_ERROR Macro	193
1.4.2.3.3 usb_host_hid.h	193
1.4.2.3.4 USB_HOST_HID_RETURN_CODES Enumeration	195
1.4.2.3.5 _USB_HOST_HID_PARSER_H_ Macro	196
1.4.2.3.6 usb_host_hid_parser.h	196
1.4.2.3.7 DSC_RPT_wValue Macro	198
1.4.2.3.8 USB_HID_TRANSFER_IN Macro	198
1.4.2.3.9 USB_HID_TRANSFER_OUT Macro	198
1.4.2.4 Mass Storage Client Driver	199
1.4.2.4.1 Functions	199
1.4.2.4.1.1 USBHostMSDDeviceStatus Function	200
1.4.2.4.1.2 USBHostMSDEventHandler Function	200
1.4.2.4.1.3 USBHostMSDInitialize Function	201
1.4.2.4.1.4 USBHostMSDResetDevice Function	201
1.4.2.4.1.5 USBHostMSDTasks Function	202
1.4.2.4.1.6 USBHostMSDTerminateTransfer Function	202
1.4.2.4.1.7 USBHostMSDTransfer Function	203
1.4.2.4.1.8 USBHostMSDTransferIsComplete Function	203
1.4.2.4.1.9 USBHostMSDRead Macro	204
1.4.2.4.1.10 USBHostMSDWrite Macro	204
1.4.2.4.2 Data Types and Constants	205
1.4.2.4.2.1 DEVICE_CLASS_MASS_STORAGE Macro	206
1.4.2.4.2.2 DEVICE_INTERFACE_PROTOCOL_BULK_ONLY Macro	206
1.4.2.4.2.3 DEVICE_SUBCLASS_CD_DVD Macro	206
1.4.2.4.2.4 DEVICE_SUBCLASS_FLOPPY_INTERFACE Macro	206
1.4.2.4.2.5 DEVICE_SUBCLASS_RBC Macro	207
1.4.2.4.2.6 DEVICE_SUBCLASS_REMOVABLE Macro	207
1.4.2.4.2.7 DEVICE_SUBCLASS_SCSI Macro	207
1.4.2.4.2.8 DEVICE_SUBCLASS_TAPE_DRIVE Macro	207
1.4.2.4.2.9 EVENT_MSD_ATTACH Macro	207
1.4.2.4.2.10 EVENT_MSD_MAX_LUN Macro	208
1.4.2.4.2.11 EVENT_MSD_NONE Macro	208
1.4.2.4.2.12 EVENT_MSD_OFFSET Macro	208
1.4.2.4.2.13 EVENT_MSD_RESET Macro	208
1.4.2.4.2.14 EVENT_MSD_TRANSFER Macro	208
1.4.2.4.2.15 MSD_COMMAND_FAILED Macro	209
1.4.2.4.2.16 MSD_COMMAND_PASSED Macro	209
1.4.2.4.2.17 MSD_PHASE_ERROR Macro	209

1.4.2.4.2.18 USB_MSD_CBW_ERROR Macro	209
1.4.2.4.2.19 USB_MSD_COMMAND_FAILED Macro	209
1.4.2.4.2.20 USB_MSD_COMMAND_PASSED Macro	210
1.4.2.4.2.21 USB_MSD_CSW_ERROR Macro	210
1.4.2.4.2.22 USB_MSD_DEVICE_BUSY Macro	210
1.4.2.4.2.23 USB_MSD_DEVICE_DETACHED Macro	210
1.4.2.4.2.24 USB_MSD_DEVICE_NOT_FOUND Macro	211
1.4.2.4.2.25 USB_MSD_ERROR Macro	211
1.4.2.4.2.26 USB_MSD_ERROR_STATE Macro	211
1.4.2.4.2.27 USB_MSD_ILLEGAL_REQUEST Macro	211
1.4.2.4.2.28 USB_MSD_INITIALIZING Macro	211
1.4.2.4.2.29 USB_MSD_INVALID_LUN Macro	212
1.4.2.4.2.30 USB_MSD_MEDIA_INTERFACE_ERROR Macro	212
1.4.2.4.2.31 USB_MSD_NORMAL_RUNNING Macro	212
1.4.2.4.2.32 USB_MSD_OUT_OF_MEMORY Macro	212
1.4.2.4.2.33 USB_MSD_PHASE_ERROR Macro	212
1.4.2.4.2.34 USB_MSD_RESET_ERROR Macro	213
1.4.2.4.2.35 USB_MSD_RESETTING_DEVICE Macro	213
1.4.2.4.3 usb_host_msd.h	213
1.4.2.4.4 _USBHOSTMSD_H_ Macro	214
1.5 Demo Board Information	215
1.5.1 Low Pin Count USB Development Board	215
1.5.2 PICDEM FS USB Board	217
1.5.3 PIC18 Starter Kit	218
1.5.4 PIC18F46J50 Plug-In-Module (PIM)	219
1.5.5 PIC18F47J53 Plug-In-Module (PIM)	220
1.5.6 PIC18F87J50 Plug-In-Module (PIM) Demo Board	221
1.5.7 PIC24F Starter Kit	222
1.5.8 PIC24FJ256DA210 Development Board	222
1.5.9 Explorer 16	224
1.5.9.1 PIC24FJ256GB110 Plug-In-Module (PIM)	225
1.5.9.2 PIC24FJ256GB210 Plug-In-Module (PIM)	225
1.5.9.3 PIC24FJ64GB004 Plug-In-Module (PIM)	225
1.5.9.4 PIC24EP512GU810 Plug-In-Module (PIM)	226
1.5.9.5 dsPIC33EP512MU810 Plug-In-Module (PIM)	226
1.5.9.6 USB PICTail Plus Daughter Board	226
1.6 Demos	228
1.6.1 Device - Audio Microphone Basic Demo	228
1.6.2 Device - Audio MIDI Demo	230
1.6.2.1 Garage Band '08 [Macintosh Computers]	230
1.6.2.2 Using Linux MultiMedia Studio (LMMS) [Linux and Windows Computers]	232

1.6.3 Device - Boot Loader - HID	234
1.6.3.1 Customizing for an Application	235
1.6.3.1.1 Importance of Change the VID/PID	235
1.6.3.1.2 Safe Boot Loading Considerations	236
1.6.3.1.3 Configuration Bits	236
1.6.3.1.4 Application Version Information	237
1.6.3.1.5 Host Application Responsibilities	237
1.6.3.2 Implementation Details	238
1.6.3.2.1 Command Set	238
1.6.3.2.1.1 QUERY_DEVICE	238
1.6.3.2.1.2 UNLOCK_CONFIG	240
1.6.3.2.1.3 ERASE_DEVICE	240
1.6.3.2.1.4 PROGRAM_DEVICE	241
1.6.3.2.1.5 PROGRAM_COMPLETE	241
1.6.3.2.1.6 GET_DATA	241
1.6.3.2.1.7 RESET_DEVICE	242
1.6.3.2.1.8 SIGN_FLASH	242
1.6.3.2.1.9 QUERY_EXTENDED_INFO	243
1.6.3.2.2 Boot Loader Entry	243
1.6.3.2.2.1 Input Button/Hardware Entry	243
1.6.3.2.2.2 Software/Application Entry	244
1.6.3.2.3 Processor Specific Implementation Details	245
1.6.3.2.3.1 PIC16 and PIC18	245
1.6.3.2.3.2 PIC24F	254
1.6.3.2.4 Flash Signature	264
1.6.4 Device - CDC Basic Demo	265
1.6.4.1 Windows	266
1.6.4.2 Linux	266
1.6.4.3 Macintosh	267
1.6.5 Device - HID - Custom Demo	268
1.6.6 Device - HID - Digitizer Demos	271
1.6.7 Device - HID - Joystick Demo	272
1.6.8 Device - CDC - Serial Emulator	273
1.6.9 Device - HID - Keyboard Demo	274
1.6.10 Device - HID - Mouse Demo	274
1.6.11 Device - HID - Uninterruptible Power Supply	275
1.6.12 Device - Mass Storage - Internal Flash Demo	275
1.6.12.1 Troubleshooting	276
1.6.13 Device - Mass Storage - SD Card Reader	277
1.6.14 Device - Vendor Driver Basic Demo	277
1.6.14.1 Windows	278
1.6.14.2 Linux	279

1.6.14.3 Android 3.1+	280
1.6.15 Device - Vendor High Bandwidth Demo	282
1.6.16 Host - CDC Serial Demo	283
1.6.17 Host - HID - Keyboard Demo	284
1.6.18 Host - HID - Mouse Demo	284
1.6.19 Host - Mass Storage - Thumb Drive Data Logger	285
1.6.20 Host - Mass Storage (MSD) - Simple Demo	286
1.6.21 Configuring the Demo	286
1.6.22 Running the Demo	287
1.6.23 Supported Demo Boards	288
1.7 Appendix (FAQs, Important Information, Reference Material, etc.)	290
1.7.1 Using breakpoints in USB host applications	290
1.7.2 Notes on .inf Files	293
1.7.3 Vendor IDs (VID) and Product IDs (PID)	293
1.7.4 Using a Diff Tool	293
1.7.5 Driver Signing and Windows 8	294
1.7.5.1 What are "Signed" Drivers?	294
1.7.5.2 Minimum Driver Signature Requirements	295
1.7.5.3 Using Older Drivers with Windows 8	295
1.7.5.4 Driver Signatures in the Microchip Libraries for Applications (MLA) Projects	297
1.7.5.5 Obtaining a Microsoft Authenticode Code Signing Certificate	298
1.7.5.6 Code Signing Certificates (Other Uses)	298
1.7.5.7 Using a Code Signing Certificate to Sign Driver Packages	298

USB Library

1 USB Library

1.1 Introduction

A brief summary of what this library is and what it contains.

Description

The USB specification was developed to replace many other non-standard buses and communication ports that used to be found on personal computers. Since its release USB has become standard on nearly all PCs and on many phones, tablets, TVs and various other hardware as a means for standardized communication.

The USB specification is available from the USB Implementer's Forum (USBIF) website, www.usb.org. The USB Library provided by Microchip interfaces to the USB modules on many Microchip microcontroller products providing a basic interface for developers to use to enable USB in their products. Beyond the physical layer interface, the USB Library also implements many of the protocol layers defined in the USB specification assisting designers to create products faster.

The examples include applications for both USB peripherals as well as embedded host examples. For USB peripheral demos, example .inf files and PC code are also provided where applicable.

1.2 Legal Information

Legal information pertinent to this project. Including the software license agreement and trademark information.

Description

Software License Agreement

Copyright 2015 Microchip Technology Inc. (www.microchip.com)

Licensed under the Apache License, Version 2.0 (the "License");

you may not use this file except in compliance with the License.

You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software

distributed under the License is distributed on an "AS IS" BASIS,

WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

See the License for the specific language governing permissions and

limitations under the License.

To request to license the code under the MLA license (www.microchip.com/mla_license),

please contact mla_licensing@microchip.com

Trademark Information

The Microchip name and logo, the Microchip logo, MPLAB, and PIC are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

PICDEM and PICtail are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

Microsoft, Windows, Windows Vista, and Authenticode are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

SD is a trademark of the SD Association in the U.S.A and other countries

1.3 Release Notes

Useful information about this release including revision history, what's next, library migration, demo board support information, operating system support information ,links to tools, support, and other online references.

Description

1.3.1 Revision History

This section describes in more detail the changes made between versions of the MCHPFSUSB stack.

Description

This section describes in more detail the changes made between versions of the MCHPFSUSB stack. This section generally discusses only changes made to the core files (those found in the <install directory>\Microchip folder). This section generally doesn't include changes to the demo projects unless those changes are important to know about. This section also doesn't encompass minor changes to the stack files such as arrangement or locations of definitions or any other organizational changes.

For more information about how to compare the actual source of two different revisions, please see the Appendix - Using a diff tool section of this document.

1.3.1.1 v2.17

- Added support for PIC32MM USB Curiosity Development Board
- Integrated many fixes/changes in the linker files from the latest compiler releases into the boot loader project.

1.3.1.2 v2.16

- Added support for PIC32MM0256GPM064 PIM

1.3.1.3 v2.15

- All demo projects: stack files reverted back to external references rather than imported locally
- Adds HID PC example of how to talk to multiple instances of the same device (same VID/PID)
- Minor fixes/improvements to code comments.
- Improved error handling and throughput for CDC PC example. Additional fix for various carriage return/new line combinations.
- Add support for PIC24F1024GB610 family to HID boot loader

1.3.1.4 v2.14

- Removed requirement to implement system_config.h for USB. Stack now only uses usb_config.h for configuration.
 - Files affected: most stack framework files.
- USBInitialize() and USBTasks() removed. Use USBDeviceTasks()/USBHostTasks() or USBHostInit()/USBDeviceInit() as required.
 - Files affected: usb_common.h
- Fixed dereferencing type-punned pointer will break strict-aliasing warning on full optimization.
 - Files affected: usb_device.c, usb_device_hid.c
- Added MSDTransferTerminated() function.
 - Files affected: usb_device_msd.c
- Added error checking for NULL data handler
 - Files affected: usb_host.c
- Added return for case in USBHostDeviceStatus() for when the device state variable was invalid value.
 - Files affected: usb_host.c
- Fixed error when parsing HID descriptors resulting in invalid errors reported on well formed HID report descriptors.
 - Files affected: usb_host_hid_parser.c
- All demo projects: stack files imported local to the project rather than files from the framework folder. Same for the board support package (bsp) files. All demo projects are now self contained.
- MSD Host bootloader (thumbdrive boot loader): removed. This method of boot loading will be supported in the EZBL boot loader at www.microchip.com/ezbl.
- Added support for the PIC24FJ1024GB610 family devices.

1.3.1.5 v2.13

- Added support for the PIC24FJ256GB410 PIM.
- Now licensed under Apache v2.0 license. Available upon request under MLA license.
- Added EVENT_ALT_INTERFACE and EVENT_HOLD_BEFORE_CONFIGURATION
 - Files affected: usb_common.h
- HID host API updated to allow for simultaneous IN and OUT transfers
 - Files affected: usb_host_hid.c, usb_host_hid.h
- USBHostHIDDeviceDetect() updated to allow for detection of new devices without having to scan all possible addresses.
 - Files affected: usb_host_hid.c, usb_host_hid.h
- Minor optimization for speed and size:
 - Files affected: usb_device.c
- Fixed issue with being able to scan multiple configurations on an attached device:
 - Files affected: usb_host.c
- Allow a driver to not have a data event handler by placing NULL in the TPL table.
 - Files affected: usb_host.c

1.3.1.6 v2.12

- Moved header files to "/inc" folder. Projects need to add path in project to point to "<install directory>/framework/usb/inc" to use the library.
- Fixed issue where stack variable might not be initialized on non-compliant host.
 - Files affected: usb_device.c
- Fixed issue where idle information was getting cleared if a bus error occurred.
 - Files affected: usb_device.c
- Added functionality to allow low speed devices to provide/get timing information to the stack for time sensitive features (control transfers, idle controls, etc.)
- Fixed issue where some drives would not work because the a double write wasn't done when flushing data to the device resulting in some drives to only commit the changes to the local NAND cache rather than committing it to the flash memory.
 - Files affected: usb_host_msd_scsi.c
- Fixed issue with newer drives where an existing device condition caused the drive not to work correctly without first sending the RequestSense request to clear the condition.
 - Files affected: usb_host_msd_scsi.c

1.3.1.7 v2.11

- Added USB Device CDC serial emulator demo for 8-bit devices
- Added USB Windows driver installation/removal tools under: \apps\usb\device\utilities
- Implemented HID idle rate control in USB Device HID Keyboard demo
- Modifications to PIC16F145x USB HID device bootloader memory mapping
- Data transfer rate improvements in USB device msd_sd_card_reader demo
- Bug fixes and project cleanup

1.3.1.8 v2.10

- Major changes to folder structure
- Changed all type definitions and API used by the stack from GenericTypeDefs.h to standard C99 types.
- Better hardware abstraction of the demo board specific features from the example application code.
- Addition of new demo board platforms.
- Expanded demo support for some existing platforms
- PIC32 product support removed. PIC32 products are supported by the MPLAB(R)Harmony Framework (www.microchip.com/Harmony).

1.3.1.9 v2.9j

Updated HID bootloader for PIC18 devices. Added software entry point at 0x001C and robustness features to allow re-entry into bootloader in the event of interrupted erase/program/verify sequence.

- Core USB stack files affected: None
- Individual project files: All PIC18 HID bootloader projects were updated. Additionally, all main.c (or equivalent) USB device project demo files were updated, to reserve the 0x1006 and 0x1016 flash words for storing the "flash signature" and application firmware version values. For usage details, see the inline comments at the top of the main.c file in the HID bootloader firmware project.
- Updated the cross platform PC application software intended to be used with the bootloader firmware. The updated version should be built with Qt 5.0.2 using MinGW 32-bit.

Added "Obtaining a WHQL Certified USB Driver Package" document to the \Microchip\Help folder, along with a reseller rights request form necessary for doing Microsoft Driver Update Acceptable (DUA) submissions to allow re-certification of drivers.

- stack files affected: none, documentation only

1.3.1.10 v2.9i

Android files changed to request for protocol version and to wait for a user configurable startup delay.

- stack files affected: usb_host_android.c

In the WinUSB based device projects, changed Microsoft specific OS descriptors to reside in ROM

- stack files affected: none (only demo specific usb_config.h and usb_descriptors.c files affected)

Added conditional compilation definitions to support PIC16F1454, PIC16F1455, and 'LF' flavored PIC16F145x family devices

- stack files affected: usb_function_cdc.c

1.3.1.11 v2.9h

Android driver condensed to remove protocol specific drivers.

- stack files affected: usb_host_android.c, usb_host_android_protocol_v1.c (removed),
usb_host_android_protocol_v1_local.h (removed)

Support added for registering Android HID reports.

- stack files affected: usb_host_android.c, usb_host_android.h

Added support to ignore protocol, subclass, and/or class in the TPL for a USB host

- stack files affected: usb_host.c, usb_host.h

Added support for a client driver to register for EP0 traffic only

- stack files affected: usb_host.c, usb_host.h

Removed unused variables

- stack files affected: usb_function_audio.c, usb_function_cdc.c

Added support for vendor class specific requests for MS descriptors

- stack files affected: usb_function_generic.c, usb_function_generic.h, usb_device.c

Fixed folder capitalization issue:

- stack files affected: usb_host_printer_primitives.c

Fixed an issue where if a USB host received a report of 0 configurations available on a device, it would cause system issues.

- stack files affected: usb_host.c

1.3.1.12 v2.9g

Android audio and HID support added to accessory driver

- stack files affected: `usb_host_android.c`, `usb_host_android_protocol_v1.c`, `usb_host_android.h`, `usb_host_android_protocol_v1.h`, `usb_host_android_local.h`, `usb_host_android_protocol_v1_local.h`

Cleaning up unused variables in the stack

- stack files affected: `usb_device_cdc.c`, `usb_device_audio.c`

Fixed build issue on Mac/Linux systems for printer host demo

- stack files affected: `usb_host_printer_primitives.c`

Modifications to enable EP0 only driver

- stack files affected: `usb_host.h`, `usb_host.c`, `usb_host_local.h`

Modifications to allow wildcards on TPL table entries

- stack files affected: `usb_host.h`, `usb_host.c`

Fixed issue where a device reporting 0 configurations available would cause the host stack to crash.

- stack files affected: `usb_host.c`

Added support for Microsoft OS Descriptors

- stack files affected: `usb_device.c`, `usb_generic.c`, `usb_generic.h`

Fixed issue with interrupt enable for PIC32MX2 family devices

- stack files affected: `usb_hal_pic32.h`

Write attempts to a drive that is write protected does not report the status correct.

- Stack files affected: `usb_function_msd_multi_sector.c`

1.3.1.13 v2.9f

XC16 and XC32 support added.

- stack files affected: `usb_hal.h`, `usb_ch9.h`, `usb_hal_*.h`, `usb_host_printer.h`, `usb_host_printer_esc_pos.h`, `usb_function_msd.c`, `usb_function_msd_multi_sector.c`, `usb_function_phdc_com_model.c`, `usb_host_printer_esc_pos.c`, `usb_host_printer_pcl_5.c`, `usb_host_printer_postscript.c`, `usb_device.c`, `usb_device_local.h`, `usb_hal_local.h`, `usb_hal_pic24.c`, `usb_hal_pic24f.c`, `usb_host_local.h`, `usb_otg.c`

Fixed issue with PIC32 access to USB registers not being atomic.

- stack files affected: `usb_hal_pic32.h`

Support for PIC16F1459 family devices.

- stack files affected: `usb_hal.h`, `usb_device.c`, `usb_hal_pic16f1.h`, `usb_device_local.h`

Removed `hid_report_in[]` and `hid_report_out[]` buffers from stack files. All HID demos responsible for allocating their own data buffers.

- stack files affected: `usb_function_hid.h`, `usb_device.c`

Moved part specific mapping of BDT to HAL files.

- stack files affected: `usb_hal_dspic33e.h`, `usb_hal_pic16f1.h`, `usb_hal_pic18.h`, `usb_hal_pic24.h`, `usb_hal_pic24e.h`, `usb_hal_pic24f.h`, `usb_hal_pic32.h`

1.3.1.14 v2.9e

1. Read-modify-write race condition in the way the USB interrupt flag was getting cleared on the PIC32 devices.
 - Stack files affected: usb_hal_pic32.h
2. Added option to disable NAK timeouts for CDC host transfers (USB_HOST_CDC_NAK_TIMEOUT)
 - Stack files affected: usb_host_cdc.c
3. The ALLOW_GLOBAL_VID_AND_PID option does not issue the EVENT_OVERRIDE_CLIENT_DRIVER_SELECTION event.
 - Stack files affected: usb_host.c
4. USB host isochronous writes did not function correctly
 - Stack files affected: usb_host.c
5. USB host isochronous writes and reads could not occur during the same frame
 - Stack files affected: usb_host.c
6. NULL pointer dereference could occur if a malloc() call failed during device enumeration in USB host stack while creating the endpoint data structure.
 - Stack files affected: usb_host.c
7. Optimazation settings other than -O0 for C30 could cause MSD internal flash demos not to work.
 - Stack files affected: None (Files.c in user folder updated)

1.3.1.15 v2.9d

1. Data event handler of Android driver not passing events to protocol handler resulting in possible memory leak.
 - Stack files affected: usb_host_android.c
2. Issues with mass storage demos on OS X 10.7 when SD-card is read-only.
 - Stack files affected: usb_function_msd.c
3. Fixed compile warnings when -Wall option selected on C32
 - Stack files affected: usb_host_msd.c
4. Fixed issue with call back redirection macro for EP0 request handler.
 - Stack files affected: usb_device_local.h
5. Added configuration option to disable DTS checking in hardware
 - Stack files affected: usb_device.c
6. Fixed a race condition between the 1msec interrupt and the detach interrupt. If the detach interrupt occurs just before the 1msec interrupt, the interrupt handler could cause the host stack state machine to go into an unknown state requiring a reset of the system to recover. Typically only seen when rapidly attaching/detaching a device repeatedly.
 - Stack files affected: usb_host.c
7. Added an error handing case to check for a size larger than 256.
 - Stack files affected: usb_function_phdc.c
8. Write attempts to a drive that is write protected does not report the status correct.
 - Stack files affected: usb_function_msd.c
9. Updated PHDC code to pass Continua testing
 - Stack files affected: usb_function_phdc.c, usb_function_phdc.h, usb_function_phdc_com_model.c/h added

1.3.1.16 v2.9c

1. Added example showing how to connect to custom HID, LibUSB, WinUSB, and MCHPUSB demos from an Android v3.1+ host.
 - Stack files affected: none
2. Updated libusb driver INF to be signed, so now it can be installed with Windows 7
 - Stack files affected: none
3. Some dsPIC projects not building correctly
 - Stack files affected: `usb_hal_dspic33e.h`, `usb_hal_pic24e.h`

1.3.1.17 v2.9b

1. UART RX functionality fixed on several demos using the PIC24FJ256DA210 development board.
 - Stack files affected: none
2. Race condition fixed in Android OpenAccessory framework that could lead to the accessory not attaching periodically.
 - Stack files affected: `usb_host_android_protocol_v1.c`
3. Added Android Accessory workaround for when Android device attaches in accessory mode without first attaching as the manufacturer's mode (happens when accessory is reset but not detached from bus).
 - Stack files affected: `usb_host_android_protocol_v1.c`, `usb_host_android.c`, `usb_host_android.h`
4. Fixed issue where non-supported Android protocol versions would try to enumerate.
 - Stack files affected: `usb_host_android.c`
5. PIC18F Starter Kit MSD SD card reader demo not working correctly.
 - Stack files affected: none
6. Null pointer dereference on Android OpenAccessory detach event.
 - Stack files affected: `usb_host_android_protocol_v1.c`
7. Removed the restriction of MSD drives with the VID = 0x0930 and PID = 0x6545 for the USB MSD host data logging demo. These drives now show no issues with recent robustness enhancements in the past several releases.
 - Stack files affected: none
8. Link issues on Linux and Macintosh machines for PIC18 demos. The latest versions of the C18 compiler for Linux and Macintosh change the linker and library file capitalization scheme resulting in link errors when using older linker files. Linker files updated to use latest capitalization scheme.
 - Stack files affected: none
9. Cleaned up the configuration bits sections for several processors in several demos.
 - Stack files affected: none
10. CCID demo descriptors updated to enable operation on Macintosh machines.
 - Stack files affected: none
11. Update the precompiled MSD library to support .elf files.
 - Stack files affected: none
12. PCL5 printer host would send out a 0-length packet if an empty string was passed to it. This results in some PCL5 printers to lock up. The updated driver will not send out a text string to a printer if it is empty.
 - Stack files affected: none
13. `USB_HID_FEATURE_REPORT` was assigned the incorrect value.
 - Stack files affected: `usb_host_hid.c`

14. Some CDC device demos had incorrect USB_MAX_NUM_INT definition.
 - Stack files affected: none
15. Added examples showing how to connect to various USB demos with the Android USB host API.
 - Stack files affected: none
16. Optional support for DTS signalling added
 - Stack files affected: usb_function_cdc.c, usb_function_cdc.h
17. Added MIDI host support
 - Stack files affected: usb_host_midi.c, usb_host_midi.h
18. Added Android OpenAccessory boot loader example
 - Stack files affected: none
19. Fixed issues with PIC32 support with the MSD host boot loader. Now supports C32 versions 2.x and later.
 - Stack files affected: none

1.3.1.18 v2.9a

1. Fixes issues in the cross-platform HID boot loader that caused certain hex files not to work if the various sections in the hex file were not in order in increasing address in the .hex file.
 - Stack files affected: none
2. Added UART output support for PIC24FJ256DA210 Development Board in Host – Printer Full sheet demo.
 - Stack files affected: none

1.3.1.19 v2.9

1. Adds PHDC peripheral support.
2. Adds Android accessory support for host mode accessories.
3. Added MPLAB X project files for most demo projects.
4. Added code to allow subclass 0x05 (SFF-8070i devices) to enumerate to the MSD host. Support limited to devices that use SCSI command set only.
 - Stack files affected: usb_host_msd.c
5. Added additional logic to MSD SCSI host code to improve support for various MSD devices by trying to reset various error conditions that may occur.
 - Stack files affected: usb_host_msd_scsi.c
6. Fixed issue with CDC host where SET_CONTROL_LINE_STATE command response was formatted incorrectly.
 - Stack files affected: usb_host_cdc.c
7. Added support for both input and output functionality in the Audio host driver.
 - Stack files affected: usb_host_audio.c
8. Added support for SOF, 1 millisecond timer, and data transfer event notifications to USB host drivers.
 - Stack files affected: usb_host.c
9. Added mechanism for a host client driver to override or reject the stacks selection for the class driver associated with an attached device.
 - Stack files affected: usb_host.c, usb_common.h
10. Fixed an issue with STALL handling behavior on non-EP0 endpoints for PIC24 and PIC32 devices.

- Stack files affected: `usb_device.c`
11. Fixed an issue where some variables/flags were not getting re-initialized correctly after a set configuration event leading to communication issues when ping-pong is enabled and multiple set configuration commands are received.
- Stack files affected: `usb_device.c`
12. Added mechanism to get the handle for the next available ping-pong transfer.
- Stack files affected: `usb_device.h`
13. Fixed incorrect value for `USB_CDC_CONTROL_LINE_LENGTH` Stack files affected: `usb_host_cdc.h`
14. Updated MSD device driver to pass command verifier tests.
- Stack files affected: `usb_device_msd.c`, `usb_device_msd.h`
15. Change to CDC device driver to allow handling of terminated transfers.
- Stack files affected: `usb_device_cdc.c`

1.3.1.20 v2.8

1. Fixed issue with `SetFeature(ENDPOINT_HALT)` handling in the device stack. Error could cause one packet of data to get lost per endpoint after clearing a `ENDPOINT_HALT` event on an endpoint. Issue could also cause the user to lose control of endpoints that may not have been enabled before the `SetFeature(ENDPOINT_HALT)` was received. Parts of the issue described in the following forum thread: <http://www.microchip.com/forums/tm.aspx?m=503200>.
 - Stack files affected: `usb_device.c`
2. Fixed stability issue in device stack when interrupts enabled related to the improper enabling of the interrupt control bits in an interrupt context.
 - Stack files affected: `usb_device.c`
3. Fixed issue STALLs were not handled correctly when event transfers are enabled. This could result in the attached device remaining in a non-responsive state where their endpoints are STALLED.
 - Stack files affected: `usb_host_msd.c`
4. Fixed issue where MSD function driver could not always reinitialize itself to a known state.
 - Stack files affected: `usb_function_msd.c`
5. Added `USBCtrlIEPAllowStatusStage()`, `USBDeferStatusStage()`, `USBCtrlIEPAllowDataStage()`, `USBDeferOUTDataStage()`, `USBOUTDataStageDeferred()`, `USBDeferInDataStage()`, and `USBINDataStageDeferred()` functions. These functions allow users to defer the handling of control transfers received in interrupt context until a later point of time.
 - Stack files affected: `usb_device.c`, `usb_device.h`
6. Fixed issue in PIC18F starter kit SD-card bootloader issue. Bootloader could have errors loading hex files if there was an hex entry starting at an odd address with an even number of bytes in the payload.
 - Stack files affected: none
7. Reorganization of many of the definitions and data types.
 - Stack files affected: `usb_hal_pic18.h`, `usb_hal_pic24.h`, `usb_hal_pic32.h`, `usb_device_local.h`, `usb_device.c`, `usb_device.h`
8. Changed the behavior of the PIC24F HID bootloader linker scripts. The remapping.s file is no longer required. Interrupt vector remapping is now handled by the provided linker scripts (no customization required). Applications should be able to run with the bootloader linker script when either programmed or loaded through the bootloader allowing for more easy development and debugging. Interrupt latency should also be the same when using the bootloader or the debugger. For more information about usage, please refer to the HID bootloader documentation.
9. Changed the behavior of the PIC32 HID bootloader linker scripts. The dual-linker script requirement has been replaced by a single required linker script that should be attached to the application project. Applications should be able to run with the bootloader linker script when either programmed or loaded through the bootloader allowing for more easy development and debugging. Interrupt latency should also be the same when using the bootloader or the debugger. For more information about usage, please refer to the HID bootloader documentation.

10. Added files for the PIC18F starter kit contest winners. Located in "<INSTALL_DIRECTORY>/PIC18F Starter Kit 1/Demos/Customer Submissions/Contest 1"
11. Added initial support for the PIC24FJ256DA210 development board.
12. Added initial support for the PIC24FJ256GB210 Plug-in module.

1.3.1.21 v2.7a

1. Fixed USBSetBDTAddress() macro, so that it correctly loads the entire U1BDTPx register set, enabling the BDT to be anywhere in RAM. Previous implementation wouldn't work on a large RAM device if the linker decided to place the BDT[] array at an address > 64kB.
 - Stack files affected: usb_hal_pic32.h
2. Fixed initialization issue where HID parse result information wasn't cleared before loading with new parse result data.
 - Stack files affected: usb_host_hid_parser.c
3. Update to support the PIC18F47J53 A1 and later revision devices.
 - Stack files affected: usb_device.c
4. Fixed an error on 16-bit and 32-bit processors where a word access could be performed on a byte pointer resulting in possible address errors with odd aligned pointers.
 - Stack files affected: usb_device.c
5. Fixed issue where the USBSleepOnSuspend() function would cause the USB communication to fail after being called when _IPL is equal to 0.
 - Stack files affected: usb_hal_pic24.c
6. Fixed issue where placing the micro in idle mode would cause the host stack to stop sending out SOF packets.
 - Stack files affected: usb_host.c
7. Fixed several issues in the USBConfig.exe
8. Made changes to the starting address of the HID bootloader for PIC32. Reduced the size used by the bootloader. Also added application linker scripts for each processor.
9. Added a three point touch digitizer example
10. Updated some of the PC examples to build and run properly in the 2010 .net Express versions.
11. Added information and batch file showing how to enter a special mode of device manager that allows removal/uninstallation of devices that are not currently attached to the system.

1.3.1.22 v2.7

1. Fixed error where USBHandleGetAddr() didn't convert the return address from a physical address to a virtual address for PIC32.
 - Stack files affected: usb_device.h
2. Added macro versions of USBDeviceAttach() and USBDeviceDetach() so they will compile without error when using polling mode.
 - Stack files affected: usb_device.h
3. Fixes issue in dual role example where a device in polling mode can still have interrupts enabled from the host mode causing an incorrect vectoring to the host interrupt controller while in device mode.
 - Stack files affected: usb_hal_pic18.h, usb_hal_pic24.h, usb_hal-pic32.h, usb_device.c
4. Modified the SetConfigurationOptions() function for PIC32 to explicitly reconfigure the pull-up/pull-down settings for the D+/D- pins in case the host code leaves the pull-downs enabled when running in a dual role configuration.

- Stack files affected: `usb_hal_pic32.h`
5. Fixed error where the USB error interrupt flag was not getting cleared properly for PIC32 resulting in extra error interrupts (<http://www.microchip.com/forums/tm.aspx?m=479085>).
- Stack files affected: `usb_device.c`
6. Updated the device stack to move to the configuration state only after the user event completes.
- Stack files affected: `usb_device.c`
7. Fixed error in the part support list of the variables section where the address of the CDC variables are defined. The PIC18F2553 was incorrectly named PIC18F2453 and the PIC18F4558 was incorrectly named PIC18F4458 (<http://www.microchip.com/forums/fb.aspx?m=487397>).
- Stack files affected: `usb_function_cdc.c`
8. Fixed an error where the `USBHostClearEndpointErrors()` function didn't properly return `USB_SUCCESS` if the errors were successfully cleared (<http://www.microchip.com/forums/fb.aspx?m=490651>).
- Stack files affected: `usb_host.c`
9. Fixed issue where `deviceInfoHID[i].rptDescriptor` was incorrectly freed twice. The second free results in possible issues in future `malloc()` calls in the C32 compiler.
- Stack files affected: `usb_host_hid.c`
10. Fixed an issue where the MSD client driver would issue a transfer events to an incorrect/invalid client driver number when transfer events are enabled.
- Stack files affected: `usb_host_msd.c`
11. Fixed issue where a device that is already connected to the embedded host when the system is initialized may not enumerate.
- Stack files affected: `usb_host.c`
12. Fixed issue where the embedded host or OTG device did not properly check `bmRequestType` when it thinks that a `HALT_ENDPOINT` request was sent to the device. This resulted in the DTS bits for the attached device getting reset causing possible communication issues.
- Stack files affected: `usb_host.c`
13. Changed how the bus sensing works. In previous revisions it was impossible to use the `USBDeviceDetach` to detach from the bus if the bus voltage was still present. This is now possible. It was also possible to move the device to the ATTACHED state in interrupt mode even if the bus voltage wasn't available. This is now prohibited unless VBUS is present.
- Stack files affected: `usb_device.c`
14. Added `USBSleepOnSuspend()` function. This function shows how to put the PIC24F to sleep while the USB module is in suspend and have the USB module wake up the device on activity on the bus.
- Stack files affected: `usb_hal_pic24.h, usb_hal_pic24.c`
15. Modified the code to allow connection of USB-RS232 dongles that do not fully comply with CDC specifications.
- Stack files affected: `usb_host_cdc.h, usb_host_cdc.c, usb_host_cdc_interface.c, usb_host_interface.h`
16. Modified API `USBHostCDC_Api_Send_OUT_Data` to allow data transfers more than 256 bytes.
- Stack files affected: `usb_host_cdc.h, usb_host_cdc.c, usb_host_cdc_interface.c, usb_host_interface.h`
17. Improved error case handling when the host sends more OUT bytes in a control transfer than the firmware was expecting to receive (based on the size parameter when calling `USBEP0Receive()`).
- Stack files affected: `usb_device.c`
18. Added CCID (Circuit Cards Interface Device) class device/function support.
- Stack Files affected: `usb_function_ccid.h, usb_function_ccid.c`
19. Added Audio v1 class embedded host support.
- Stack files affected: `usb_host_audio_v1.h, usb_host_audio_v1.c`

1.3.2 Support

Find out how to get help with your USB design, support questions, or USB training.

Description

The Microchip Web Site

Microchip provides online support via our web site at <http://www.microchip.com>. This web site is used as a means to make files and information easily available to customers. Accessible by using your favorite Internet browser, the web site contains the following information:

- Product Support - Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- General Technical Support - Frequently Asked Questions (FAQs), technical support requests (<http://support.microchip.com>), online discussion groups/forums (<http://forum.microchip.com>, or more specifically the USB forum topic), Microchip consultant program member listing
- Business of Microchip - Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

Development Systems Customer Change Notification Service

Microchip's customer notification service helps keep customers current on Microchip products. Subscribers will receive e-mail notification whenever there are changes, updates, revisions or errata related to a specified product family or development tool of interest.

To register, access the Microchip web site at www.microchip.com, click on Customer Change Notification and follow the registration instructions.

Additional Support

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Field Application Engineer (FAE)
- Technical Support

Customers should contact their distributor, representative or field application engineer (FAE) for support. Local sales offices are also available to help customers. A listing of sales offices and locations is available on our website.

Technical support is available through the web site at: <http://support.microchip.com>

Training

- Regional Training Centers: <http://www.microchip.com/rtc>
- MASTERs Conference: <http://www.microchip.com/masters>
- Webseminars: <http://techtrain.microchip.com/webseminars/QuickList.aspx>

1.3.3 Online Reference and Resources

This section includes useful links to online USB development resources.

Description

Note: Newer versions of the documents below may be available. Please check www.microchip.com for the latest version.

USB Design Center

<http://www.microchip.com/usb>

Application Notes

[Microchip USB Device Firmware Framework User's Guide](#)

[AN950 – Power Management for PIC18 USB Microcontrollers with nanoWatt Technology](#)

[AN956 – Migrating Applications to USB from RS-232 UART with Minimal Impact on PC Software](#)

[AN1140 – USB Embedded Host Stack](#)

[AN1141 – USB Embedded Host Stack Programmer's Guide](#)

[AN1142 – USB Mass Storage Class on an Embedded Host](#)

[AN1143 – Generic Client Driver for a USB Embedded Host](#)

[AN1144 - USB Human Interface Device Class on an Embedded Host](#)

[AN1145 – Using a USB Flash Drive on an Embedded Host](#)

[AN1189 – Implementing a Mass Storage Device Using the Microchip](#)

[AN1212 – Using USB Keyboard with an Embedded Host](#)

[AN1233 – USB Printer Class on an Embedded Host](#)

USB Demonstration Videos

<http://www.youtube.com/watch?v=ljF4KQ2mfD0>

http://www.youtube.com/watch?v=cmtjKUv_yPs&feature=related

<http://www.youtube.com/watch?v=BOosLeO7D58&feature=related>

1.3.4 Device (Slave) Demo Board Support and Limitations

This section shows which USB device demos are supported on each of the USB demo boards.

Description

This section shows which USB device demos are supported on each of the USB demo boards.

Legend

Supported
See Limitations
Not Supported

	Audio - Microphone	Audio - MIDI	Bootloader	Composite - HID + MSD	CDC - Basic	CDC - UART bridge example	HID - Custom	HID - Digitizer Multi Touch	HID - Digitizer Single Touch	HID - Joystick	HID - Keyboard	HID - Mouse	HID - Uninterruptible Power Supply (UPS)	MSD - Internal Flash	MSD - SD Card Reader	Vendor Class - Basic	Vendor Class - High Throughput
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
8-bit demo boards (PIC16 and PIC18 families)																	
Low Pin Count USB Development Kit (PIC16F1459 Family)			2														
Low Pin Count USB Development Kit (PIC18F14K50 Family)				2													
PIC18F46J50 Full Speed USB Development Board (PIM)																	
PIC18F47J53 Full Speed USB Development Board (PIM)																	
PIC18F87J50 Full Speed USB Development Board (PIM)				2													
PIC18F87J94 Full Speed USB Development Board (PIM)																	
PIC18F97J94 Full Speed USB Development Board (PIM)																	
PIC18F Starter Kit 1 (PIC18F46J50 Family)																	
PICDEM Full Speed USB (PIC18F4550 Family)																	
PICDEM Full Speed USB (PIC18F45K50 Family)																	
16-bit demo boards (PIC24 and dsPIC families)																	
PIC24EP512GU810 Plug-In-Module (PIM)																	
PIC24FJ64GB004 Plug-In-Module (PIM)																	
PIC24FJ128GB204 Plug-In Module (PIM)																	
PIC24FJ256GB110 Plug-In-Module (PIM)																	
PIC24FJ256GB210 Plug-In-Module (PIM)																	
PIC24FJ256GB410 Plug-In-Module (PIM)																	
PIC24FJ256DA210 Development Board																	
PIC24FJ1024GB610 Plug-In-Module (PIM)																	
dsPIC33EP256MU506 Plug-In-Module (PIM)																	
dsPIC33EP512MU810 Plug-In-Module (PIM)																	
MPLAB Starter Kit for PIC24F	1	1			1	1			1	1	1	1	1	1			1
MPLAB Starter Kit for PIC24F Intelligent Integrated Analog																	
MPLAB Starter Kit for PIC24E MCUs																	
MPLAB Starter Kit for dsPIC33E DSCs																	
32-bit demo boards (PIC32MM families)																	
PIC32MM0256GPM064 Plug-In-Module (PIM)																	
PIC32MM USB Curiosity Development Board																	

Limitations

- 1) The PIC24F starter kit does not have a physical push button. The board uses capacitive touch buttons instead. The cap touch functionality has not been added to the demos yet so the functionality required by the demos so some or all of the features of this demo are limited.
- 2) Bootloader operation with the PIC16F145x device family requires the use of revision A5 or later silicon

1.3.5 Host Demo Board Support and Limitations

This section shows which USB host demos are supported on each of the USB demo boards.

Description

This section shows which USB host demos are supported on each of the USB demo boards.

Legend	
Supported	
See Limitations	
Not Supported	

	CDC - Basic	HD - Keyboard	HD - Mouse	MSD - Data Logger	MSD - Simple
	A	B	C	D	E
16-bit demo boards (PIC24 and dsPIC families)					
PIC24FJ64GB004 Plug-In-Module (PIM)		1			
PIC24FJ128GB204 Plug-In Module (PIM)	2	1,2	2		
PIC24FJ256GB110 Plug-In-Module (PIM)		1			
PIC24FJ256GB210 Plug-In-Module (PIM)		1			
PIC24FJ256GB410 Plug-In-Module (PIM)		1			
PIC24FJ1024GB610 Plug-In-Module (PIM)		1			
PIC24EP512GU810 Plug-In-Module (PIM)		1			
dsPIC33EP256MU506 Plug-In-Module (PIM)	2	1,2	2		
dsPIC33EP512MU810 Plug-In-Module (PIM)		1			
PIC24FJ256DA210 Development Board	2	1,2	2		
MPLAB Starter Kit for PIC24F		1		3	
MPLAB Starter Kit for PIC24F Intelligent Integrated Analog		1			
MPLAB Starter Kit for PIC24E MCUs		1		4	
MPLAB Starter Kit for dsPIC33E DSCs		1		4	
32-bit demo boards (PIC32MM families)	A	B	C	D	E
PIC32MM0256GPM064 Plug-In-Module (PIM)		1			

Limitations

- 1) Neither compound nor composite devices are supported. Some keyboards are either compound or composite.

The “~” prints as an arrow character instead (“->”). This is an effect of the LCD screen on the Explorer 16. The ascii character for “~” is remapped in the LCD controller.

The “\” prints as a “¥” character instead. This is an effect of the LCD screen on the Explorer 16. The ascii character for “\” is remapped in the LCD controller.

Backspace and arrow keys may have issues on Explorer 16 boards with certain LCD modules

- 2) The display features of this demo have not been ported to this board yet.
- 3) This board does not have a push button and the code required to implement the capacitive touch buttons has not been implemented yet.
- 4) There is no potentiometer available on this board. A fixed value of 512 is reported in place of the potentiometer data.

1.3.6 Operating System Support and Limitations

This section describes which operating systems support each of the provided demos.

Description

This section describes which operating systems support each of the provided demos.

Legend	
Supported	Green
Limited Support	Yellow
Not Supported	Red
Not Tested	Purple

Operating System

K Android OS v3.1
J Macintosh OS 10.7
I Macintosh OS 10.5
H Linux
G Windows 7 (64-bit)
F Windows 7 (32-bit)
E Windows Vista (64-bit)
D Windows Vista (32-bit)
C Windows XP (64-bit)
B Windows XP (32-bit)
A Windows 2000

Demos

	A	B	C	D	E	F	G	H	I	J	K
USB Device - Audio - Microphone											8
USB Device - Audio - MIDI											8
USB Device - Audio - Speaker											8
USB Device - Bootloaders											1
USB Device - CCID - SmartCard Reader											8
USB Device - CDC - Basic Demo					6	6					8
USB Device - CDC - Serial Emulator					6	6					8
USB Device - Composite - HID + MSD											8
USB Device - Composite - MSD + CDC		4	4					9			8
USB Device - HID - Custom Demos											1
USB Device - HID - Digitizers				3	3						8
USB Device - HID - Joystick											8
USB Device - HID - Keyboard	7										8
USB Device - HID - Mouse											8
USB Device - HID - Uninterruptible Power Supply											8
USB Device - LibUSB - Generic Driver Demo								2	2	1	8
USB Device - Mass Storage - Internal Flash											8
USB Device - Mass Storage - SD Card data logger											8
USB Device - Mass Storage - SD Card reader											8
USB Device - PHDC Demos			5	5	5	5	5	8	8	8	8
USB Device - MCHPUSB - Generic Driver Demo	5	5	5	5	5	5	5				1
USB Device - WinUSB - Simple Custom Demo											1
USB Device - WinUSB - High Bandwidth Demo											1
USB Dual Role - MSD host + HID device											1
USB OTG - MCHPUSB - Generic Driver Demo											1
USB PC - WM_DEVICECHANGE Demo											8
USB Configuration Tool											8

Limitations

- 1) These devices enumerate successfully by the OS but currently there is not an example program to interface these devices.
- 2) Devices that implement the LibUSB demo will enumerate successfully on Macintosh based operating systems (provided the correct drivers are installed). Currently there is not an example program to communicate to these devices on these operating systems in this installation.
- 3) Only single touch gestures are supported in Windows Vista. For the multi touch demo only the single touch gestures will work as a gesture. The multi touch gestures in Vista will appear as two separate touch events that do not produce a usable pattern.
- 4) When used with Windows XP SP2 or earlier, this demo requires a Microsoft hotfix in order to run properly. This hotfix is linked from the demo folder. Windows XP SP3 works properly without needing any hotfix.
- 5) When adding a VID/PID string to the "%DESCRIPTION%=DriverInstall" and "%DESCRIPTION%=DriverInstall64" sections in the mchpusb.inf file, remove one or more of the pre-existing VID/PID strings from the list. There is a limit to the maximum number of VID/PID strings that can be supported simultaneously. If the list contains too many entries, the following error message will occur when installing the driver under Vista: "The Data Area Passed to a System Call Is Too Small"
- 6) The CDC PC example code does not run as implemented on the 64-bit version of the Windows Vista operating system with some versions of the .net framework. The .NET SerialPort object does not appear to receive data as implemented in these examples in the early versions of the .net framework for Vista.
- 7) The HID keyboard example does not work as implemented on the Windows 2000 operating system or any earlier revisions of the Windows operating systems.

- 8) Firmware successfully enumerates but test machine was unable to verify functionality. This is either due to lack of support in the OS for these types of devices or lack of an Application that uses these devices.
- 9) This demo uses the USB IAD specification. Some versions of Macintosh OSX do not support IAD.

1.3.7 Tool Information

Specifies the versions of the tools used to test this release.

Description

This release was tested with the following tools:

Compiler	Version
MPLAB XC8	1.37
MPLAB XC16	1.26

IDE	Version
MPLAB X	3.26

Some demos in this release require the full versions of the above compilers (the boot loaders and a few of the demo applications). For most demos, either the commercial version, or the evaluation version can be used to build the example projects. Some The compilers may be obtained from <http://www.microchip.com/xc8> and <http://www.microchip.com/xc16>.

1.3.8 Library Migration

1.3.8.1 From v2.13 to v2.14+

- USBTasks() has been removed.
 - For device mode, call USBDeviceTasks()
 - For host mode, call USBHostTasks()
- USBInitialize() has been removed.
 - For device mode, call USBDeviceInit()
 - For host mode, call USBHostInit()

1.3.8.2 From v2.12 to v2.13

USB Host users:

- Make sure you are returning FALSE at the end of the event handler for all unhandled cases. There are two new events added and by returning FALSE, they will operate just as v2.12 and previous revisions did.

USB HID Host users:

- USBHostHIDDeviceDetect() used to take in an address and return a bool if it was ready or not. Now the API doesn't take in any parameters and returns the address of an attached but previously unannounced device.

- Note: every address is only reported once. If you are using multiple HID applications (e.g. targeting both a mouse or a keyboard), you may need to share the device attach information between these two applications. Alternatively you can use the attach event rather than the polling function.
- The following legacy APIs were removed and users should migrate to the corresponding updated functions:
 - USBHostHID_ApiDeviceDetect() should be replaced by USBHostHIDDeviceDetect()
 - USBHostHID_ApiGetReport() should be replaced by USBHostHIDRead()
 - USBHostHID_ApiSendReport() should be replaced by USBHostHIDWrite()
 - USBHostHID_ApiResetDevice() should be replaced by USBHostHIDResetDevice()
 - USBHostHIDTransferIsComplete() is now replaced by either USBHostHIDWriteIsComplete() or USBHostHIDReadIsComplete() based on the direction of the transfer.

1.3.8.3 From v2.11 to v2.12

- Add "<install directory>/framework/usb/inc" to project include path.
- Remove any "usb/" tags in the include files of any files in your project. For example: change from "<usb/usb.h>" to "<usb.h>".

1.3.8.4 From v2.10 to v2.11

Any application projects targeting a PIC16F145x device that is also using the HID bootloader should be modified to use the new bootloader firmware and updated application linker settings. The application "codeoffset" has been changed to 0x904, and the "ROM ranges" has changed to "default,-0-903". The new settings resolve a potential linking problem that could previously occur if the application interrupt handler code section grew too large and would attempt to overlap the locations reserved for the signature word and version word.

1.3.8.5 From v2.9j to v2.10

- Type definition changes

In this release all type definitions were changed from GenericTypeDefs.h to the standard C99 types. The size and signedness of the variables remained the same. Applications that used the GenericTypeDefs.h file may need to port their types as well but it should not affect behavior at all. Below is a list of some of the most common usages and their transitions:

GenericTypeDefs.h (old)	C99 (new)	New Header Required
DWORD, UINT32	uint32_t	stdint.h
WORD, UINT16	uint16_t	stdint.h
BYTE, UINT8	uint8_t	stdint.h
BOOL	bool	stdbool.h
TRUE	true	stdbool.h
FALSE	false	stdbool.h

- All 16-bit peripheral/device applications:

Previous versions of the stack redefined the USBDeviceTasks() function to the interrupt vector function, _USB1Interrupt(), when the stack was run in interrupt mode. This meant that users didn't have to call the USBDeviceTasks() function for 16-bit products in interrupt mode. The side effect, however, is that since the host stack took the same approach, dual role or OTG solutions could not use interrupt mode for their peripheral/device operation. In this release of the stack this redefinition has been removed. As such, 16-bit applications using interrupt mode must now define the USB interrupt handler function in the application space and call the USBDeviceTasks() function:

```
#if defined(USB_INTERRUPT)
```

```
void __attribute__((interrupt,auto_psv)) _USB1Interrupt()
{
    USBDeviceTasks();
}
#endif
```

- All 16-bit host applications:

Previous versions of the stack redefined the USB interrupt vector function, _USB1Interrupt(), in the host stack in order to handle USB interrupts. The side effect, however, is that since the host stack took control over the USB interrupt vector, dual role or OTG solutions could not use interrupt mode for their peripheral/device operation. In this release of the stack this behavior has been changed so that the host stack doesn't take control over the USB interrupt vector. As such, 16-bit applications must now define the USB interrupt handler function in the application space and call the USB_HostInterruptHandler() function:

```
void __attribute__((interrupt,auto_psv)) _USB1Interrupt()
{
    USB_HostInterruptHandler();
}
```

- Include paths

Since the overall folder structure of the MLA has changed, if porting between two versions of the USB Library, a user will need to modify the include paths so that the application points to the new library folder and to the application space. To point to the library, the include path should have a link to the "<MLA install directory>\framework" folder. To include a USB header file you would designate the "usb\" folder before specifying the header file required.

Example - #include "usb\usb.h"

1.3.8.6 From v2.9i to v2.9j

No changes required.

However, if using the new HID bootloader features for PIC18 devices, you must rebuild both the bootloader firmware, and the application firmware project (using the updated vector remapping section from the main.c file of the application project). You must also use the updated HID bootloader firmware + application firmware with the updated HID bootloader cross platform software.

1.3.8.7 From v2.9h to v2.9i

No changes required.

1.3.8.8 From v2.9g to v2.9h

No changes required.

1.3.8.9 From v2.9f to v2.9g

No changes required.

1.3.8.10 From v2.9e to v2.9f

1. hid_report_in and hid_report_out were removed from the stack. For HID based demos, the user buffers must be defined in user space. For certain product families that have specific USB RAM limitations, make sure that these buffers get

located in that USB RAM space. Please refer to the existing HID demos to see how the hid_report_in and hid_report_out were moved to user space for those demos.

1.3.8.11 From v2.9d to v2.9e

No changes required

1.3.8.12 From v2.9c to v2.9d

No changes required.

1.3.8.13 From v2.9b to v2.9c

No changes required.

1.3.8.14 From v2.9a to v2.9b

No changes required.

1.3.8.15 From v2.9 to v2.9a

No changes required.

1.3.8.16 From v2.8 to v2.9

No changes required.

1.3.8.17 From v2.7a to v2.8

1. HID Bootloader for PIC32 devices

- An error was fixed in PIC32 bootloader. The previous implementations placed the interrupt vector table on a 1K-page aligned boundary. This table should be on a such a boundary. The user reset vector and the interrupt vector section addresses were switched to meet this requirement. Applications/bootloaders using the old reset vector will not work with applications/bootloaders using the new bootloader linker files.

1.3.8.18 From v2.7 to v2.7a

1. HID Bootloader for PIC32 devices

- The PIC32 bootloader was changed in this revision. The memory region used by the HID bootloader was reduced. This could result in issues loading application projects built with the new linker scripts on a system with the old bootloader. It could also result in issues loading an old application with the new bootloader.

1.3.8.19 From v2.6a to v2.7

No changes required.

1.3.8.20 From v2.6 to v2.6a

1. HID Bootloader for PIC24F devices

- The HID Bootloader for PIC24F has been reworked for the v2.6a release. The change involve how interrupt remapping is handled and how applications relocate their code to make room for the bootloader. Applications built with the v2.6 or earlier PIC24F compiler should continue using the v2.6 bootloader and support files. It is recommended for new projects that new bootloader and support files should be used.
- In previous revisions of the stack there was a “PIC24F HID Bootloader Remapping.s” file that was added to any PIC24F project to relocate the application code out of the bootloader space. These files have been deprecated and should not be used with the new revision of the bootloader. Instead there is a custom linker script (boot_hid_p24fjxxxGBxxx.gld) file in the HID bootloader folder specifically designed for the application. These are located in the “Application Files” folder in each of the respective bootloader folders. Copy this file from this folder into the application folder and add it to the target project. All of the possible interrupts should already be remapped. To use an interrupt, merely define the interrupt handler as you normally would if you weren’t using a bootloader.
- The bootloader for PIC18 and PIC32 devices were not modified.

1.3.8.21 From v2.5 to v2.6

1. Include Files

- The files that must be included into a project has changed from v2.5 to v2.6.
- Version v2.5 of the MCHPFSUSB stack required multiple include files in order to work properly in device mode. The usb_device.h, usb.h, usb_config.h, and class specific files (i.e. - “./usb/usb_function_msd.h”) had to be included in all of the application files that accessed the USB stack as well as other common include files like the GenericTypeDefs.h and Compiler.h files.
- In MCHPFSUSB v2.6, only the usb.h file and the class specific files (i.e. - “./usb/usb_function_msd.h”) must be included in the project. The usb_device.h and usb_config.h files should no longer be included in the application specific files.

2. Include Search Paths and Build Directory Policy

- The preferred include path list has changed since the initial v2.x release. MPLAB now support compiling projects with respect to the project file instead of the source file. This is now the preferred method. With this modification the required include paths are the following:
 - .
 - ..//Microchip/Include
- If your project file located in a different format than the example projects, please add or remove the appropriate path modifiers such that the include path indirectly points to the /Microchip/Include folder.
- To change the build directory policy and set the include paths, go to the “Project->Build Options->Project” menu. On the directories tab, select the include directories from the show directories drop down box.

3. Disabling Interrupt Handlers

- In MCHPFSUSB v2.6, the interrupt handler routines are disabled through the usb_config.h file using the following definitions:
 - USB_DISABLE_SET_CONFIGURATION_HANDLER

- `USB_DISABLE_SUSPEND_HANDLER`
- `USB_DISABLE_WAKEUP_FROM_SUSPEND_HANDLER`
- `USB_DISABLE_SOF_HANDLER`
- `USB_DISABLE_ERROR_HANDLER`
- `USB_DISABLE_NONSTANDARD_EP0_REQUEST_HANDLER`
- `USB_DISABLE_SET_DESCRIPTOR_HANDLER`
- `USB_DISABLE_TRANSFER_COMPLETE_HANDLER`
- Defining any of these definitions in the `usb_config.h` file will disable the callback from the stack during these events. Please note that some of these events are required to be USB compliant. For example all USB devices must go into suspend mode when requested. The suspend handler is how the stack notifies the user that the bus has requested the device to go into suspend mode.
- Also note that some device classes or demos may require certain handlers to be available in order to operate properly. For example, the audio class demo uses the start of frames provided by the SOF handler to properly synchronize the audio data playback.

1.4 Library Interface

This section describes the Application Programming Interface (API) functions of the USB Library.

Refer to each section for a detailed description.

1.4.1 Device/Peripheral

Modules

Name	Description
Device Stack	
Audio Function Driver	
CDC Function Driver	
HID Function Driver	
MSD Function Driver	
Vendor Class (Generic) Function Driver	

Description

1.4.1.1 Device Stack

Files

Name	Description
usb_device.h	This is file usb_device.h.

Macros

Name	Description
_USB_DEVICE_H	Defines types and APIs associated with the USB device stack.

Description

1.4.1.1.1 Functions

Functions

	Name	Description
💡	USB_APPLICATION_EVENT_HANDLER	This function is called whenever the USB stack wants to notify the user of an event.
💡	USBCancelIO	This function cancels the transfers pending on the specified endpoint. This function can only be used after a SETUP packet is received and before that setup packet is handled. This is the time period in which the EVENT_EP0_REQUEST is thrown, before the event handler function returns to the stack.

	USBCtrlEPAllowDataStage	This function allows the data stage of either a host-to-device or device-to-host control transfer (with data stage) to complete. This function is meant to be used in conjunction with either the USBDeferOUTDataStage() or USBDeferINDataStage(). If the firmware does not call either USBDeferOUTDataStage() or USBDeferINDataStage(), then the firmware does not need to manually call USBCtrlEPAllowDataStage(), as the USB stack will call this function instead.
	USBCtrlEPAllowStatusStage	This function prepares the proper endpoint 0 IN or endpoint 0 OUT (based on the controlTransferState) to allow the status stage packet of a control transfer to complete. This function gets used internally by the USB stack itself, but it may also be called from the application firmware, IF the application firmware called the USBDeferStatusStage() function during the initial processing of the control transfer request. In this case, the application must call the USBCtrlEPAllowStatusStage() once, after it has fully completed processing and handling the data stage portion of the request. If the application firmware has no need for delaying... more
	USBDeferINDataStage	This function will cause the USB hardware to continuously NAK the IN token packets sent from the host, during the data stage of a device to host control transfer. This allows the firmware more time to process and prepare the IN data packets that will eventually be sent to the host. This is also useful, if the firmware needs to process/prepare the IN data in a different context than what the USBDeviceTasks() function executes at. Calling this function (macro) will assert ownership of the currently pending control transfer. Therefore, the USB stack will not STALL when it reaches the... more
	USBDeferOUTDataStage	This function will cause the USB hardware to continuously NAK the OUT data packets sent from the host, during the data stage of a device to host control transfer. This allows the firmware more time to prepare the RAM buffer that will eventually be used to receive the data from the host. This is also useful, if the firmware wishes to receive the OUT data in a different context than what the USBDeviceTasks() function executes at. Calling this function (macro) will assert ownership of the currently pending control transfer. Therefore, the USB stack will not STALL when it reaches... more
	USBDeferStatusStage	Calling this function will prevent the USB stack from automatically enabling the status stage for the currently pending control transfer from completing immediately after all data bytes have been sent or received. This is useful if a class handler or USB application firmware project uses control transfers for sending/receiving data over EP0, but requires time in order to finish processing and/or to consume the data. For example: Consider an application which receives OUT data from the USB host, through EP0 using control transfers. Now assume that this application wishes to do something time consuming with this data (ex: transmit it... more
	USBDeviceAttach	Checks if VBUS is present, and that the USB module is not already initialized, and if so, enables the USB module so as to signal device attachment to the USB host.
	USBDeviceDetach	This function configures the USB module to "soft detach" itself from the USB host.
	USBDeviceInit	This function initializes the device stack it in the default state. The USB module will be completely reset including all of the internal variables, registers, and interrupt flags.

	USBDeviceTasks	This function is the main state machine/transaction handler of the USB device side stack. When the USB stack is operated in "USB_POLLING" mode (usb_config.h user option) the USBDeviceTasks() function should be called periodically to receive and transmit packets through the stack. This function also takes care of control transfers associated with the USB enumeration process, and detecting various USB events (such as suspend). This function should be called at least once every 1.8ms during the USB enumeration process. After the enumeration process is complete (which can be determined when USBGetDeviceState() returns CONFIGURED_STATE), the USBDeviceTasks() handler may be called the... more
	USBEnableEndpoint	This function will enable the specified endpoint with the specified options
	USBEP0Receive	Sets the destination, size, and a function to call on the completion of the next control write.
	USBEP0SendRAMPtr	Sets the source, size, and options of the data you wish to send from a RAM source
	USBEP0SendROMPtr	Sets the source, size, and options of the data you wish to send from a const source
	USBEP0Transmit	Sets the address of the data to send over the control endpoint
	USBGet1msTickCount	This function retrieves a 32-bit unsigned integer that normally increments by one every one millisecond. The count value starts from zero when the USBDeviceInit() function is first called. See the remarks section for details on special circumstances where the tick count will not increment.
	USBGetTicksSinceSuspendEnd	This function retrieves a 8-bit unsigned byte that represents the number of milliseconds that have elapsed since the end of a USB suspend event. The value saturates at 255.
	USBGetDeviceState	This function will return the current state of the device on the USB. This function should return CONFIGURED_STATE before an application tries to send information on the bus.
	USBGetNextHandle	Retrieves the handle to the next endpoint BDT entry that the USBTransferOnePacket() will use.
	USBGetRemoteWakeUpStatus	This function indicates if remote wakeup has been enabled by the host. Devices that support remote wakeup should use this function to determine if it should send a remote wakeup.
	USBGetSuspendState	This function indicates if the USB port that this device is attached to is currently suspended. When suspended, it will not be able to transfer data over the bus.
	USBHandleBusy	Checks to see if the input handle is busy
	USBHandleGetAddr	Retrieves the address of the destination buffer of the input handle
	USBHandleGetLength	Retrieves the length of the destination buffer of the input handle
	USBIncrement1msInternalTimers	This function increments internal 1ms time base counters, which are useful for application code (that can use a 1ms time base/counter), and for certain USB event timing specific purposes. In USB full speed applications, the application code does not need to (and should not) explicitly call this function, as the USBDeviceTasks() function will automatically call this function whenever a 1ms time interval has elapsed (assuming the code is calling USBDeviceTasks() frequently enough in USB_POLLING mode, or that USB interrupts aren't being masked for more than 1ms at a time in USB_INTERRUPT mode). In USB low speed applications, the... more

USBINDDataStageDeferred	Returns true if a control transfer with IN data stage is pending, and the firmware has called USBDeferINDataStage(), but has not yet called USBCtrlIEPAllowDataStage(). Returns false if a control transfer with IN data stage is either not pending, or the firmware did not call USBDeferINDataStage() at the start of the control transfer. This function (macro) would typically be used in the case where the USBDeviceTasks() function executes in the interrupt context (ex: USB_INTERRUPT option selected in usb_config.h), but the firmware wishes to take care of handling the data stage of the control transfer in the main... more
USBIsBusSuspended	This function indicates if the USB bus is in suspend mode.
USBIsDeviceSuspended	This function indicates if the USB module is in suspend mode.
USBOUTDataStageDeferred	Returns true if a control transfer with OUT data stage is pending, and the firmware has called USBDeferOUTDataStage(), but has not yet called USBCtrlIEPAllowDataStage(). Returns false if a control transfer with OUT data stage is either not pending, or the firmware did not call USBDeferOUTDataStage() at the start of the control transfer. This function (macro) would typically be used in the case where the USBDeviceTasks() function executes in the interrupt context (ex: USB_INTERRUPT option selected in usb_config.h), but the firmware wishes to take care of handling the data stage of the control transfer in the main... more
USBRxOnePacket	Receives the specified data out the specified endpoint
USBSoftDetach	This function performs a detach from the USB bus via software.
USBStallEndpoint	Configures the specified endpoint to send STALL to the host, the next time the host tries to access the endpoint.
USBTransferOnePacket	Transfers a single packet (one transaction) of data on the USB bus.
USBTxOnePacket	Sends the specified data out the specified endpoint

Module

Device Stack

Description**1.4.1.1.1 USB_APPLICATION_EVENT_HANDLER Function**

This function is called whenever the USB stack wants to notify the user of an event.

File

usb_device.h

Syntax

```
bool USB_APPLICATION_EVENT_HANDLER(uint8_t address, USB_EVENT event, void * pdata, uint16_t size);
```

Returns

None

Description

This function is called whenever the USB stack wants to notify the user of an event. This function should be implemented by the user.

Example Usage:

Remarks

None

Preconditions

None

Function

```
bool USB_APPLICATION_EVENT_HANDLER(uint8_t address, USB_EVENT event, void *pdata, uint16_t size);
```

1.4.1.1.1.2 USBCancelIO Function

File

usb_device.h

Syntax

```
void USBCancelIO(uint8_t endpoint);
```

Description

This function cancels the transfers pending on the specified endpoint. This function can only be used after a SETUP packet is received and before that setup packet is handled. This is the time period in which the EVENT_EPO_REQUEST is thrown, before the event handler function returns to the stack.

Remarks

None

Function

```
void USBCancelIO(uint8_t endpoint)
```

1.4.1.1.1.3 USBCtrlIEPAllowDataStage Function

This function allows the data stage of either a host-to-device or device-to-host control transfer (with data stage) to complete. This function is meant to be used in conjunction with either the USBDeferOUTDataStage() or USBDeferINDataStage(). If the firmware does not call either USBDeferOUTDataStage() or USBDeferINDataStage(), then the firmware does not need to manually call USBCtrlIEPAllowDataStage(), as the USB stack will call this function instead.

File

usb_device.h

Syntax

```
void USBCtrlIEPAllowDataStage();
```

Preconditions

A control transfer (with data stage) should already be pending, if the firmware calls this function. Additionally, the firmware should have called either USBDeferOUTDataStage() or USBDeferINDataStage() at the start of the control transfer, if the firmware will be calling this function manually.

Function

```
void USBCtrlIEPAllowDataStage(void);
```

1.4.1.1.1.4 USBCtrlIEPAllowStatusStage Function

This function prepares the proper endpoint 0 IN or endpoint 0 OUT (based on the controlTransferState) to allow the status stage packet of a control transfer to complete. This function gets used internally by the USB stack itself, but it may also be called from the application firmware, IF the application firmware called the USBDeferStatusStage() function during the initial processing of the control transfer request. In this case, the application must call the USBCtrlIEPAllowStatusStage() once,

after it has fully completed processing and handling the data stage portion of the request.

If the application firmware has no need for delaying control transfers, and therefore never calls `USBDeferStatusStage()`, then the application firmware should not call `USBCtrlEPAllowStatusStage()`.

File

`usb_device.h`

Syntax

```
void USBCtrlEPAllowStatusStage( );
```

Remarks

None

Preconditions

None

Function

```
void USBCtrlEPAllowStatusStage(void);
```

1.4.1.1.5 `USBDeferINDataStage` Function

This function will cause the USB hardware to continuously NAK the IN token packets sent from the host, during the data stage of a device to host control transfer. This allows the firmware more time to process and prepare the IN data packets that will eventually be sent to the host. This is also useful, if the firmware needs to process/prepare the IN data in a different context than what the `USBDeviceTasks()` function executes at.

Calling this function (macro) will assert ownership of the currently pending control transfer. Therefore, the USB stack will not STALL when it reaches the data stage of the control transfer, even if the firmware has not (yet) called the `USBEP0SendRAMPtr()` or `USBEP0SendROMPtr()` API function. However, the application firmware must still (eventually, once it is ready) call one of the aforementioned API functions.

Example Usage:

1. Host sends a SETUP packet to the device, requesting a device to host control transfer, with data stage.
2. `USBDeviceTasks()` executes, and then calls the `USBCBCheckOtherReq()` callback event handler. The `USBCBCheckOtherReq()` calls the application specific/device class specific handler that detects the type of control transfer.
3. If the firmware needs more time to prepare the first IN data packet, or, if the firmware wishes to process the command in a different context (ex: if `USBDeviceTasks()` executes as an interrupt handler, but the IN data stage data needs to be prepared in the main loop context), then it may call `USBDeferINDataStage()`, in the context of the `USBCBCheckOtherReq()` handler function.
4. If the firmware called `USBDeferINDataStage()` in step #3 above, then the hardware will NAK the IN token packets sent by the host, for the IN data stage.
5. Once the firmware is ready, and has successfully prepared the data to be sent to the host in fulfillment of the control transfer, it should then call `USBEP0SendRAMPtr()` or `USBEP0SendROMPtr()`, to prepare the USB stack to know how many bytes to send to the host, and from what source location.
6. The firmware should now call `USBCtrlEPAllowDataStage()`. This will allow the data stage to complete. The USB stack will send the data buffer specified by the `USBEP0SendRAMPtr()` or `USBEP0SendROMPtr()` function, when it was called.
7. Once all data has been sent to the host, or if the host performs early termination, the status stage (a 0-byte OUT packet) will complete automatically (assuming the firmware did not call `USBDeferStatusStage()` during step #3).

File

`usb_device.h`

Syntax

```
void USBDeferINDataStage();
```

Remarks

Section 9.2.6 of the official USB 2.0 specifications indicates that the USB device must return the first IN data packet within 500ms of the start of the control transfer. In order to meet this specification, the firmware must call USBEP0SendRAMPtr() or USBEP0SendROMPtr(), and then call USBCtrlEPAllowDataStage(), in less than 500ms from the start of the control transfer.

If the firmware calls USBDeferINDataStage(), it must eventually call USBEP0SendRAMPtr() or USBEP0SendROMPtr(), and then call USBCtrlEPAllowDataStage(). If it does not do this, the control transfer will never be able to complete.

The firmware should never call both USBDeferINDataStage() and USBDeferOUTDataStage() during the same control transfer. These functions are mutually exclusive (a control transfer with data stage can never contain both IN and OUT data packets during the data stage).

Preconditions

Before calling USBDeferINDataStage(), the firmware should first verify that the control transfer has a data stage, and that it is of type device-to-host (IN).

Function

```
void USBDeferINDataStage(void);
```

1.4.1.1.6 USBDeferOUTDataStage Function

This function will cause the USB hardware to continuously NAK the OUT data packets sent from the host, during the data stage of a device to host control transfer. This allows the firmware more time to prepare the RAM buffer that will eventually be used to receive the data from the host. This is also useful, if the firmware wishes to receive the OUT data in a different context than what the USBDeviceTasks() function executes at.

Calling this function (macro) will assert ownership of the currently pending control transfer. Therefore, the USB stack will not STALL when it reaches the data stage of the control transfer, even if the firmware has not (yet) called the USBEP0Receive() API function. However, the application firmware must still (eventually, once it is ready) call one of the aforementioned API function.

Example Usage:

1. Host sends a SETUP packet to the device, requesting a host to device control transfer, with data stage (OUT data packets).
2. USBDeviceTasks() executes, and then calls the USBCBCheckOtherReq() callback event handler. The USBCBCheckOtherReq() calls the application specific/device class specific handler that detects the type of control transfer.
3. If the firmware needs more time before it wishes to receive the first OUT data packet, or, if the firmware wishes to process the command in a different context, then it may call USBDeferOUTDataStage(), in the context of the USBCBCheckOtherReq() handler function.
4. If the firmware called USBDeferOUTDataStage() in step #3 above, then the hardware will NAK the OUT data packets sent by the host, for the OUT data stage.
5. Once the firmware is ready, it should then call USBEP0Receive(), to prepare the USB stack to receive the OUT data from the host, and to write it to the user specified buffer.
6. The firmware should now call USBCtrlEPAllowDataStage(). This will allow the data stage to complete. Once all OUT data has been received, the user callback function (provided by the function pointer provided when calling USBEP0Receive()) will get called.
7. Once all data has been received from the host, the status stage (a 0-byte IN packet) will complete automatically (assuming the firmware did not call USBDeferStatusStage() during step #3).

File

usb_device.h

Syntax

```
void USBDeferOUTDataStage();
```

Remarks

Section 9.2.6 of the official USB 2.0 specifications indicates that the USB device must be able to receive all bytes and complete the control transfer within a maximum of 5 seconds.

If the firmware calls `USBDeferOUTDataStage()`, it must eventually call `USBEP0Receive()`, and then call `USBCtrlEPAllowDataStage()`. If it does not do this, the control transfer will never be able to complete. This will break the USB connection, as the host needs to be able to communicate over EP0, in order to perform basic tasks including enumeration.

The firmware should never call both `USBDeferINDataStage()` and `USBDeferOUTDataStage()` during the same control transfer. These functions are mutually exclusive (a control transfer with data stage can never contain both IN and OUT data packets during the data stage).

Preconditions

Before calling `USBDeferOUTDataStage()`, the firmware should first verify that the control transfer has a data stage, and that it is of type host-to-device (OUT).

Function

```
void USBDeferOUTDataStage(void);
```

1.4.1.1.7 USBDeferStatusStage Function

Calling this function will prevent the USB stack from automatically enabling the status stage for the currently pending control transfer from completing immediately after all data bytes have been sent or received. This is useful if a class handler or USB application firmware project uses control transfers for sending/receiving data over EP0, but requires time in order to finish processing and/or to consume the data.

For example: Consider an application which receives OUT data from the USB host, through EP0 using control transfers. Now assume that this application wishes to do something time consuming with this data (ex: transmit it to and save it to an external EEPROM device, connected via SPI/I2C/etc.). In this case, it would typically be desirable to defer allowing the USB status stage of the control transfer to complete, until after the data has been fully sent to the EEPROM device and saved.

If the USB class handler firmware that processes the control transfer SETUP packet determines that it will need extra time to complete the control transfer, it may optionally call `USBDeferStatusStage()`. If it does so, it is then the responsibility of the application firmware to eventually call `USBCtrlEPAllowStatusStage()`, once the firmware has finished processing the data associated with the control transfer.

If the firmware calls `USBDeferStatusStage()`, but never calls `USBCtrlEPAllowStatusStage()`, then one of two possibilities will occur.

1. If the "USB_ENABLE_STATUS_STAGE_TIMEOUTS" option is commented in `usb_config.h`, then the status stage of the control transfer will never be able to complete. This is an error case and should be avoided.
2. If the "USB_ENABLE_STATUS_STAGE_TIMEOUTS" option is enabled in `usb_config.h`, then the `USBDeviceTasks()` function will automatically call `USBCtrlEPAllowStatusStage()`, after the "USB_STATUS_STAGE_TIMEOUT" has elapsed, since the last quanta of "progress" has occurred in the control transfer. Progress is defined as the last successful transaction completing on EP0 IN or EP0 OUT. Although the timeouts feature allows the status stage to [eventually] complete, it is still preferable to manually call `USBCtrlEPAllowStatusStage()` after the application firmware has finished processing/consuming the control transfer data, as this will allow for much faster processing of control transfers, and therefore much higher data rates and better user responsiveness.

File

usb_device.h

Syntax`void USBDeferStatusStage();`**Remarks**

If this function is called, it should get called after the SETUP packet has arrived (the control transfer has started), but before the USBCtrlEPServiceComplete() function has been called by the USB stack. Therefore, the normal place to call USBDeferStatusStage() would be from within the USBCBCheckOtherReq() handler context. For example, in a HID application using control transfers, the USBDeferStatusStage() function would be called from within the USER_GET_REPORT_HANDLER or USER_SET_REPORT_HANDLER functions.

Preconditions

None

Function`void USBDeferStatusStage(void);`

1.4.1.1.8 USBDeviceAttach Function

Checks if VBUS is present, and that the USB module is not already initialized, and if so, enables the USB module so as to signal device attachment to the USB host.

File

usb_device.h

Syntax`void USBDeviceAttach();`**Description**

This function indicates to the USB host that the USB device has been attached to the bus. This function needs to be called in order for the device to start to enumerate on the bus.

Remarks

See also the USBDeviceDetach() API function documentation.

Preconditions

Should only be called when USB_INTERRUPT is defined. Also, should only be called from the main() loop context. Do not call USBDeviceAttach() from within an interrupt handler, as the USBDeviceAttach() function may modify global interrupt enable bits and settings.

For normal USB devices: Make sure that if the module was previously on, that it has been turned off for a long time (ex: 100ms+) before calling this function to re-enable the module. If the device turns off the D+ (for full speed) or D- (for low speed) ~1.5k ohm pull up resistor, and then turns it back on very quickly, common hosts will sometimes reject this event, since no human could ever unplug and re-attach a USB device in a microseconds (or nanoseconds) timescale. The host could simply treat this as some kind of glitch and ignore the event altogether.

Function`void USBDeviceAttach(void)`

1.4.1.1.9 USBDeviceDetach Function

This function configures the USB module to "soft detach" itself from the USB host.

File

usb_device.h

Syntax

```
void USBDeviceDetach();
```

Description

This function configures the USB module to perform a "soft detach" operation, by disabling the D+ (or D-) ~1.5k pull up resistor, which lets the host know the device is present and attached. This will make the host think that the device has been unplugged. This is potentially useful, as it allows the USB device to force the host to re-enumerate the device (on the firmware has re-enabled the USB module/pull up, by calling USBDeviceAttach(), to "soft re-attach" to the host).

Remarks

If the application firmware calls USBDeviceDetach(), it is strongly recommended that the firmware wait at least >= 80ms before calling USBDeviceAttach(). If the firmware performs a soft detach, and then re-attaches too soon (ex: after a few micro seconds for instance), some hosts may interpret this as an unexpected "glitch" rather than as a physical removal/re-attachment of the USB device. In this case the host may simply ignore the event without re-enumerating the device. To ensure that the host properly detects and processes the device soft detach/re-attach, it is recommended to make sure the device remains detached long enough to mimic a real human controlled USB unplug/re-attach event (ex: after calling USBDeviceDetach(), do not call USBDeviceAttach() for at least 80+ms, preferably longer).

Neither the USBDeviceDetach() or USBDeviceAttach() functions are blocking or take long to execute. It is the application firmwares responsibility for adding the 80+ms delay, when using these API functions.

The Windows plug and play event handler processing is fairly slow, especially in certain versions of Windows, and for certain USB device classes. It has been observed that some device classes need to provide even more USB detach dwell interval (before calling USBDeviceAttach()), in order to work correctly after re-enumeration. If the USB device is a CDC class device, it is recommended to wait at least 1.5 seconds or longer, before soft re-attaching to the host, to provide the plug and play event handler enough time to finish processing the removal event, before the re-attach occurs.

If the application is using the USB_POLLING mode option, then the USBDeviceDetach() and USBDeviceAttach() functions are not available. In this mode, the USB stack relies on the "#define USE_USB_BUS_SENSE_IO" and "#define USB_BUS_SENSE" options in the HardwareProfile *i>% [platform name].h file.*

When using the USB_POLLING mode option, and the "#define USE_USB_BUS_SENSE_IO" definition has been commented out, then the USB stack assumes that it should always enable the USB module at pretty much all times. Basically, anytime the application firmware calls USBDeviceTasks(), the firmware will automatically enable the USB module. This mode would typically be selected if the application was designed to be a purely bus powered device. In this case, the application is powered from the +5V VBUS supply from the USB port, so it is correct and sensible in this type of application to power up and turn on the USB module, at anytime that the microcontroller is powered (which implies the USB cable is attached and the host is also powered).

In a self powered application, the USB stack is designed with the intention that the user will enable the "#define USE_USB_BUS_SENSE_IO" option in the HardwareProfile *i>% [platform name].h file.* When this option is defined, then the USBDeviceTasks() function will automatically check the I/O pin port value of the designated pin (based on the #define USB_BUS_SENSE option in the HardwareProfile *i>% [platform name].h file*), every time the application calls USBDeviceTasks(). If the USBDeviceTasks() function is executed and finds that the pin defined by the #define USB_BUS_SENSE is in a logic low state, then it will automatically disable the USB module and tri-state the D+ and D- pins. If however the USBDeviceTasks() function is executed and finds the pin defined by the #define USB_BUS_SENSE is in a logic high state, then it will automatically enable the USB module, if it has not already been enabled.

Preconditions

Should only be called when USB_INTERRUPT is defined. See remarks section if USB_POLLING mode option is being used (*usb_config.h* option).

Additionally, this function should only be called from the main() loop context. Do not call this function from within an interrupt handler, as this function may modify global interrupt enable bits and settings.

Function

```
void USBDeviceDetach(void)
```

1.4.1.1.10 USBDeviceInit Function**File**

usb_device.h

Syntax

```
void USBDeviceInit();
```

Description

This function initializes the device stack it in the default state. The USB module will be completely reset including all of the internal variables, registers, and interrupt flags.

Remarks

None

Preconditions

This function must be called before any of the other USB Device functions can be called, including USBDeviceTasks().

Function

```
void USBDeviceInit(void)
```

1.4.1.1.11 USBDeviceTasks Function

This function is the main state machine/transaction handler of the USB device side stack. When the USB stack is operated in "USB_POLLING" mode (usb_config.h user option) the USBDeviceTasks() function should be called periodically to receive and transmit packets through the stack. This function also takes care of control transfers associated with the USB enumeration process, and detecting various USB events (such as suspend). This function should be called at least once every 1.8ms during the USB enumeration process. After the enumeration process is complete (which can be determined when USBGetDeviceState() returns CONFIGURED_STATE), the USBDeviceTasks() handler may be called the faster of: either once every 9.8ms, or as often as needed to make sure that the hardware USTAT FIFO never gets full. A good rule of thumb is to call USBDeviceTasks() at a minimum rate of either the frequency that USBTransferOnePacket() gets called, or, once/1.8ms, whichever is faster. See the inline code comments near the top of usb_device.c for more details about minimum timing requirements when calling USBDeviceTasks().

When the USB stack is operated in "USB_INTERRUPT" mode, it is not necessary to call USBDeviceTasks() from the main loop context. In the USB_INTERRUPT mode, the USBDeviceTasks() handler only needs to execute when a USB interrupt occurs, and therefore only needs to be called from the interrupt context.

File

usb_device.h

Syntax

```
void USBDeviceTasks();
```

Description

This function is the main state machine/transaction handler of the USB device side stack. When the USB stack is operated in "USB_POLLING" mode (usb_config.h user option) the USBDeviceTasks() function should be called periodically to receive and transmit packets through the stack. This function also takes care of control transfers associated with the USB enumeration process, and detecting various USB events (such as suspend). This function should be called at least once every 1.8ms during the USB enumeration process. After the enumeration process is complete (which can be determined when USBGetDeviceState() returns CONFIGURED_STATE), the USBDeviceTasks() handler may be called the faster of:

either once every 9.8ms, or as often as needed to make sure that the hardware USTAT FIFO never gets full. A good rule of thumb is to call USBDeviceTasks() at a minimum rate of either the frequency that USBTransferOnePacket() gets called, or, once/1.8ms, whichever is faster. See the inline code comments near the top of usb_device.c for more details about minimum timing requirements when calling USBDeviceTasks().

When the USB stack is operated in "USB_INTERRUPT" mode, it is not necessary to call USBDeviceTasks() from the main loop context. In the USB_INTERRUPT mode, the USBDeviceTasks() handler only needs to execute when a USB interrupt occurs, and therefore only needs to be called from the interrupt context.

Typical usage:

```
void main(void)
{
    USBDeviceInit();
    while(1)
    {
        USBDeviceTasks(); //Takes care of enumeration and other USB events
        if((USBGetDeviceState() < CONFIGURED_STATE) ||
           (USBIsDeviceSuspended() == true))
        {
            //Either the device is not configured or we are suspended,
            // so we don't want to execute any USB related application code
            continue; //go back to the top of the while loop
        }
        else
        {
            //Otherwise we are free to run USB and non-USB related user
            //application code.
            UserApplication();
        }
    }
}
```

Remarks

USBDeviceTasks() does not need to be called while in the USB suspend mode, if the user application firmware in the USBCBSuspend() callback function enables the ACTVIF USB interrupt source and put the microcontroller into sleep mode. If the application firmware decides not to sleep the microcontroller core during USB suspend (ex: continues running at full frequency, or clock switches to a lower frequency), then the USBDeviceTasks() function must still be called periodically, at a rate frequent enough to ensure the 10ms resume recovery interval USB specification is met. Assuming a worst case primary oscillator and PLL start up time of <5ms, then USBDeviceTasks() should be called once every 5ms in this scenario.

When the USB cable is detached, or the USB host is not actively powering the VBUS line to +5V nominal, the application firmware does not always have to call USBDeviceTasks() frequently, as no USB activity will be taking place. However, if USBDeviceTasks() is not called regularly, some alternative means of promptly detecting when VBUS is powered (indicating host attachment), or not powered (host powered down or USB cable unplugged) is still needed. For self or dual self/bus powered USB applications, see the USBDeviceAttach() and USBDeviceDetach() API documentation for additional considerations.

Preconditions

Make sure the USBDeviceInit() function has been called prior to calling USBDeviceTasks() for the first time.

Function

void USBDeviceTasks(void)

1.4.1.1.1.12 USBEnableEndpoint Function

This function will enable the specified endpoint with the specified options

File

usb_device.h

Syntax

```
void USBEEnableEndpoint(uint8_t ep, uint8_t options);
```

Returns

None

Description

This function will enable the specified endpoint with the specified options.

Typical Usage:

```
void USBCBInitEP(void)
{
    USBEEnableEndpoint(MSD_DATA_IN_EP,USB_IN_ENABLED|USB_OUT_ENABLED|USB_HANDSHAKE_ENABLED|US
B_DISALLOW_SETUP);
    USBMSDInit();
}
```

In the above example endpoint number MSD_DATA_IN_EP is being configured for both IN and OUT traffic with handshaking enabled. Also since MSD_DATA_IN_EP is not endpoint 0 (MSD does not allow this), then we can explicitly disable SETUP packets on this endpoint.

Remarks

None

Preconditions

None

Function

```
void USBEEnableEndpoint(uint8_t ep, uint8_t options)
```

1.4.1.1.1.13 USBEPOReceive Function

Sets the destination, size, and a function to call on the completion of the next control write.

File

usb_device.h

Syntax

```
void USBEPOReceive(uint8_t* dest, uint16_t size, void (*function));
```

Remarks

None

Preconditions

None

Parameters

Parameters	Description
uint8_t* dest	address of where the incoming data will go (make sure that this address is directly accessible by the USB module for parts with dedicated USB RAM this address must be in that space)
uint16_t size	the size of the data being received (is almost always going to be presented by the preceeding setup packet SetupPkt.wLength) (*function) - a function that you want called once the data is received. If this is specified as NULL then no function is called.

Function

```
void USBEP0Receive(uint8_t* dest, uint16_t size, void (*function))
```

1.4.1.1.14 USBEP0SendRAMPtr Function

Sets the source, size, and options of the data you wish to send from a RAM source

File

usb_device.h

Syntax

```
void USBEP0SendRAMPtr(uint8_t* src, uint16_t size, uint8_t Options);
```

Remarks

None

Preconditions

None

Parameters

Parameters	Description
uint8_t* src	address of the data to send
uint16_t size	the size of the data needing to be transmitted
options	<p>the various options that you want when sending the control data. Options are:</p> <ul style="list-style-type: none"> • USB_EP0_ROM • USB_EP0_RAM • USB_EP0_BUSY • USB_EP0_INCLUDE_ZERO • USB_EP0_NO_DATA • USB_EP0_NO_OPTIONS

Function

```
void USBEP0SendRAMPtr(uint8_t* src, uint16_t size, uint8_t Options)
```

1.4.1.1.15 USBEP0SendROMPtr Function

Sets the source, size, and options of the data you wish to send from a const source

File

usb_device.h

Syntax

```
void USBEP0SendROMPtr(uint8_t* src, uint16_t size, uint8_t Options);
```

Remarks

None

Preconditions

None

Parameters

Parameters	Description
uint8_t* src	address of the data to send
uint16_t size	the size of the data needing to be transmitted
options	<p>the various options that you want when sending the control data. Options are:</p> <ul style="list-style-type: none"> • USB_EP0_ROM • USB_EP0_RAM • USB_EP0_BUSY • USB_EP0_INCLUDE_ZERO • USB_EP0_NO_DATA • USB_EP0_NO_OPTIONS

Function

```
void USBEP0SendROMPtr(uint8_t* src, uint16_t size, uint8_t Options)
```

1.4.1.1.1.16 USBEP0Transmit Function

Sets the address of the data to send over the control endpoint

File

usb_device.h

Syntax

```
void USBEP0Transmit(uint8_t options);
```

Remarks

None

Preconditions

None

Parameters

Parameters	Description
uint8_t options	the various options that you want when sending the control data. Options are: USB_EP0_ROM USB_EP0_RAM USB_EP0_BUSY USB_EP0_INCLUDE_ZERO USB_EP0_NO_DATA USB_EP0_NO_OPTIONS

Function

```
void USBEP0Transmit(uint8_t options)
```

1.4.1.1.1.17 USBGet1msTickCount Function**File**

usb_device.h

Syntax

```
uint32_t USBGet1msTickCount();
```

Description

This function retrieves a 32-bit unsigned integer that normally increments by one every one millisecond. The count value starts from zero when the `USBDeviceInit()` function is first called. See the remarks section for details on special circumstances where the tick count will not increment.

Remarks

On 8-bit USB full speed devices, the internal counter is incremented on every SOF packet detected. Therefore, it will not increment during suspend or when the USB cable is detached. However, on 16-bit devices, the T1MSECIF hardware interrupt source is used to increment the internal counter. Therefore, on 16-bit devices, the count continues to increment during USB suspend or detach events, so long as the application code has not put the microcontroller to sleep during these events, and the application firmware is regularly calling the `USBDeviceTasks()` function (or allowing it to execute, if using `USB_INTERRUPT` mode operation).

In USB low speed applications, the host does not broadcast SOF packets to the device, so the application firmware becomes responsible for calling `USBIncrement1msInternalTimers()` periodically (ex: from a general purpose timer interrupt handler), or else the returned value from this function will not increment.

Prior to calling `USBDeviceInit()` for the first time the returned value will be unpredictable.

This function is `USB_INTERRUPT` mode safe and may be called from main loop code without risk of retrieving a partially updated 32-bit number.

However, this value only increments when the `USBDeviceTasks()` function is allowed to execute. If `USB_INTERRUPT` mode is used, it is allowable to block on this function. If however `USB_POLLING` mode is used, one must not block on this function without also calling `USBDeviceTasks()` continuously for the blocking duration (since the USB stack must still be allowed to execute, and the USB stack is also responsible for updating the tick counter internally).

If the application is operating in `USB_POLLING` mode, this function should only be called from the main loop context, and not from an interrupt handler, as the returned value could be incorrect, if the main loop context code was in the process of updating the internal count at the moment of the interrupt event.

Preconditions

This function should be called only after `USBDeviceInit()` has been called (at least once at the start of the application).

Function

```
uint32_t USBGet1msTickCount(void)
```

1.4.1.1.18 `USBGetTicksSinceSuspendEnd` Function

File

`usb_device.h`

Syntax

```
uint8_t USBGetTicksSinceSuspendEnd();
```

Description

This function retrieves a 8-bit unsigned byte that represents the number of milliseconds that have elapsed since the end of a USB suspend event. The value saturates at 255.

Remarks

This function does not increment during USB suspend conditions, or when the USB cable is detached from the host. Prior to calling `USBDeviceInit()` for the first time the returned value will be unpredictable.

This function is `USB_INTERRUPT` mode safe and may be called from main loop code without risk of retrieving a partially updated 32-bit number.

However, this value only increments when the `USBDeviceTasks()` function is allowed to execute. If `USB_INTERRUPT` mode is used, it is allowable to block on this function. If however `USB_POLLING` mode is used, one must not block on this function without also calling `USBDeviceTasks()` continuously for the blocking duration (since the USB stack must still be allowed to

execute, and the USB stack is also responsible for updating the tick counter internally).

Preconditions

This function should be called only after USBDeviceInit() has been called (at least once at the start of the application).

Function

```
uint8_t USBGetTicksSinceSuspendEnd(void);
```

1.4.1.1.19 USBGetDeviceState Function

This function will return the current state of the device on the USB. This function should return CONFIGURED_STATE before an application tries to send information on the bus.

File

usb_device.h

Syntax

```
USB_DEVICE_STATE USBGetDeviceState();
```

Description

This function returns the current state of the device on the USB. This function is used to determine when the device is ready to communicate on the bus. Applications should not try to send or receive data until this function returns CONFIGURED_STATE.

It is also important that applications yield as much time as possible to the USBDeviceTasks() function as possible while this function returns any value between ATTACHED_STATE through CONFIGURED_STATE.

For more information about the various device states, please refer to the USB specification section 9.1 available from www.usb.org.

Typical usage:

```
void main(void)
{
    USBDeviceInit()
    while(1)
    {
        USBDeviceTasks();
        if((USBGetDeviceState() < CONFIGURED_STATE) ||
           (USBIsDeviceSuspended() == true))
        {
            //Either the device is not configured or we are suspended
            // so we don't want to do execute any application code
            continue; //go back to the top of the while loop
        }
        else
        {
            //Otherwise we are free to run user application code.
            UserApplication();
        }
    }
}
```

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_DEVICE_STATE	the current state of the device on the bus

Function

```
USB_DEVICE_STATE USBGetDeviceState(void)
```

1.4.1.1.1.20 USBGetNextHandle Function

Retrieves the handle to the next endpoint BDT entry that the USBTransferOnePacket() will use.

File

```
usb_device.h
```

Syntax

```
USB_HANDLE USBGetNextHandle(uint8_t ep_num, uint8_t ep_dir);
```

Description

Retrieves the handle to the next endpoint BDT that the USBTransferOnePacket() will use. Useful for initialization and when ping pong buffering will be used on application endpoints.

Remarks

This API is useful for initializing USB_HANDLES during initialization of the application firmware. It is also useful when ping-pong buffering is enabled, and the application firmware wishes to arm both the even and odd BDTs for an endpoint simultaneously. In this case, the application firmware for sending data to the host would typically be something like follows:

```
USB_HANDLE Handle1;
USB_HANDLE Handle2;
USB_HANDLE* pHandle = &Handle1;
uint8_t UserDataBuffer1[64];
uint8_t UserDataBuffer2[64];
uint8_t* pDataBuffer = &UserDataBuffer1[0];

//Add some code that loads UserDataBuffer1[] with useful data to send,
//using the pDataBuffer pointer, for example:
//for(i = 0; i < 64; i++)
//{
//    *pDataBuffer++ = [useful data value];
//}

//Check if the next USB endpoint BDT is available
if(!USBHandleBusy(USBGetNextHandle(ep_num, IN_TO_HOST)))
{
    //The endpoint is available. Send the data.
    *pHandle = USBTransferOnePacket(ep_num, ep_dir, pDataBuffer, bytecount);
    //Toggle the handle and buffer pointer for the next transaction
    if(pHandle == &Handle1)
    {
        pHandle = &Handle2;
        pDataBuffer = &UserDataBuffer2[0];
    }
    else
    {
        pHandle = &Handle1;
        pDataBuffer = &UserDataBuffer1[0];
    }
}

//The firmware can then load the next data buffer (in this case
//UserDataBuffer2)with useful data, and send it using the same
//process. For example:

//Add some code that loads UserDataBuffer2[] with useful data to send,
//using the pDataBuffer pointer, for example:
//for(i = 0; i < 64; i++)
//{
//    *pDataBuffer++ = [useful data value];
//}
```

```

//Check if the next USB endpoint BDT is available
if( !USBHandleBusy(USBGetNextHandle(ep_num, IN_TO_HOST) )
{
    //The endpoint is available. Send the data.
    *pHandle = USBTransferOnePacket(ep_num, ep_dir, pDataBuffer, bytecount);
    //Toggle the handle and buffer pointer for the next transaction
    if(pHandle == &Handle1)
    {
        pHandle = &Handle2;
        pDataBuffer = &UserDataBuffer2[0];
    }
    else
    {
        pHandle = &Handle1;
        pDataBuffer = &UserDataBuffer1[0];
    }
}

```

Preconditions

Will return NULL if the USB device has not yet been configured/the endpoint specified has not yet been initialized by USBEnableEndpoint().

Return Values

Return Values	Description
USB_HANDLE	Returns the USB_HANDLE (a pointer) to the BDT that will be used next time the USBTransferOnePacket() function is called, for the given ep_num and ep_dir

Function

USB_HANDLE USBGetNextHandle(uint8_t ep_num, uint8_t ep_dir)

1.4.1.1.1.21 USBGetRemoteWakeupStatus Function

This function indicates if remote wakeup has been enabled by the host. Devices that support remote wakeup should use this function to determine if it should send a remote wakeup.

File

usb_device.h

Syntax

```
bool USBGetRemoteWakeupStatus();
```

Description

This function indicates if remote wakeup has been enabled by the host. Devices that support remote wakeup should use this function to determine if it should send a remote wakeup.

If a device does not support remote wakeup (the Remote wakeup bit, bit 5, of the bmAttributes field of the Configuration descriptor is set to 1), then it should not send a remote wakeup command to the PC and this function is not of any use to the device. If a device does support remote wakeup then it should use this function as described below.

If this function returns false and the device is suspended, it should not issue a remote wakeup (resume).

If this function returns true and the device is suspended, it should issue a remote wakeup (resume).

A device can add remote wakeup support by having the _RWU symbol added in the configuration descriptor (located in the usb_descriptors.c file in the project). This done in the 8th byte of the configuration descriptor. For example:

```

const uint8_t configDescriptor1[]={
    0x09,                                // Size
    USB_DESCRIPTOR_CONFIGURATION,           // descriptor type
    DESC_CONFIG_WORD(0x0022),              // Total length
    1,                                     // Number of interfaces
    1,                                     // Index value of this cfg
    0,                                     // Configuration string index

```

```
_DEFAULT | _SELF | _RWU,  
50, // Attributes, see usb_device.h  
// Max power consumption in 2X mA( 100mA)  
  
//The rest of the configuration descriptor should follow
```

For more information about remote wakeup, see the following section of the USB v2.0 specification available at www.usb.org:

- Section 9.2.5.2
- Table 9-10
- Section 7.1.7.7
- Section 9.4.5

Remarks

None

Preconditions

None

Return Values

Return Values	Description
true	Remote Wakeup has been enabled by the host
false	Remote Wakeup is not currently enabled

Function

```
bool USBGetRemoteWakeupStatus(void)
```

1.4.1.1.1.22 USBGetSuspendState Function

This function indicates if the USB port that this device is attached to is currently suspended. When suspended, it will not be able to transfer data over the bus.

File

usb_device.h

Syntax

```
bool USBGetSuspendState();
```

Description

This function indicates if the USB port that this device is attached to is currently suspended. When suspended, it will not be able to transfer data over the bus. This function can be used by the application to skip over section of code that do not need to execute if the device is unable to send data over the bus. This function can also be used to help determine when it is legal to perform USB remote wakeup signaling, for devices supporting this feature.

Typical usage:

```
void main(void)
{
    USBDeviceInit()
    while(1)
    {
        USBDeviceTasks();
        if((USBGetDeviceState() < CONFIGURED_STATE) ||
           (USBGetSuspendState() == true))
        {
            //Either the device is not configured or we are suspended
            // so we don't want to do execute any application code
            continue; //go back to the top of the while loop
        }
        else
        {
            //Otherwise we are free to run user application code.
        }
    }
}
```

```

        UserApplication( );
    }
}
}

```

Remarks

This function is the same as USBIsBusSuspended().

Preconditions

None

Return Values

Return Values	Description
true	the USB port this device is attached to is suspended.
false	the USB port this device is attached to is not suspended.

Function

```
bool USBGetSuspendState(void)
```

1.4.1.1.1.23 USBHandleBusy Function

Checks to see if the input handle is busy

File

usb_device.h

Syntax

```
bool USBHandleBusy(USB_HANDLE handle);
```

Description

Checks to see if the input handle is busy

Typical Usage

```
//make sure that the last transfer isn't busy by checking the handle
if(!USBHandleBusy(USBGenericInHandle))
{
    //Send the data contained in the INPacket[] array out on
    // endpoint USBGEN_EP_NUM
    USBGenericInHandle = USBGenWrite(USBGEN_EP_NUM,(uint8_t*)&INPacket[0],sizeof(INPacket));
}
```

Remarks

None

Preconditions

None

Return Values

Return Values	Description
true	The specified handle is busy
false	The specified handle is free and available for a transfer

Function

```
bool USBHandleBusy( USB_HANDLE handle)
```

1.4.1.1.1.24 USBHandleGetAddr Function

Retrieves the address of the destination buffer of the input handle

File

usb_device.h

Syntax

```
uint16_t USBHandleGetAddr(USB_HANDLE);
```

Description

Retrieves the address of the destination buffer of the input handle

Remarks

None

Preconditions

None

Return Values

Return Values	Description
uint16_t	address of the current buffer that the input handle points to.

Function

```
uint16_t USBHandleGetAddr(USB_HANDLE)
```

1.4.1.1.1.25 USBHandleGetLength Function

Retrieves the length of the destination buffer of the input handle

File

usb_device.h

Syntax

```
uint16_t USBHandleGetLength(USB_HANDLE handle);
```

Description

Retrieves the length of the destination buffer of the input handle

Remarks

None

Preconditions

None

Return Values

Return Values	Description
uint16_t	length of the current buffer that the input handle points to. If the transfer is complete then this is the length of the data transmitted or the length of data actually received.

Function

```
uint16_t USBHandleGetLength(USB_HANDLE handle)
```

1.4.1.1.1.26 USBIncrement1msInternalTimers Function

File

usb_device.h

Syntax

```
void USBIncrement1msInternalTimers();
```

Description

This function increments internal 1ms time base counters, which are useful for application code (that can use a 1ms time base/counter), and for certain USB event timing specific purposes.

In USB full speed applications, the application code does not need to (and should not) explicitly call this function, as the USBDeviceTasks() function will automatically call this function whenever a 1ms time interval has elapsed (assuming the code is calling USBDeviceTasks() frequently enough in USB_POLLING mode, or that USB interrupts aren't being masked for more than 1ms at a time in USB_INTERRUPT mode).

In USB low speed applications, the application firmware is responsible for periodically calling this function at a ~1ms rate. This can be done using a general purpose microcontroller timer set to interrupt every 1ms for example. If the low speed application code does not call this function, the internal timers will not increment, and the USBGet1msTickCount() API function will not be available. Additionally, certain USB stack operations (like control transfer timeouts) may be unavailable.

Remarks

This function does not need to be called during USB suspend conditions, when the USB module/stack is disabled, or when the USB cable is detached from the host.

Preconditions

This function should be called only after USBDeviceInit() has been called (at least once at the start of the application). Ordinarily, application code should never call this function, unless it is a low speed USB device.

Function

```
void USBIncrement1msInternalTimers(void)
```

1.4.1.1.1.27 USBINDataStageDeferred Function

Returns true if a control transfer with IN data stage is pending, and the firmware has called USBDeferINDataStage(), but has not yet called USBCtrlEPAllowDataStage(). Returns false if a control transfer with IN data stage is either not pending, or the firmware did not call USBDeferINDataStage() at the start of the control transfer.

This function (macro) would typically be used in the case where the USBDeviceTasks() function executes in the interrupt context (ex: USB_INTERRUPT option selected in usb_config.h), but the firmware wishes to take care of handling the data stage of the control transfer in the main loop context.

In this scenario, typical usage would be:

1. Host starts a control transfer with IN data stage.
2. USBDeviceTasks() (in this scenario, interrupt context) executes.
3. USBDeviceTasks() calls USBCBCheckOtherReq(), which in turn determines that the control transfer is class specific, with IN data stage.
4. The user code in USBCBCheckOtherReq() (also in interrupt context, since it is called from USBDeviceTasks(), and therefore executes at the same priority/context) calls USBDeferINDataStage().

5. Meanwhile, in the main loop context, a polling handler may be periodically checking if(USBINDataStageDeferred() == true). Ordinarily, it would evaluate false, but when a control transfer becomes pending, and after the USBDeferINDataStage() macro has been called (ex: in the interrupt context), the if() statement will evaluate true. In this case, the main loop context can then take care of sending the data (when ready), by calling USBEPOSendRAMPtr() or USBEPOSendROMPtr() and USBCtrlEPAllowDataStage().

File

usb_device.h

Syntax

```
bool USBINDataStageDeferred( );
```

Function

bool USBINDataStageDeferred(void);

1.4.1.1.1.28 USBIsBusSuspended Function

This function indicates if the USB bus is in suspend mode.

File

usb_device.h

Syntax

```
bool USBIsBusSuspended( );
```

Returns

None

Description

This function indicates if the USB bus is in suspend mode. This function is typically used for checking if the conditions are consistent with performing a USB remote wakeup sequence.

Typical Usage:

```
if((USBIsBusSuspended() == true) && (USBGetRemoteWakeupStatus() == true))
{
    //Check if some stimulus occurred, which will be used as the wakeup source
    if(sw3 == 0)
    {
        USBCBSendResume(); //Send the remote wakeup signalling to the host
    }
}
// otherwise do some other application specific tasks
```

Remarks

The USBIsBusSuspended() function relies on the USBBusIsSuspended boolean variable, which gets updated by the USBDeviceTasks() function. Therefore, in order to be sure the return value is not "stale", it is suggested to make sure USBDeviceTasks() has executed recently (if using USB polling mode).

Preconditions

None

Function

bool USBIsBusSuspended(void);

1.4.1.1.1.29 USBIsDeviceSuspended Function

This function indicates if the USB module is in suspend mode.

File

usb_device.h

Syntax

```
bool USBIsDeviceSuspended();
```

Returns

None

Description

This function indicates if the USB module is in suspend mode. This function does NOT indicate that a suspend request has been received. It only reflects the state of the USB module.

Typical Usage:

```
if(USBIsDeviceSuspended() == true)
{
    return;
}
// otherwise do some application specific tasks
```

Remarks

None

Preconditions

None

Function

```
bool USBIsDeviceSuspended(void)
```

1.4.1.1.1.30 USBOUTDataStageDeferred Function

Returns true if a control transfer with OUT data stage is pending, and the firmware has called USBDeferOUTDataStage(), but has not yet called USBCtrlEPAllowDataStage(). Returns false if a control transfer with OUT data stage is either not pending, or the firmware did not call USBDeferOUTDataStage() at the start of the control transfer.

This function (macro) would typically be used in the case where the USBDeviceTasks() function executes in the interrupt context (ex: USB_INTERRUPT option selected in usb_config.h), but the firmware wishes to take care of handling the data stage of the control transfer in the main loop context.

In this scenario, typical usage would be:

1. Host starts a control transfer with OUT data stage.
2. USBDeviceTasks() (in this scenario, interrupt context) executes.
3. USBDeviceTasks() calls USBCBCheckOtherReq(), which in turn determines that the control transfer is class specific, with OUT data stage.
4. The user code in USBCBCheckOtherReq() (also in interrupt context, since it is called from USBDeviceTasks(), and therefore executes at the same priority/context) calls USBDeferOUTDataStage().

5. Meanwhile, in the main loop context, a polling handler may be periodically checking if(USBOUTDataStageDeferred() == true). Ordinarily, it would evaluate false, but when a control transfer becomes pending, and after the USBDeferOUTDataStage() macro has been called (ex: in the interrupt context), the if() statement will evaluate true. In this case, the main loop context can then take care of receiving the data, by calling USBEP0Receive() and USBCtrlEPAllowDataStage().

File

usb_device.h

Syntax

```
bool USBOUTDataStageDeferred();
```

Function

```
bool USBOUTDataStageDeferred(void);
```

1.4.1.1.1.31 USBRxOnePacket Function

Receives the specified data out the specified endpoint

File

```
usb_device.h
```

Syntax

```
USB_HANDLE USBRxOnePacket(uint8_t ep, uint8_t* data, uint16_t len);
```

Remarks

None

Preconditions

None

Parameters

Parameters	Description
uint8_t ep	The endpoint number you want to receive the data on.
uint8_t* data	Pointer to a user buffer where the data will go when it arrives from the host. Note: This RAM must be USB module accessible.
uint16_t len	The len parameter should always be set to the maximum endpoint packet size, specified in the USB descriptor for this endpoint. The host may send <= the number of bytes as the endpoint size in the endpoint descriptor. After the transaction is complete, the application firmware can call USBHandleGetLength() to determine how many bytes the host actually sent in the last transaction on this endpoint.

Return Values

Return Values	Description
USB_HANDLE	Returns a pointer to the BDT entry associated with the transaction. The firmware can check for completion of the transaction by using the USBHandleBusy() function, using the returned USB_HANDLE value.

Function

```
USB_HANDLE USBRxOnePacket(uint8_t ep, uint8_t* data, uint16_t len)
```

1.4.1.1.1.32 USBSoftDetach Function

This function performs a detach from the USB bus via software.

File

```
usb_device.h
```

Syntax

```
void USBSoftDetach();
```

Returns

None

Description

This function performs a detach from the USB bus via software.

Remarks

Caution should be used when detaching from the bus. Some PC drivers and programs may require additional time after a detach before a device can be reattached to the bus.

Preconditions

None

Function

```
void USBSoftDetach(void);
```

1.4.1.1.1.33 USBStallEndpoint Function

Configures the specified endpoint to send STALL to the host, the next time the host tries to access the endpoint.

File

usb_device.h

Syntax

```
void USBStallEndpoint(uint8_t ep, uint8_t dir);
```

Remarks

None

Preconditions

None

Function

```
void USBStallEndpoint(uint8_t ep, uint8_t dir)
```

1.4.1.1.1.34 USBTransferOnePacket Function

Transfers a single packet (one transaction) of data on the USB bus.

File

usb_device.h

Syntax

```
USB_HANDLE USBTransferOnePacket(uint8_t ep, uint8_t dir, uint8_t* data, uint8_t len);
```

Description

The USBTransferOnePacket() function prepares a USB endpoint so that it may send data to the host (an IN transaction), or receive data from the host (an OUT transaction). The USBTransferOnePacket() function can be used both to receive and send data to the host. This function is the primary API function provided by the USB stack firmware for sending or receiving application data over the USB port.

The USBTransferOnePacket() is intended for use with all application endpoints. It is not used for sending or receiving application data through endpoint 0 by using control transfers. Separate API functions, such as USBEPOReceive(), USBEPOSendRAMPtr(), and USBEPOSendROMPtr() are provided for this purpose.

The USBTransferOnePacket() writes to the Buffer Descriptor Table (BDT) entry associated with an endpoint buffer, and sets the UOWN bit, which prepares the USB hardware to allow the transaction to complete. The application firmware can use the USBHandleBusy() macro to check the status of the transaction, to see if the data has been successfully transmitted yet.

Typical Usage

```

//make sure that we are in the configured state
if(USBGetDeviceState() == CONFIGURED_STATE)
{
    //make sure that the last transaction isn't busy by checking the handle
    if( !USBHandleBusy(USBInHandle) )
    {
        //Write the new data that we wish to send to the host to the INPacket[] array
        INPacket[0] = USEFUL_APPLICATION_VALUE1;
        INPacket[1] = USEFUL_APPLICATION_VALUE2;
        //INPacket[2] = ... (fill in the rest of the packet data)

        //Send the data contained in the INPacket[] array through endpoint "EP_NUM"
        USBInHandle =
        USBTransferOnePacket(EP_NUM, IN_TO_HOST,(uint8_t*)&INPacket[0],sizeof(INPacket));
    }
}

```

Remarks

If calling the USBTransferOnePacket() function from within the USBCBInitEP() callback function, the set configuration is still being processed and the USBDeviceState may not be == CONFIGURED_STATE yet. In this special case, the USBTransferOnePacket() may still be called, but make sure that the endpoint has been enabled and initialized by the USBEnableEndpoint() function first.

Preconditions

Before calling USBTransferOnePacket(), the following should be true.

1. The USB stack has already been initialized (USBDeviceInit() was called).
2. A transaction is not already pending on the specified endpoint. This is done by checking the previous request using the USBHandleBusy() macro (see the typical usage example).
3. The host has already sent a set configuration request and the enumeration process is complete. This can be checked by verifying that the USBGetDeviceState() macro returns "CONFIGURED_STATE", prior to calling USBTransferOnePacket().

Return Values

Return Values	Description
USB_HANDLE	handle to the transfer. The handle is a pointer to the BDT entry associated with this transaction. The
status of the transaction (ex	if it is complete or still pending) can be checked using the USBHandleBusy() macro and supplying the USB_HANDLE provided by USBTransferOnePacket().

Function

USB_HANDLE USBTransferOnePacket(uint8_t ep, uint8_t dir, uint8_t* data, uint8_t len)

1.4.1.1.1.35 USBTxOnePacket Function

Sends the specified data out the specified endpoint

File

usb_device.h

Syntax

```
USB_HANDLE USBTxOnePacket(uint8_t ep, uint8_t* data, uint16_t len);
```

Remarks

None

Preconditions

None

Parameters

Parameters	Description
uint8_t ep	the endpoint number you want to send the data out of
uint8_t* data	pointer to a user buffer that contains the data that you wish to send to the host. Note: This RAM buffer must be accessible by the USB module.
uint16_t len	the number of bytes of data that you wish to send to the host, in the next transaction on this endpoint. Note: this value should always be less than or equal to the endpoint size, as specified in the USB endpoint descriptor.

Return Values

Return Values	Description
USB_HANDLE	Returns a pointer to the BDT entry associated with the transaction. The firmware can check for completion of the transaction by using the USBHandleBusy() function, using the returned USB_HANDLE value.

Function

```
USB_HANDLE USBTxOnePacket(uint8_t ep, uint8_t* data, uint16_t len)
```

1.4.1.1.2 Data Types and Constants**Macros**

Name	Description
DESC_CONFIG_uint32_t	The DESC_CONFIG_uint32_t() macro is implemented for convenience. Since the configuration descriptor array is a uint8_t array, each entry needs to be a uint8_t in LSB format. The DESC_CONFIG_uint32_t() macro breaks up a uint32_t into the appropriate uint8_t entries in LSB.
DESC_CONFIG_uint8_t	The DESC_CONFIG_uint8_t() macro is implemented for convenience. The DESC_CONFIG_uint8_t() macro provides a consistent macro for use with a byte when generating a configuration descriptor when using either the DESC_CONFIG_WORD() or DESC_CONFIG_uint32_t() macros.
DESC_CONFIG_WORD	The DESC_CONFIG_WORD() macro is implemented for convenience. Since the configuration descriptor array is a uint8_t array, each entry needs to be a uint8_t in LSB format. The DESC_CONFIG_WORD() macro breaks up a uint16_t into the appropriate uint8_t entries in LSB. Typical Usage:
USB_EP0_BUSY	The PIPE is busy
USB_EP0_INCLUDE_ZERO	include a trailing zero packet
USB_EP0_NO_DATA	no data to send
USB_EP0_NO_OPTIONS	no options set
USB_EP0_RAM	Data comes from const
USB_EP0_ROM	Data comes from RAM
USB_HANDLE	USB_HANDLE is a pointer to an entry in the BDT. This pointer can be used to read the length of the last transfer, the status of the last transfer, and various other information. Insure to initialize USB_HANDLE objects to NULL so that they are in a known state during their first usage.

Module

Device Stack

Types

Name	Description
USB_DEVICE_STACK_EVENTS	USB device stack events description here - DWF
USB_DEVICE_STATE	USB Device States as returned by USBGetDeviceState(). Only the definitions for these states should be used. The actual value for each state should not be relied upon as constant and may change based on the implementation.

Description**1.4.1.1.2.1 USB_DEVICE_STACK_EVENTS Type****File**

usb_device.h

Syntax

```
typedef enum USB_DEVICE_STACK_EVENTS@1 USB_DEVICE_STACK_EVENTS;
```

Description

USB device stack events description here - DWF

1.4.1.1.2.2 USB_DEVICE_STATE Type**File**

usb_device.h

Syntax

```
typedef enum USB_DEVICE_STATE@1 USB_DEVICE_STATE;
```

Description

USB Device States as returned by USBGetDeviceState(). Only the definitions for these states should be used. The actual value for each state should not be relied upon as constant and may change based on the implementation.

1.4.1.1.2.3 DESC_CONFIG_uint32_t Macro**File**

usb_device.h

Syntax

```
#define DESC_CONFIG_uint32_t(a) ((a&0xFF),((a>>8)&0xFF),((a>>16)&0xFF),((a>>24)&0xFF))
```

Description

The DESC_CONFIG_uint32_t() macro is implemented for convenience. Since the configuration descriptor array is a uint8_t array, each entry needs to be a uint8_t in LSB format. The DESC_CONFIG_uint32_t() macro breaks up a uint32_t into the appropriate uint8_t entries in LSB.

1.4.1.1.2.4 DESC_CONFIG_uint8_t Macro**File**

usb_device.h

Syntax

```
#define DESC_CONFIG_uint8_t(a) (a)
```

Description

The DESC_CONFIG_uint8_t() macro is implemented for convenience. The DESC_CONFIG_uint8_t() macro provides a consistent macro for use with a byte when generating a configuration descriptor when using either the DESC_CONFIG_WORD() or DESC_CONFIG_uint32_t() macros.

1.4.1.1.2.5 DESC_CONFIG_WORD Macro**File**

usb_device.h

Syntax

```
#define DESC_CONFIG_WORD(a) (a&0xFF), ((a>>8)&0xFF)
```

Description

The DESC_CONFIG_WORD() macro is implemented for convenience. Since the configuration descriptor array is a uint8_t array, each entry needs to be a uint8_t in LSB format. The DESC_CONFIG_WORD() macro breaks up a uint16_t into the appropriate uint8_t entries in LSB. Typical Usage:

```
const uint8_t configDescriptor1[]={
    0x09,           // Size of this descriptor in bytes
    USB_DESCRIPTOR_CONFIGURATION, // CONFIGURATION descriptor type
    DESC_CONFIG_WORD(0x0022),      // Total length of data for this cfg
```

1.4.1.1.2.6 USB_EP0_BUSY Macro**File**

usb_device.h

Syntax

```
#define USB_EP0_BUSY 0x80      //The PIPE is busy
```

Description

The PIPE is busy

1.4.1.1.2.7 USB_EP0_INCLUDE_ZERO Macro**File**

usb_device.h

Syntax

```
#define USB_EP0_INCLUDE_ZERO 0x40      //include a trailing zero packet
```

Description

include a trailing zero packet

1.4.1.1.2.8 USB_EP0_NO_DATA Macro**File**

usb_device.h

Syntax

```
#define USB_EP0_NO_DATA 0x00      //no data to send
```

Description

no data to send

1.4.1.1.2.9 USB_EP0_NO_OPTIONS Macro

File

usb_device.h

Syntax

```
#define USB_EP0_NO_OPTIONS 0x00      //no options set
```

Description

no options set

1.4.1.1.2.10 USB_EP0_RAM Macro

File

usb_device.h

Syntax

```
#define USB_EP0_RAM 0x01      //Data comes from const
```

Description

Data comes from const

1.4.1.1.2.11 USB_EP0_ROM Macro

File

usb_device.h

Syntax

```
#define USB_EP0_ROM 0x00      //Data comes from RAM
```

Description

Data comes from RAM

1.4.1.1.2.12 USB_HANDLE Macro

File

usb_device.h

Syntax

```
#define USB_HANDLE void*
```

Description

USB_HANDLE is a pointer to an entry in the BDT. This pointer can be used to read the length of the last transfer, the status of the last transfer, and various other information. Insure to initialize USB_HANDLE objects to NULL so that they are in a known state during their first usage.

1.4.1.1.3 _USB_DEVICE_H Macro

Defines types and APIs associated with the USB device stack.

File

usb_device.h

Syntax

```
#define _USB_DEVICE_H
```

Module

Device Stack

Description

Module for Microchip USB Library

Defines types and APIs associated with the USB device stack.

File Name

usb_device.h

Company

Microchip Technology Inc.

1.4.1.1.4 usb_device.h**Functions**

	Name	Description
💡	USB_APPLICATION_EVENT_HANDLER	This function is called whenever the USB stack wants to notify the user of an event.
💡	USBCancelIO	This function cancels the transfers pending on the specified endpoint. This function can only be used after a SETUP packet is received and before that setup packet is handled. This is the time period in which the EVENT_EP0_REQUEST is thrown, before the event handler function returns to the stack.
💡	USBCtrlEPAllowDataStage	This function allows the data stage of either a host-to-device or device-to-host control transfer (with data stage) to complete. This function is meant to be used in conjunction with either the USBDeferOUTDataStage() or USBDeferINDataStage(). If the firmware does not call either USBDeferOUTDataStage() or USBDeferINDataStage(), then the firmware does not need to manually call USBCtrlEPAllowDataStage(), as the USB stack will call this function instead.
💡	USBCtrlEPAllowStatusStage	This function prepares the proper endpoint 0 IN or endpoint 0 OUT (based on the controlTransferState) to allow the status stage packet of a control transfer to complete. This function gets used internally by the USB stack itself, but it may also be called from the application firmware, IF the application firmware called the USBDeferStatusStage() function during the initial processing of the control transfer request. In this case, the application must call the USBCtrlEPAllowStatusStage() once, after it has fully completed processing and handling the data stage portion of the request. If the application firmware has no need for delaying... more
💡	USBDeferINDataStage	This function will cause the USB hardware to continuously NAK the IN token packets sent from the host, during the data stage of a device to host control transfer. This allows the firmware more time to process and prepare the IN data packets that will eventually be sent to the host. This is also useful, if the firmware needs to process/prepare the IN data in a different context than what the USBDeviceTasks() function executes at. Calling this function (macro) will assert ownership of the currently pending control transfer. Therefore, the USB stack will not STALL when it reaches the... more

	USBDeferOUTDataStage	This function will cause the USB hardware to continuously NAK the OUT data packets sent from the host, during the data stage of a device to host control transfer. This allows the firmware more time to prepare the RAM buffer that will eventually be used to receive the data from the host. This is also useful, if the firmware wishes to receive the OUT data in a different context than what the USBDeviceTasks() function executes at. Calling this function (macro) will assert ownership of the currently pending control transfer. Therefore, the USB stack will not STALL when it reaches... more
	USBDeferStatusStage	Calling this function will prevent the USB stack from automatically enabling the status stage for the currently pending control transfer from completing immediately after all data bytes have been sent or received. This is useful if a class handler or USB application firmware project uses control transfers for sending/receiving data over EP0, but requires time in order to finish processing and/or to consume the data. For example: Consider an application which receives OUT data from the USB host, through EP0 using control transfers. Now assume that this application wishes to do something time consuming with this data (ex: transmit it... more
	USBDeviceAttach	Checks if VBUS is present, and that the USB module is not already initialized, and if so, enables the USB module so as to signal device attachment to the USB host.
	USBDeviceDetach	This function configures the USB module to "soft detach" itself from the USB host.
	USBDeviceInit	This function initializes the device stack it in the default state. The USB module will be completely reset including all of the internal variables, registers, and interrupt flags.
	USBDeviceTasks	This function is the main state machine/transaction handler of the USB device side stack. When the USB stack is operated in "USB_POLLING" mode (usb_config.h user option) the USBDeviceTasks() function should be called periodically to receive and transmit packets through the stack. This function also takes care of control transfers associated with the USB enumeration process, and detecting various USB events (such as suspend). This function should be called at least once every 1.8ms during the USB enumeration process. After the enumeration process is complete (which can be determined when USBGetDeviceState() returns CONFIGURED_STATE), the USBDeviceTasks() handler may be called the... more
	USBEnableEndpoint	This function will enable the specified endpoint with the specified options
	USBEP0Receive	Sets the destination, size, and a function to call on the completion of the next control write.
	USBEP0SendRAMPtr	Sets the source, size, and options of the data you wish to send from a RAM source
	USBEP0SendROMPtr	Sets the source, size, and options of the data you wish to send from a const source
	USBEP0Transmit	Sets the address of the data to send over the control endpoint
	USBGet1msTickCount	This function retrieves a 32-bit unsigned integer that normally increments by one every one millisecond. The count value starts from zero when the USBDeviceInit() function is first called. See the remarks section for details on special circumstances where the tick count will not increment.
	USBGetDeviceState	This function will return the current state of the device on the USB. This function should return CONFIGURED_STATE before an application tries to send information on the bus.
	USBGetNextHandle	Retrieves the handle to the next endpoint BDT entry that the USBTransferOnePacket() will use.

	USBGetRemoteWakeUpStatus	This function indicates if remote wakeup has been enabled by the host. Devices that support remote wakeup should use this function to determine if it should send a remote wakeup.
	USBGetSuspendState	This function indicates if the USB port that this device is attached to is currently suspended. When suspended, it will not be able to transfer data over the bus.
	USBGetTicksSinceSuspendEnd	This function retrieves a 8-bit unsigned byte that represents the number of milliseconds that have elapsed since the end of a USB suspend event. The value saturates at 255.
	USBHandleBusy	Checks to see if the input handle is busy
	USBHandleGetAddr	Retrieves the address of the destination buffer of the input handle
	USBHandleGetLength	Retrieves the length of the destination buffer of the input handle
	USBIncrement1msInternalTimers	<p>This function increments internal 1ms time base counters, which are useful for application code (that can use a 1ms time base/counter), and for certain USB event timing specific purposes.</p> <p>In USB full speed applications, the application code does not need to (and should not) explicitly call this function, as the <code>USBDeviceTasks()</code> function will automatically call this function whenever a 1ms time interval has elapsed (assuming the code is calling <code>USBDeviceTasks()</code> frequently enough in <code>USB_POLLING</code> mode, or that USB interrupts aren't being masked for more than 1ms at a time in <code>USB_INTERRUPT</code> mode).</p> <p>In USB low speed applications, the... more</p>
	USBINDataStageDeferred	Returns true if a control transfer with IN data stage is pending, and the firmware has called <code>USBDeferINDataStage()</code> , but has not yet called <code>USBCtrlIEPAllowDataStage()</code> . Returns false if a control transfer with IN data stage is either not pending, or the firmware did not call <code>USBDeferINDataStage()</code> at the start of the control transfer.
	USBINDataStageDeferred	This function (macro) would typically be used in the case where the <code>USBDeviceTasks()</code> function executes in the interrupt context (ex: <code>USB_INTERRUPT</code> option selected in <code>usb_config.h</code>), but the firmware wishes to take care of handling the data stage of the control transfer in the main... more
	USBIsBusSuspended	This function indicates if the USB bus is in suspend mode.
	USBIsDeviceSuspended	This function indicates if the USB module is in suspend mode.
	USBOUTDataStageDeferred	<p>Returns true if a control transfer with OUT data stage is pending, and the firmware has called <code>USBDeferOUTDataStage()</code>, but has not yet called <code>USBCtrlIEPAllowDataStage()</code>. Returns false if a control transfer with OUT data stage is either not pending, or the firmware did not call <code>USBDeferOUTDataStage()</code> at the start of the control transfer.</p> <p>This function (macro) would typically be used in the case where the <code>USBDeviceTasks()</code> function executes in the interrupt context (ex: <code>USB_INTERRUPT</code> option selected in <code>usb_config.h</code>), but the firmware wishes to take care of handling the data stage of the control transfer in the main... more</p>
	USBRxOnePacket	Receives the specified data out the specified endpoint
	USBSoftDetach	This function performs a detach from the USB bus via software.
	USBStallEndpoint	Configures the specified endpoint to send STALL to the host, the next time the host tries to access the endpoint.
	USBTransferOnePacket	Transfers a single packet (one transaction) of data on the USB bus.
	USBTxOnePacket	Sends the specified data out the specified endpoint

Macros

Name	Description
<code>_USB_DEVICE_H</code>	Defines types and APIs associated with the USB device stack.

DESC_CONFIG_uint32_t	The DESC_CONFIG_uint32_t() macro is implemented for convenience. Since the configuration descriptor array is a uint8_t array, each entry needs to be a uint8_t in LSB format. The DESC_CONFIG_uint32_t() macro breaks up a uint32_t into the appropriate uint8_t entries in LSB.
DESC_CONFIG_uint8_t	The DESC_CONFIG_uint8_t() macro is implemented for convenience. The DESC_CONFIG_uint8_t() macro provides a consistent macro for use with a byte when generating a configuration descriptor when using either the DESC_CONFIG_WORD() or DESC_CONFIG_uint32_t() macros.
DESC_CONFIG_WORD	The DESC_CONFIG_WORD() macro is implemented for convenience. Since the configuration descriptor array is a uint8_t array, each entry needs to be a uint8_t in LSB format. The DESC_CONFIG_WORD() macro breaks up a uint16_t into the appropriate uint8_t entries in LSB. Typical Usage:
USB_EP0_BUSY	The PIPE is busy
USB_EP0_INCLUDE_ZERO	include a trailing zero packet
USB_EP0_NO_DATA	no data to send
USB_EP0_NO_OPTIONS	no options set
USB_EP0_RAM	Data comes from const
USB_EP0_ROM	Data comes from RAM
USB_HANDLE	USB_HANDLE is a pointer to an entry in the BDT. This pointer can be used to read the length of the last transfer, the status of the last transfer, and various other information. Insure to initialize USB_HANDLE objects to NULL so that they are in a known state during their first usage.

Module

Device Stack

Types

Name	Description
USB_DEVICE_STACK_EVENTS	USB device stack events description here - DWF
USB_DEVICE_STATE	USB Device States as returned by USBGetDeviceState(). Only the definitions for these states should be used. The actual value for each state should not be relied upon as constant and may change based on the implementation.

Description

This is file usb_device.h.

1.4.1.2 Audio Function Driver

Files

Name	Description
usb_device_audio.h	This is file usb_device_audio.h.

Description

1.4.1.2.1 Functions

Functions

	Name	Description
♫	USBCheckAudioRequest	This routine checks the setup data packet to see if it knows how to handle it

Module

Audio Function Driver

Description**1.4.1.2.1.1 USBCheckAudioRequest Function**

This routine checks the setup data packet to see if it knows how to handle it

File

usb_device_audio.h

Syntax

```
void USBCheckAudioRequest();
```

Description

This routine checks the setup data packet to see if it knows how to handle it

Remarks

None

Preconditions

None

Function

```
void USBCheckAudioRequest(void)
```

1.4.1.2.2 usb_device_audio.h**Functions**

	Name	Description
	USBCheckAudioRequest	This routine checks the setup data packet to see if it knows how to handle it

Module

Audio Function Driver

Description

This is file usb_device_audio.h.

1.4.1.3 CDC Function Driver**Files**

Name	Description
usb_device_cdc.h	This is file usb_device_cdc.h.

Macros

Name	Description
CDC_H	This is macro CDC_H.

Description

1.4.1.3.1 usb_device_cdc.h

Functions

Name	Description
CDCInitEP	This function initializes the CDC function driver. This function should be called after the SET_CONFIGURATION command (ex: within the context of the USBCBInitEP() function).
CDCNotificationHandler	Checks for changes in DSR status and reports them to the USB host.
CDCTxService	CDCTxService handles device-to-host transaction(s). This function should be called once per Main Program loop after the device reaches the configured state.
getsUSBUSART	getsUSBUSART copies a string of BYTEs received through USB CDC Bulk OUT endpoint to a user's specified location. It is a non-blocking function. It does not wait for data if there is no data available. Instead it returns '0' to notify the caller that there is no data available.
putrsUSBUSART	putrsUSBUSART writes a string of data to the USB including the null character. Use this version, 'putrs', to transfer data literals and data located in program memory.
putsUSBUSART	putsUSBUSART writes a string of data to the USB including the null character. Use this version, 'puts', to transfer data from a RAM buffer.
putUSBUSART	putUSBUSART writes an array of data to the USB. Use this version, is capable of transferring 0x00 (what is typically a NULL character in any of the string transfer functions).
USBCDCEventHandler	Handles events from the USB stack, which may have an effect on the CDC endpoint(s).
USBCheckCDCRequest	This routine checks the most recently received SETUP data packet to see if the request is specific to the CDC class. If the request was a CDC specific request, this function will take care of handling the request and responding appropriately.

Macros

Name	Description
CDC_H	This is macro CDC_H.
CDCSetBaudRate	This macro is used set the baud rate reported back to the host during a get line coding request. (optional)
CDCSetCharacterFormat	This macro is used manually set the character format reported back to the host during a get line coding request. (optional)
CDCSetDataSize	This function is used manually set the number of data bits reported back to the host during a get line coding request. (optional)
CDCSetLineCoding	This function is used to manually set the data reported back to the host during a get line coding request. (optional)
CDCSetParity	This function is used manually set the parity format reported back to the host during a get line coding request. (optional)
mUSBUSARTIsTxTrfReady	Deprecated in MCHPFSUSB v2.3. This macro has been replaced by USBUSARTIsTxTrfReady().

mUSBUSARTTxRam	<p>Use this macro to transfer data located in data memory. Use this macro when:</p> <ol style="list-style-type: none"> 1. Data stream is not null-terminated 2. Transfer length is known <p>Remember: cdc_trf_state must == CDC_TX_READY Unlike putsUSBUSART, there is not code double checking the transfer state. Unexpected behavior will occur if this function is called when cdc_trf_state != CDC_TX_READY</p> <p>Typical Usage:</p>
mUSBUSARTTxRom	<p>Use this macro to transfer data located in program memory. Use this macro when:</p> <ol style="list-style-type: none"> 1. Data stream is not null-terminated 2. Transfer length is known <p>Remember: cdc_trf_state must == CDC_TX_READY Unlike putrsUSBUSART, there is not code double checking the transfer state. Unexpected behavior will occur if this function is called when cdc_trf_state != CDC_TX_READY</p> <p>Typical Usage:</p>
NUM_STOP_BITS_1	1 stop bit - used by CDCSetLineCoding() and CDCSetCharacterFormat()
NUM_STOP_BITS_1_5	1.5 stop bit - used by CDCSetLineCoding() and CDCSetCharacterFormat()
NUM_STOP_BITS_2	2 stop bit - used by CDCSetLineCoding() and CDCSetCharacterFormat()
PARITY_EVEN	even parity - used by CDCSetLineCoding() and CDCSetParity()
PARITY_MARK	mark parity - used by CDCSetLineCoding() and CDCSetParity()
PARITY_NONE	no parity - used by CDCSetLineCoding() and CDCSetParity()
PARITY_ODD	odd parity - used by CDCSetLineCoding() and CDCSetParity()
PARITY_SPACE	space parity - used by CDCSetLineCoding() and CDCSetParity()
USBUSARTIsTxTrfReady	This macro is used to check if the CDC class is ready to send more data.

Module

CDC Function Driver

Description

This is file usb_device_cdc.h.

1.4.1.3.2 Functions

Functions and macro functions used to interface with the CDC module.

Functions

	Name	Description
💡	CDCInitEP	This function initializes the CDC function driver. This function should be called after the SET_CONFIGURATION command (ex: within the context of the USBCBInitEP() function).
💡	CDCNotificationHandler	Checks for changes in DSR status and reports them to the USB host.
💡	CDCTxService	CDCTxService handles device-to-host transaction(s). This function should be called once per Main Program loop after the device reaches the configured state.

	getsUSBUSART	getsUSBUSART copies a string of BYTES received through USB CDC Bulk OUT endpoint to a user's specified location. It is a non-blocking function. It does not wait for data if there is no data available. Instead it returns '0' to notify the caller that there is no data available.
	putrsUSBUSART	putrsUSBUSART writes a string of data to the USB including the null character. Use this version, 'putrs', to transfer data literals and data located in program memory.
	putsUSBUSART	putsUSBUSART writes a string of data to the USB including the null character. Use this version, 'puts', to transfer data from a RAM buffer.
	putUSBUSART	putUSBUSART writes an array of data to the USB. Use this version, is capable of transferring 0x00 (what is typically a NULL character in any of the string transfer functions).
	USBCDCEventHandler	Handles events from the USB stack, which may have an effect on the CDC endpoint(s).
	USBCheckCDCRequest	This routine checks the most recently received SETUP data packet to see if the request is specific to the CDC class. If the request was a CDC specific request, this function will take care of handling the request and responding appropriately.

Macros

Name	Description
CDCSetBaudRate	This macro is used set the baud rate reported back to the host during a get line coding request. (optional)
CDCSetCharacterFormat	This macro is used manually set the character format reported back to the host during a get line coding request. (optional)
CDCSetDataSize	This function is used manually set the number of data bits reported back to the host during a get line coding request. (optional)
CDCSetLineCoding	This function is used to manually set the data reported back to the host during a get line coding request. (optional)
CDCSetParity	This function is used manually set the parity format reported back to the host during a get line coding request. (optional)
mUSBUSARTIsTxTrfReady	Deprecated in MCHPFSUSB v2.3. This macro has been replaced by USBUSARTIsTxTrfReady().
mUSBUSARTTxRam	<p>Use this macro to transfer data located in data memory. Use this macro when:</p> <ol style="list-style-type: none"> 1. Data stream is not null-terminated 2. Transfer length is known <p>Remember: cdc_trf_state must == CDC_TX_READY Unlike putsUSBUSART, there is not code double checking the transfer state. Unexpected behavior will occur if this function is called when cdc_trf_state != CDC_TX_READY</p> <p>Typical Usage:</p>
mUSBUSARTTxRom	<p>Use this macro to transfer data located in program memory. Use this macro when:</p> <ol style="list-style-type: none"> 1. Data stream is not null-terminated 2. Transfer length is known <p>Remember: cdc_trf_state must == CDC_TX_READY Unlike putrsUSBUSART, there is not code double checking the transfer state. Unexpected behavior will occur if this function is called when cdc_trf_state != CDC_TX_READY</p> <p>Typical Usage:</p>
USBUSARTIsTxTrfReady	This macro is used to check if the CDC class is ready to send more data.

Module

CDC Function Driver

Description

Functions and macro functions used to interface with the CDC module.

1.4.1.3.2.1 CDCInitEP Function

This function initializes the CDC function driver. This function should be called after the SET_CONFIGURATION command (ex: within the context of the USBCBInitEP() function).

File

usb_device_cdc.h

Syntax

```
void CDCInitEP();
```

Description

This function initializes the CDC function driver. This function sets the default line coding (baud rate, bit parity, number of data bits, and format). This function also enables the endpoints and prepares for the first transfer from the host.

This function should be called after the SET_CONFIGURATION command. This is most simply done by calling this function from the USBCBInitEP() function.

Typical Usage:

```
void USBCBInitEP(void)
{
    CDCInitEP();
}
```

Remarks

None

Preconditions

None

Function

```
void CDCInitEP(void)
```

1.4.1.3.2.2 CDCNotificationHandler Function

Checks for changes in DSR status and reports them to the USB host.

File

usb_device_cdc.h

Syntax

```
void CDCNotificationHandler();
```

Description

Checks for changes in DSR pin state and reports any changes to the USB host.

Remarks

This function is only implemented and needed when the USB_CDC_SUPPORT_DSR_REPORTING option has been enabled. If the function is enabled, it should be called periodically to sample the DSR pin and feed the information to the USB host. This can be done by calling CDCNotificationHandler() by itself, or, by calling CDCTxService() which also calls

CDCNotificationHandler() internally, when appropriate.

Preconditions

CDCInitEP() must have been called previously, prior to calling CDCNotificationHandler() for the first time.

Function

```
void CDCNotificationHandler(void)
```

1.4.1.3.2.3 CDCTxService Function

CDCTxService handles device-to-host transaction(s). This function should be called once per Main Program loop after the device reaches the configured state.

File

```
usb_device_cdc.h
```

Syntax

```
void CDCTxService();
```

Description

CDCTxService handles device-to-host transaction(s). This function should be called once per Main Program loop after the device reaches the configured state (after the CDCInitEP() function has already executed). This function is needed, in order to advance the internal software state machine that takes care of sending multiple transactions worth of IN USB data to the host, associated with CDC serial data. Failure to call CDCTxService() periodically will prevent data from being sent to the USB host, over the CDC serial data interface.

Typical Usage:

```
void main(void)
{
    USBDeviceInit();
    while(1)
    {
        USBDeviceTasks();
        if((USBGetDeviceState() < CONFIGURED_STATE) ||
           (USBIsDeviceSuspended() == true))
        {
            //Either the device is not configured or we are suspended
            // so we don't want to do execute any application code
            continue; //go back to the top of the while loop
        }
        else
        {
            //Keep trying to send data to the PC as required
            CDCTxService();

            //Run application code.
            UserApplication();
        }
    }
}
```

Remarks

None

Preconditions

CDCInitEP() function should have already executed/the device should be in the CONFIGURED_STATE.

Function

```
void CDCTxService(void)
```

1.4.1.3.2.4 getsUSBUSART Function

getsUSBUSART copies a string of BYTES received through USB CDC Bulk OUT endpoint to a user's specified location. It is a non-blocking function. It does not wait for data if there is no data available. Instead it returns '0' to notify the caller that there is no data available.

File

usb_device_cdc.h

Syntax

```
uint8_t getsUSBUSART(uint8_t * buffer, uint8_t len);
```

Returns

uint8_t - Returns a byte indicating the total number of bytes that were actually received and copied into the specified buffer. The returned value can be anything from 0 up to the len input value. A return value of 0 indicates that no new CDC bulk OUT endpoint data was available.

Description

getsUSBUSART copies a string of BYTES received through USB CDC Bulk OUT endpoint to a user's specified location. It is a non-blocking function. It does not wait for data if there is no data available. Instead it returns '0' to notify the caller that there is no data available.

Typical Usage:

```
uint8_t numBytes;
uint8_t buffer[64]

numBytes = getsUSBUSART(buffer, sizeof(buffer)); //until the buffer is free.
if(numBytes > 0)
{
    //we received numBytes bytes of data and they are copied into
    // the "buffer" variable. We can do something with the data
    // here.
}
```

Preconditions

Value of input argument 'len' should be smaller than the maximum endpoint size responsible for receiving bulk data from USB host for CDC class. Input argument 'buffer' should point to a buffer area that is bigger or equal to the size specified by 'len'.

Parameters

Parameters	Description
uint8_t * buffer	Pointer to where received BYTES are to be stored
uint8_t len	The number of BYTES expected.

Function

```
uint8_t getsUSBUSART(char *buffer, uint8_t len)
```

1.4.1.3.2.5 putrsUSBUSART Function

putrsUSBUSART writes a string of data to the USB including the null character. Use this version, 'putrs', to transfer data literals and data located in program memory.

File

usb_device_cdc.h

Syntax

```
void putrsUSBUSART(const char * data);
```

Description

`putrsUSBUSART` writes a string of data to the USB including the null character. Use this version, '`putrs`', to transfer data literals and data located in program memory.

Typical Usage:

```
if(USBUSARTIsTxTrfReady())
{
    putrsUSBUSART("Hello World");
}
```

The transfer mechanism for device-to-host(put) is more flexible than host-to-device(get). It can handle a string of data larger than the maximum size of bulk IN endpoint. A state machine is used to transfer a long string of data over multiple USB transactions. `CDCTxService()` must be called periodically to keep sending blocks of data to the host.

Preconditions

`USBUSARTIsTxTrfReady()` must return true. This indicates that the last transfer is complete and is ready to receive a new block of data. The string of characters pointed to by 'data' must equal to or smaller than 255 BYTES.

Function

```
void putrsUSBUSART(const const char *data)
```

1.4.1.3.2.6 `putsUSBUSART` Function

`putsUSBUSART` writes a string of data to the USB including the null character. Use this version, '`puts`', to transfer data from a RAM buffer.

File

`usb_device_cdc.h`

Syntax

```
void putsUSBUSART(char * data);
```

Description

`putsUSBUSART` writes a string of data to the USB including the null character. Use this version, '`puts`', to transfer data from a RAM buffer.

Typical Usage:

```
if(USBUSARTIsTxTrfReady())
{
    char data[] = "Hello World";
    putsUSBUSART(data);
}
```

The transfer mechanism for device-to-host(put) is more flexible than host-to-device(get). It can handle a string of data larger than the maximum size of bulk IN endpoint. A state machine is used to transfer a long string of data over multiple USB transactions. `CDCTxService()` must be called periodically to keep sending blocks of data to the host.

Preconditions

`USBUSARTIsTxTrfReady()` must return true. This indicates that the last transfer is complete and is ready to receive a new block of data. The string of characters pointed to by 'data' must equal to or smaller than 255 BYTES.

Function

```
void putsUSBUSART(char *data)
```

1.4.1.3.2.7 putUSBUSART Function

putUSBUSART writes an array of data to the USB. Use this version, is capable of transferring 0x00 (what is typically a NULL character in any of the string transfer functions).

File

usb_device_cdc.h

Syntax

```
void putUSBUSART(uint8_t * data, uint8_t Length);
```

Description

putUSBUSART writes an array of data to the USB. Use this version, is capable of transferring 0x00 (what is typically a NULL character in any of the string transfer functions).

Typical Usage:

```
if(USBUSARTIsTxTrfReady( ))
{
    char data[] = {0x00, 0x01, 0x02, 0x03, 0x04};
    putUSBUSART(data,5);
}
```

The transfer mechanism for device-to-host(put) is more flexible than host-to-device(get). It can handle a string of data larger than the maximum size of bulk IN endpoint. A state machine is used to transfer a long string of data over multiple USB transactions. CDCTxService() must be called periodically to keep sending blocks of data to the host.

Preconditions

USBUSARTIsTxTrfReady() must return true. This indicates that the last transfer is complete and is ready to receive a new block of data. The string of characters pointed to by 'data' must equal to or smaller than 255 BYTES.

Function

```
void putUSBUSART(char *data, uint8_t length)
```

1.4.1.3.2.8 USBCDCEventHandler Function

Handles events from the USB stack, which may have an effect on the CDC endpoint(s).

File

usb_device_cdc.h

Syntax

```
bool USBCDCEventHandler(USB_EVENT event, void * pdata, uint16_t size);
```

Description

Handles events from the USB stack. This function should be called when there is a USB event that needs to be processed by the CDC driver.

Preconditions

Value of input argument 'len' should be smaller than the maximum endpoint size responsible for receiving bulk data from USB host for CDC class. Input argument 'buffer' should point to a buffer area that is bigger or equal to the size specified by 'len'.

Parameters

Parameters	Description
USB_EVENT event	the type of event that occurred
void * pdata	pointer to the data that caused the event
uint16_t size	the size of the data that is pointed to by pdata

Function

```
bool USBCDCEventHandler(USB_EVENT event, void *pdata, uint16_t size)
```

1.4.1.3.2.9 USBCheckCDCRequest Function

File

usb_device_cdc.h

Syntax

```
void USBCheckCDCRequest();
```

Description

This routine checks the most recently received SETUP data packet to see if the request is specific to the CDC class. If the request was a CDC specific request, this function will take care of handling the request and responding appropriately.

Remarks

This function does not change status or do anything if the SETUP packet did not contain a CDC class specific request.

Preconditions

This function should only be called after a control transfer SETUP packet has arrived from the host.

Function

```
void USBCheckCDCRequest(void)
```

1.4.1.3.2.10 CDCSetBaudRate Macro

This macro is used set the baud rate reported back to the host during a get line coding request. (optional)

File

usb_device_cdc.h

Syntax

```
#define CDCSetBaudRate(baudRate) {line_coding.dwDTERate=baudRate;}
```

Description

This macro is used set the baud rate reported back to the host during a get line coding request.

Typical Usage:

```
CDCSetBaudRate(19200);
```

This function is optional for CDC devices that do not actually convert the USB traffic to a hardware UART.

Remarks

None

Preconditions

None

Function

```
void CDCSetBaudRate(uint32_t baudRate)
```

1.4.1.3.2.11 CDCSetCharacterFormat Macro

This macro is used manually set the character format reported back to the host during a get line coding request. (optional)

File

usb_device_cdc.h

Syntax

```
#define CDCSetCharacterFormat(charFormat) {line_coding.bCharFormat=charFormat;}
```

Description

This macro is used manually set the character format reported back to the host during a get line coding request.

Typical Usage:

```
CDCSetCharacterFormat(NUM_STOP_BITS_1);
```

This function is optional for CDC devices that do not actually convert the USB traffic to a hardware UART.

Remarks

None

Preconditions

None

Function

```
void CDCSetCharacterFormat(uint8_t charFormat)
```

1.4.1.3.2.12 CDCSetDataSize Macro

This function is used manually set the number of data bits reported back to the host during a get line coding request. (optional)

File

usb_device_cdc.h

Syntax

```
#define CDCSetDataSize(dataBits) {line_coding.bDataBits=dataBits;}
```

Description

This function is used manually set the number of data bits reported back to the host during a get line coding request.

Typical Usage:

```
CDCSetDataSize(8);
```

This function is optional for CDC devices that do not actually convert the USB traffic to a hardware UART.

Remarks

None

Preconditions

None

Function

```
void CDCSetDataSize(uint8_t dataBits)
```

1.4.1.3.2.13 CDCSetLineCoding Macro

This function is used to manually set the data reported back to the host during a get line coding request. (optional)

File

usb_device_cdc.h

Syntax

```
#define CDCSetLineCoding(baud,format,parity,dataSize) {\  
    CDCS...  
    CDCS...  
    CDCS...  
    CDCS...  
}
```

Description

This function is used to manually set the data reported back to the host during a get line coding request.

Typical Usage:

```
CDCSetLineCoding(19200, NUM_STOP_BITS_1, PARITY_NONE, 8);
```

This function is optional for CDC devices that do not actually convert the USB traffic to a hardware UART.

Remarks

None

Preconditions

None

Function

```
void CDCSetLineCoding(uint32_t baud, uint8_t format, uint8_t parity, uint8_t dataSize)
```

1.4.1.3.2.14 CDCSetParity Macro

This function is used manually set the parity format reported back to the host during a get line coding request. (optional)

File

usb_device_cdc.h

Syntax

```
#define CDCSetParity(parityType) {line_coding.bParityType=parityType;}
```

Description

This macro is used manually set the parity format reported back to the host during a get line coding request.

Typical Usage:

```
CDCSetParity(PARITY_NONE);
```

This function is optional for CDC devices that do not actually convert the USB traffic to a hardware UART.

Remarks

None

Preconditions

None

Function

```
void CDCSetParity(uint8_t parityType)
```

1.4.1.3.2.15 mUSBUSARTIsTxTrfReady Macro

File

usb_device_cdc.h

Syntax

```
#define mUSBUSARTIsTxTrfReady USBUSARTIsTxTrfReady()
```

Description

Deprecated in MCHPFSUSB v2.3. This macro has been replaced by USBUSARTIsTxTrfReady().

Function

```
void mUSBUSARTTxRam(uint8_t *pData, uint8_t len)
```

1.4.1.3.2.16 mUSBUSARTTxRam Macro

File

usb_device_cdc.h

Syntax

```
#define mUSBUSARTTxRam(pData,len) \
{ \
    pCDCSrc.bRam = pData; \
    cdc_tx_len = len; \
    cdc_mem_type = USB_EP0_RAM; \
    cdc_trf_state = CDC_TX_BUSY; \
}
```

Description

Use this macro to transfer data located in data memory. Use this macro when:

1. Data stream is not null-terminated
2. Transfer length is known

Remember: cdc_trf_state must == CDC_TX_READY Unlike putsUSBUSART, there is not code double checking the transfer state. Unexpected behavior will occur if this function is called when cdc_trf_state != CDC_TX_READY

Typical Usage:

```
if(USBUSARTIsTxTrfReady() ) \
{ \
    mUSBUSARTTxRam(&UserDataBuffer[0], 200); \
}
```

Remarks

This macro only handles the setup of the transfer. The actual transfer is handled by CDCTxService(). This macro does not "double buffer" the data. The application firmware should not modify the contents of the pData buffer until all of the data has been sent, as indicated by the USBUSARTIsTxTrfReady() function returning true, subsequent to calling mUSBUSARTTxRam().

Preconditions

cdc_trf_state must be in the CDC_TX_READY state. Value of 'len' must be equal to or smaller than 255 bytes. The USB stack should have reached the CONFIGURED_STATE prior to calling this API function for the first time.

Parameters

Parameters	Description
pData	Pointer to the starting location of data bytes
len	Number of bytes to be transferred

Function

```
void mUSBUSARTTxRam(uint8_t *pData, uint8_t len)
```

1.4.1.3.2.17 mUSBUSARTTxRom Macro**File**

usb_device_cdc.h

Syntax

```
#define mUSBUSARTTxRom(pData, len) \
{ \
    pCDCSrc.bRom = pData; \
    cdc_tx_len = len; \
    cdc_mem_type = USB_EP0_ROM; \
    cdc_trf_state = CDC_TX_BUSY; \
}
```

Description

Use this macro to transfer data located in program memory. Use this macro when:

1. Data stream is not null-terminated
2. Transfer length is known

Remember: `cdc_trf_state` must == `CDC_TX_READY` Unlike `putrsUSBUSART`, there is not code double checking the transfer state. Unexpected behavior will occur if this function is called when `cdc_trf_state != CDC_TX_READY`

Typical Usage:

```
if(USBUSARTIsTxTrfReady() ) \
{ \
    mUSBUSARTTxRom(&SomeRomString[0], 200); \
}
```

Remarks

This macro only handles the setup of the transfer. The actual transfer is handled by `CDCTxService()`.

Preconditions

`cdc_trf_state` must be in the `CDC_TX_READY` state. Value of '`len`' must be equal to or smaller than 255 bytes.

Parameters

Parameters	Description
<code>pDdata</code>	Pointer to the starting location of data bytes
<code>len</code>	Number of bytes to be transferred

Function

```
void mUSBUSARTTxRom(rom uint8_t *pData, uint8_t len)
```

1.4.1.3.2.18 USBUSARTIsTxTrfReady Macro

This macro is used to check if the CDC class is ready to send more data.

File

usb_device_cdc.h

Syntax

```
#define USBUSARTIsTxTrfReady (cdc_trf_state == CDC_TX_READY)
```

Description

This macro is used to check if the CDC class handler firmware is ready to send more data to the host over the CDC bulk IN endpoint.

Typical Usage:

```
if(USBUSARTIsTxTrfReady( ))  
{  
    putrsUSART("Hello World");  
}
```

Remarks

Make sure the application periodically calls the CDCTxService() handler, or pending USB IN transfers will not be able to advance and complete.

Preconditions

The return value of this function is only valid if the device is in a configured state (i.e. - USBDeviceGetState() returns CONFIGURED_STATE)

Function

```
bool USBUSARTIsTxTrfReady(void)
```

1.4.1.3.3 CDC_H Macro**File**

```
usb_device_cdc.h
```

Syntax

```
#define CDC_H
```

Module

CDC Function Driver

Description

This is macro CDC_H.

1.4.1.3.4 Data Types and Constants

Data types and constants used to interface with the CDC module.

Macros

Name	Description
NUM_STOP_BITS_1	1 stop bit - used by CDCSetLineCoding() and CDCSetCharacterFormat()
NUM_STOP_BITS_1_5	1.5 stop bit - used by CDCSetLineCoding() and CDCSetCharacterFormat()
NUM_STOP_BITS_2	2 stop bit - used by CDCSetLineCoding() and CDCSetCharacterFormat()
PARITY_EVEN	even parity - used by CDCSetLineCoding() and CDCSetParity()
PARITY_MARK	mark parity - used by CDCSetLineCoding() and CDCSetParity()
PARITY_NONE	no parity - used by CDCSetLineCoding() and CDCSetParity()
PARITY_ODD	odd parity - used by CDCSetLineCoding() and CDCSetParity()
PARITY_SPACE	space parity - used by CDCSetLineCoding() and CDCSetParity()

Module

CDC Function Driver

Description

Data types and constants used to interface with the CDC module.

1.4.1.3.4.1 NUM_STOP_BITS_1 Macro

File

usb_device_cdc.h

Syntax

```
#define NUM_STOP_BITS_1 0 //1 stop bit - used by CDCSetLineCoding() and  
CDCSetCharacterFormat()
```

Description

1 stop bit - used by CDCSetLineCoding() and CDCSetCharacterFormat()

1.4.1.3.4.2 NUM_STOP_BITS_1_5 Macro

File

usb_device_cdc.h

Syntax

```
#define NUM_STOP_BITS_1_5 1 //1.5 stop bit - used by CDCSetLineCoding() and  
CDCSetCharacterFormat()
```

Description

1.5 stop bit - used by CDCSetLineCoding() and CDCSetCharacterFormat()

1.4.1.3.4.3 NUM_STOP_BITS_2 Macro

File

usb_device_cdc.h

Syntax

```
#define NUM_STOP_BITS_2 2 //2 stop bit - used by CDCSetLineCoding() and  
CDCSetCharacterFormat()
```

Description

2 stop bit - used by CDCSetLineCoding() and CDCSetCharacterFormat()

1.4.1.3.4.4 PARITY_EVEN Macro

File

usb_device_cdc.h

Syntax

```
#define PARITY_EVEN 2 //even parity - used by CDCSetLineCoding() and CDCSetParity()
```

Description

even parity - used by CDCSetLineCoding() and CDCSetParity()

1.4.1.3.4.5 PARITY_MARK Macro

File

usb_device_cdc.h

Syntax

```
#define PARITY_MARK 3 //mark parity - used by CDCSetLineCoding() and CDCSetParity()
```

Description

mark parity - used by CDCSetLineCoding() and CDCSetParity()

1.4.1.3.4.6 PARITY_NONE Macro**File**

usb_device_cdc.h

Syntax

```
#define PARITY_NONE 0 //no parity - used by CDCSetLineCoding() and CDCSetParity()
```

Description

no parity - used by CDCSetLineCoding() and CDCSetParity()

1.4.1.3.4.7 PARITY_ODD Macro**File**

usb_device_cdc.h

Syntax

```
#define PARITY_ODD 1 //odd parity - used by CDCSetLineCoding() and CDCSetParity()
```

Description

odd parity - used by CDCSetLineCoding() and CDCSetParity()

1.4.1.3.4.8 PARITY_SPACE Macro**File**

usb_device_cdc.h

Syntax

```
#define PARITY_SPACE 4 //space parity - used by CDCSetLineCoding() and CDCSetParity()
```

Description

space parity - used by CDCSetLineCoding() and CDCSetParity()

1.4.1.4 HID Function Driver**Files**

Name	Description
usb_device_hid.h	This is file usb_device_hid.h.

Description**1.4.1.4.1 Functions****Macros**

Name	Description
HIDRxHandleBusy	Retrieves the status of the buffer ownership

HIDRxPacket	Receives the specified data out the specified endpoint
HIDTxHandleBusy	Retrieves the status of the buffer ownership
HIDTxPacket	Sends the specified data out the specified endpoint

Module

HID Function Driver

Description**1.4.1.4.1.1 HIDRxHandleBusy Macro**

Retrieves the status of the buffer ownership

File

usb_device_hid.h

Syntax

```
#define HIDRxHandleBusy(handle) USBHandleBusy(handle)
```

Description

Retrieves the status of the buffer ownership. This function will indicate if the previous transfer is complete or not.

This function will take the input handle (pointer to a BDT entry) and will check the UOWN bit. If the UOWN bit is set then that indicates that the transfer is not complete and the USB module still owns the data memory. If the UOWN bit is clear that means that the transfer is complete and that the CPU now owns the data memory.

For more information about the BDT, please refer to the appropriate datasheet for the device in use.

Typical Usage:

```
if( !HIDRxHandleBusy(USBOutHandle) )
{
    //The data is available in the buffer that was specified when the
    // HIDRxPacket() was called.
}
```

Remarks

None

Preconditions

None

Return Values

Return Values	Description
TRUE	the HID handle is still busy
FALSE	the HID handle is not busy and is ready to receive additional data.

Function

```
bool HIDRxHandleBusy( USB_HANDLE handle)
```

1.4.1.4.1.2 HIDRxPacket Macro

Receives the specified data out the specified endpoint

File

usb_device_hid.h

Syntax

```
#define HIDRxPacket USBRxOnePacket
```

Description

Receives the specified data out the specified endpoint.

Typical Usage:

```
//Read 64-uint8_ts from endpoint HID_EP, into the ReceivedDataBuffer array.
// Make sure to save the return handle so that we can check it later
// to determine when the transfer is complete.
USBOutHandle = HIDRxPacket(HID_EP,(uint8_t*)&ReceivedDataBuffer,64);
```

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_HANDLE	a handle for the transfer. This information should be kept to track the status of the transfer

Function

```
USB_HANDLE HIDRxPacket(uint8_t ep, uint8_t* data, uint16_t len)
```

1.4.1.4.1.3 HIDTxHandleBusy Macro

Retrieves the status of the buffer ownership

File

usb_device_hid.h

Syntax

```
#define HIDTxHandleBusy(handle) USBHandleBusy(handle)
```

Description

Retrieves the status of the buffer ownership. This function will indicate if the previous transfer is complete or not.

This function will take the input handle (pointer to a BDT entry) and will check the UOWN bit. If the UOWN bit is set then that indicates that the transfer is not complete and the USB module still owns the data memory. If the UOWN bit is clear that means that the transfer is complete and that the CPU now owns the data memory.

For more information about the BDT, please refer to the appropriate datasheet for the device in use.

Typical Usage:

```
//make sure that the last transfer isn't busy by checking the handle
if(!HIDTxHandleBusy(USBInHandle))
{
    //Send the data contained in the ToSendDataBuffer[] array out on
    // endpoint HID_EP
    USBInHandle =
    HIDTxPacket(HID_EP,(uint8_t*)&ToSendDataBuffer[0],sizeof(ToSendDataBuffer));
}
```

Remarks

None

Preconditions

None.

Return Values

Return Values	Description
TRUE	the HID handle is still busy
FALSE	the HID handle is not busy and is ready to send additional data.

Function

```
bool HIDTxHandleBusy( USB_HANDLE handle)
```

1.4.1.4.1.4 HIDTxPacket Macro

Sends the specified data out the specified endpoint

File

usb_device_hid.h

Syntax

```
#define HIDTxPacket USBTxOnePacket
```

Description

This function sends the specified data out the specified endpoint and returns a handle to the transfer information.

Typical Usage:

```
//make sure that the last transfer isn't busy by checking the handle
if(!HIDTxHandleBusy(USBInHandle))
{
    //Send the data contained in the ToSendDataBuffer[] array out on
    // endpoint HID_EP
    USBInHandle =
    HIDTxPacket(HID_EP,(uint8_t*)&ToSendDataBuffer[0],sizeof(ToSendDataBuffer));
}
```

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_HANDLE	a handle for the transfer. This information should be kept to track the status of the transfer

Function

```
USB_HANDLE HIDTxPacket(uint8_t ep, uint8_t* data, uint16_t len)
```

1.4.1.4.2 Data Types and Constants**Macros**

Name	Description
BOOT_INFT_SUBCLASS	HID Interface Class SubClass Codes

BOOT_PROTOCOL	Protocol Selection
HID_PROTOCOL_KEYBOARD	This is macro HID_PROTOCOL_KEYBOARD.
HID_PROTOCOL_MOUSE	This is macro HID_PROTOCOL_MOUSE.
HID_PROTOCOL_NONE	HID Interface Class Protocol Codes

Module

HID Function Driver

Description**1.4.1.4.2.1 BOOT_INTF_SUBCLASS Macro****File**

usb_device_hid.h

Syntax

```
#define BOOT_INTF_SUBCLASS 0x01
```

Description

HID Interface Class SubClass Codes

1.4.1.4.2.2 BOOT_PROTOCOL Macro**File**

usb_device_hid.h

Syntax

```
#define BOOT_PROTOCOL 0x00
```

Description

Protocol Selection

1.4.1.4.2.3 HID_PROTOCOL_KEYBOARD Macro**File**

usb_device_hid.h

Syntax

```
#define HID_PROTOCOL_KEYBOARD 0x01
```

Description

This is macro HID_PROTOCOL_KEYBOARD.

1.4.1.4.2.4 HID_PROTOCOL_MOUSE Macro**File**

usb_device_hid.h

Syntax

```
#define HID_PROTOCOL_MOUSE 0x02
```

Description

This is macro HID_PROTOCOL_MOUSE.

1.4.1.4.2.5 HID_PROTOCOL_NONE Macro

File

usb_device_hid.h

Syntax

```
#define HID_PROTOCOL_NONE 0x00
```

Description

HID Interface Class Protocol Codes

1.4.1.4.3 usb_device_hid.h

Macros

Name	Description
BOOT_INTF_SUBCLASS	HID Interface Class SubClass Codes
BOOT_PROTOCOL	Protocol Selection
HID_PROTOCOL_KEYBOARD	This is macro HID_PROTOCOL_KEYBOARD.
HID_PROTOCOL_MOUSE	This is macro HID_PROTOCOL_MOUSE.
HID_PROTOCOL_NONE	HID Interface Class Protocol Codes
HIDRxHandleBusy	Retrieves the status of the buffer ownership
HIDRxPacket	Receives the specified data out the specified endpoint
HIDTxHandleBusy	Retrieves the status of the buffer ownership
HIDTxPacket	Sends the specified data out the specified endpoint

Module

HID Function Driver

Description

This is file usb_device_hid.h.

1.4.1.5 MSD Function Driver

Files

Name	Description
usb_device_msd.h	This is file usb_device_msd.h.

Description

1.4.1.5.1 Functions

Functions

	Name	Description
💡	MSDTasks	This is function MSDTasks.
💡	MSDTransferTerminated	Check if the host recently did a clear endpoint halt on the MSD OUT endpoint. In this case, we want to re-arm the MSD OUT endpoint, so we are prepared to receive the next CBW that the host might want to send.
💡	USBCheckMSDRequest	
💡	USBMSDInit	This is function USBMSDInit.

Module

MSD Function Driver

Description

1.4.1.5.1.1 MSDTasks Function

File

usb_device_msd.h

Syntax

```
uint8_t MSDTasks();
```

Description

This is function MSDTasks.

1.4.1.5.1.2 MSDTransferTerminated Function

File

usb_device_msd.h

Syntax

```
void MSDTransferTerminated(USB_HANDLE handle);
```

Description

Check if the host recently did a clear endpoint halt on the MSD OUT endpoint. In this case, we want to re-arm the MSD OUT endpoint, so we are prepared to receive the next CBW that the host might want to send.

Remarks

If however the STALL was due to a CBW not valid condition, then we are required to have a persistent STALL, where it cannot be cleared (until MSD reset recovery takes place). See MSD BOT specs v1.0, section 6.6.1.

None

Preconditions

A transfer was terminated. This should be called from the transfer terminated event handler.

Function

```
void MSDTransferTerminated(USB_HANDLE handle)
```

1.4.1.5.1.3 USBCheckMSDRequest Function

File

usb_device_msd.h

Syntax

```
void USBCheckMSDRequest();
```

Section

Public Prototypes

1.4.1.5.1.4 USBMSDInit Function

File

usb_device_msd.h

Syntax

```
void USBMSDInit();
```

Description

This is function USBMSDInit.

1.4.1.5.2 Data Types and Constants

Module

MSD Function Driver

Structures

Name	Description
LUN_FUNCTIONS	LUN_FUNCTIONS is a structure of function pointers that tells the stack where to find each of the physical layer functions it is looking for. This structure needs to be defined for any project for PIC24F or PIC32.

Description

1.4.1.5.2.1 LUN_FUNCTIONS Structure

LUN_FUNCTIONS is a structure of function pointers that tells the stack where to find each of the physical layer functions it is looking for. This structure needs to be defined for any project for PIC24F or PIC32.

File

usb_device_msd.h

Syntax

```
typedef struct {
    FILEIO_MEDIA_INFORMATION* (* MediaInitialize)(void * config);
    uint32_t (* ReadCapacity)(void * config);
    uint16_t (* ReadSectorSize)(void * config);
    bool (* MediaDetect)(void * config);
    uint8_t (* SectorRead)(void * config, uint32_t sector_addr, uint8_t* buffer);
    uint8_t (* WriteProtectState)(void * config);
    uint8_t (* SectorWrite)(void * config, uint32_t sector_addr, uint8_t* buffer, uint8_t
allowWriteToZero);
    void * mediaParameters;
    uint8_t (* AsyncWriteTasks)(void* config, void* pAsyncIO);
    uint8_t (* AsyncReadTasks)(void* config, void* pAsyncIO);
} LUN_FUNCTIONS;
```

Members

Members	Description
FILEIO_MEDIA_INFORMATION* (* MediaInitialize)(void * config);	Function pointer to the MediaInitialize() function of the physical media being used.
uint32_t (* ReadCapacity)(void * config);	Function pointer to the ReadCapacity() function of the physical media being used.
uint16_t (* ReadSectorSize)(void * config);	Function pointer to the ReadSectorSize() function of the physical media being used.
bool (* MediaDetect)(void * config);	Function pointer to the MediaDetect() function of the physical media being used.

<code>uint8_t (* SectorRead)(void * config, uint32_t sector_addr, uint8_t* buffer);</code>	Function pointer to the SectorRead() function of the physical media being used.
<code>uint8_t (* WriteProtectState)(void * config);</code>	Function pointer to the WriteProtectState() function of the physical media being used.
<code>uint8_t (* SectorWrite)(void * config, uint32_t sector_addr, uint8_t* buffer, uint8_t allowWriteToZero);</code>	Function pointer to the SectorWrite() function of the physical media being used.
<code>void * mediaParameters;</code>	Pointer to a media-specific parameter structure
<code>uint8_t (* AsyncWriteTasks)(void* config, void* pAsyncIO);</code>	Function pointer to the async write tasks function of the physical media being used.
<code>uint8_t (* AsyncReadTasks)(void* config, void* pAsyncIO);</code>	Function pointer to the async read tasks function of the physical media being used.

Description

LUN_FUNCTIONS is a structure of function pointers that tells the stack where to find each of the physical layer functions it is looking for. This structure needs to be defined for any project for PIC24F or PIC32.

Typical Usage:

```
LUN_FUNCTIONS LUN[MAX_LUN + 1] =
{
    {
        &MDD_SDSPI_MediaInitialize,
        &MDD_SDSPI_ReadCapacity,
        &MDD_SDSPI_ReadSectorSize,
        &MDD_SDSPI_MediaDetect,
        &MDD_SDSPI_SectorRead,
        &MDD_SDSPI_WriteProtectState,
        &MDD_SDSPI_SectorWrite
    }
};
```

In the above code we are passing the address of the SDSPI functions to the corresponding member of the LUN_FUNCTIONS structure. In the above case we have created an array of LUN_FUNCTIONS structures so that it is possible to have multiple physical layers by merely increasing the MAX_LUN variable and by adding one more set of entries in the array. Please take caution to insure that each function is in the the correct location in the structure. Incorrect alignment will cause the USB stack to call the incorrect function for a given command.

See the MDD File System Library for additional information about the available physical media, their requirements, and how to use their associated functions.

1.4.1.5.3 usb_device_msd.h

Functions

	Name	Description
💡	MSDTask	This is function MSDTask.
💡	MSDTransferTerminated	Check if the host recently did a clear endpoint halt on the MSD OUT endpoint. In this case, we want to re-arm the MSD OUT endpoint, so we are prepared to receive the next CBW that the host might want to send.
💡	USBCheckMSDRequest	
💡	USBMSDInit	This is function USBMSDInit.

Module

MSD Function Driver

Structures

Name	Description
LUN_FUNCTIONS	LUN_FUNCTIONS is a structure of function pointers that tells the stack where to find each of the physical layer functions it is looking for. This structure needs to be defined for any project for PIC24F or PIC32.

Description

This is file `usb_device_msd.h`.

1.4.1.6 Vendor Class (Generic) Function Driver

Files

Name	Description
<code>usb_device_generic.h</code>	This is file <code>usb_device_generic.h</code> .

Description

1.4.1.6.1 Functions

Functions

	Name	Description
💡	<code>USBCheckVendorRequest</code>	This routine handles vendor class specific requests that happen on EP0. This function should be called from the <code>USBCBCheckOtherReq()</code> call back function whenever implementing a custom/vendor class device.

Macros

Name	Description
<code>USBGEN_H</code>	This is macro <code>USBGEN_H</code> .
<code>USBGenRead</code>	Receives the specified data out the specified endpoint
<code>USBGenWrite</code>	Sends the specified data out the specified endpoint

Module

Vendor Class (Generic) Function Driver

Description

1.4.1.6.1.1 `USBCheckVendorRequest` Function

This routine handles vendor class specific requests that happen on EP0. This function should be called from the `USBCBCheckOtherReq()` call back function whenever implementing a custom/vendor class device.

File

`usb_device_generic.h`

Syntax

```
void USBCheckVendorRequest();
```

Description

This routine handles vendor specific requests that may arrive on EP0 as a control transfer. These can include, but are not necessarily limited to, requests for Microsoft specific OS feature descriptor(s). This function should be called from the `USBCBCheckOtherReq()` call back function whenever using a vendor class device.

Typical Usage:

```
void USBCBCheckOtherReq(void)
{
    //Since the stack didn't handle the request I need to check
    // my class drivers to see if it is for them
    USBCheckVendorRequest();
}
```

```
}
```

Remarks

This function normally gets called within the same context as the `USBDeviceTasks()` function, just after a new control transfer request from the host has arrived. If the USB stack is operated in `USB_INTERRUPT` mode (a `usb_config.h` option), then this function will be executed in the interrupt context. If however the USB stack is operated in the `USB_POLLING` mode, then this function executes in the main loop context.

In order to respond to class specific control transfer request(s) in this handler function, it is suggested to use one or more of the `USBEP0SendRAMPtr()`, `USBEP0SendROMPtr()`, or `USBEP0Receive()` API functions.

Preconditions

None

Function

```
void USBCheckVendorRequest(void)
```

1.4.1.6.1.2 USBGEN_H Macro**File**

`usb_device_generic.h`

Syntax

```
#define USBGEN_H
```

Description

This is macro `USBGEN_H`.

1.4.1.6.1.3 USBGenRead Macro

Receives the specified data out the specified endpoint

File

`usb_device_generic.h`

Syntax

```
#define USBGenRead(ep,data,len) USBRxOnePacket(ep,data,len)
```

Description

Receives the specified data out the specified endpoint.

Typical Usage:

```
//Read 64-bytes from endpoint USBGEN_EP_NUM, into the OUTPacket array.
// Make sure to save the return handle so that we can check it later
// to determine when the transfer is complete.
if(!USBHandleBusy(USBOutHandle))
{
    USBOutHandle = USBGenRead(USBGEN_EP_NUM,(BYTE*)&OUTPacket,64);
}
```

Remarks

None

Preconditions

None

Return Values

Return Values	Description
<code>USB_HANDLE</code>	a handle for the transfer. This information should be kept to track the status of the transfer

Function

```
USB_HANDLE USBGenRead(BYTE ep, BYTE* data, WORD len)
```

1.4.1.6.1.4 USBGenWrite Macro

Sends the specified data out the specified endpoint

File

```
usb_device_generic.h
```

Syntax

```
#define USBGenWrite(ep,data,len) USBTxOnePacket(ep,data,len)
```

Description

This function sends the specified data out the specified endpoint and returns a handle to the transfer information.

Typical Usage:

```
//make sure that the last transfer isn't busy by checking the handle
if( !USBHandleBusy(USBGenericInHandle) )
{
    //Send the data contained in the INPacket[] array out on
    // endpoint USBDEN_EP_NUM
    USBGenericInHandle = USBGenWrite(USBDEN_EP_NUM, (BYTE*)&INPacket[0], sizeof(INPacket));
}
```

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_HANDLE	a handle for the transfer. This information should be kept to track the status of the transfer

Function

```
USB_HANDLE USBGenWrite(BYTE ep, BYTE* data, WORD len)
```

1.4.1.6.2 usb_device_generic.h**Functions**

	Name	Description
💡	USBCheckVendorRequest	This routine handles vendor class specific requests that happen on EP0. This function should be called from the USBCBCheckOtherReq() call back function whenever implementing a custom/vendor class device.

Macros

Name	Description
USBGEN_H	This is macro USBGEN_H.
USBGenRead	Receives the specified data out the specified endpoint
USBGenWrite	Sends the specified data out the specified endpoint

Module

Vendor Class (Generic) Function Driver

Description

This is file `usb_device_generic.h`.

1.4.2 Embedded Host API

These are the various client drivers that are available for use with the USB Embedded Host driver.

Modules

Name	Description
Embedded Host Stack	The USB Embedded Host driver provides low-level USB functionality for all host client drivers.
CDC Client Driver	This is a CDC client driver for use with the USB Embedded Host driver.
HID Client Driver	This client driver provides USB Embedded Host support for HID devices.
Mass Storage Client Driver	This client driver provides USB Embedded Host support for mass storage devices.

Description

1.4.2.1 Embedded Host Stack

The USB Embedded Host driver provides low-level USB functionality for all host client drivers.

Files

Name	Description
<code>usb_host.h</code>	This is file <code>usb_host.h</code> .

Macros

Name	Description
<code>__USBHOST_H__</code>	DOM-IGNORE-END

Description

The USB Embedded Host driver provides low-level USB functionality for all host client drivers. This layer is responsible for enumerating devices, managing data transfers, and detecting device detach.

Typically, only host client drivers will interact with this layer. Applications can be configured to receive some events from this layer, such as `EVENT_REQUEST_POWER` and `EVENT_RELEASE_POWER`.

See [AN1140 USB Embedded Host Stack](#) for more information about this layer. See [AN1141 USB Embedded Host Stack Programmer's Guide](#) for more information about creating a client driver that uses this layer.

1.4.2.1.1 Functions

Functions

	Name	Description
	<code>USB_HOST_APP_DATA_EVENT_HANDLER</code>	This is a typedef to use when defining the application level data events handler.

	USB_HOST_APP_EVENT_HANDLER	This is a typedef to use when defining the application level events handler.
	USB_HostInterruptHandler	This function handles the interrupts when the USB module is running in host mode.
	USBHostClearEndpointErrors	This function clears an endpoint's internal error condition.
	USBHostDeviceSpecificClientDriver	This function indicates if the specified device has explicit client driver support specified in the TPL.
	USBHostDeviceStatus	This function returns the current status of a device.
	USBHostInit	This function initializes the variables of the USB host stack.
	USBHostIsochronousBuffersCreate	This function initializes the isochronous data buffer information and allocates memory for each buffer. This function will not allocate memory if the buffer pointer is not NULL.
	USBHostIsochronousBuffersDestroy	This function releases all of the memory allocated for the isochronous data buffers. It also resets all other information about the buffers.
	USBHostIsochronousBuffersReset	This function resets all the isochronous data buffers. It does not do anything with the space allocated for the buffers.
	USBHostIssueDeviceRequest	This function sends a standard device request to the attached device.
	USBHostRead	This function initiates a read from the attached device.
	USBHostResetDevice	This function resets an attached device.
	USBHostResumeDevice	This function issues a RESUME to the attached device.
	USBHostSetDeviceConfiguration	This function changes the device's configuration.
	USBHostSetNAKTimeout	This function specifies NAK timeout capability.
	USBHostShutdown	This function turns off the USB module and frees all unnecessary memory. This routine can be called by the application layer to shut down all USB activity, which effectively detaches all devices. The event EVENT_DETACH will be sent to the client drivers for the attached device, and the event EVENT_VBUS_RELEASE_POWER will be sent to the application layer.
	USBHostSuspendDevice	This function suspends a device.
	USBHostTasks	This function executes the host tasks for USB host operation.
	USBHostTerminateTransfer	This function terminates the current transfer for the given endpoint.
	USBHostTransferIsComplete	This function initiates whether or not the last endpoint transaction is complete.
	USBHostVbusEvent	This function handles Vbus events that are detected by the application.
	USBHostWrite	This function initiates a write to the attached device.

Macros

Name	Description
USBHostGetCurrentConfigurationDescriptor	This function returns a pointer to the current configuration descriptor of the requested device.
USBHostGetDeviceDescriptor	This function returns a pointer to the device descriptor of the requested device.
USBHostGetStringDescriptor	This routine initiates a request to obtain the requested string descriptor.
USBHostReadIsochronous	This function initiates a read from an isochronous endpoint on the attached device.
USBHostWriteIsochronous	This function initiates a write to an isochronous endpoint on the attached device.

Module

Embedded Host Stack

Description**1.4.2.1.1.1 USB_HOST_APP_DATA_EVENT_HANDLER Function**

This is a typedef to use when defining the application level data events handler.

File

usb_host.h

Syntax

```
bool USB_HOST_APP_DATA_EVENT_HANDLER(uint8_t address, USB_EVENT event, void * data,  
uint32_t size);
```

Description

This function is implemented by the application. The function name can be anything - the macro USB_HOST_APP_EVENT_HANDLER must be set in usb_config.h to the name of the application function.

In the application layer, this function is responsible for handling all application-level data events that are generated by the stack. See the enumeration USB_EVENT for a complete list of all events that can occur. Note that only data events, such as EVENT_DATA_ISOC_READ, will be passed to this event handler.

If the application can handle the event successfully, the function should return true.

Remarks

If this function is not provided by the application, then all application events are assumed to function without error.

Preconditions

None

Return Values

Return Values	Description
true	Event was processed successfully
false	Event was not processed successfully

Function

```
bool USB_HOST_APP_DATA_EVENT_HANDLER ( uint8_t address, USB_EVENT event,  
void *data, uint32_t size )
```

1.4.2.1.1.2 USB_HOST_APP_EVENT_HANDLER Function

This is a typedef to use when defining the application level events handler.

File

usb_host.h

Syntax

```
bool USB_HOST_APP_EVENT_HANDLER(uint8_t address, USB_EVENT event, void * data, uint32_t  
size);
```

Description

This function is implemented by the application. The function name can be anything - the macro USB_HOST_APP_EVENT_HANDLER must be set in usb_config.h to the name of the application function.

In the application layer, this function is responsible for handling all application-level events that are generated by the stack. See the enumeration USB_EVENT for a complete list of all events that can occur. Note that some of these events are intended for client drivers (e.g. EVENT_TRANSFER), while some are intended for the application layer (e.g. EVENT_UNSUPPORTED_DEVICE).

If the application can handle the event successfully, the function should return true. For example, if the function receives the event EVENT_VBUS_REQUEST_POWER and the system can allocate that much power to an attached device, the function should return true. If, however, the system cannot allocate that much power to an attached device, the function should return false.

Remarks

If this function is not provided by the application, then all application events are assumed to function without error.

Preconditions

None

Return Values

Return Values	Description
true	Event was processed successfully
false	Event was not processed successfully

Function

```
bool USB_HOST_APP_EVENT_HANDLER ( uint8_t address, USB_EVENT event,
void *data, uint32_t size )
```

1.4.2.1.1.3 USB_HostInterruptHandler Function

This function handles the interrupts when the USB module is running in host mode.

File

usb_host.h

Syntax

```
void USB_HostInterruptHandler();
```

Description

This function handles the interrupts when the USB module is running in host mode. It will clear all USB based interrupts as applicable. It should only be called when the module is in host mode.

Preconditions

Should only be called when in host mode.

Function

```
void USB_HostInterruptHandler(void);
```

1.4.2.1.1.4 USBHostClearEndpointErrors Function

This function clears an endpoint's internal error condition.

File

usb_host.h

Syntax

```
uint8_t USBHostClearEndpointErrors(uint8_t deviceAddress, uint8_t endpoint);
```

Description

This function is called to clear the internal error condition of a device's endpoint. It should be called after the application has dealt with the error condition on the device. This routine clears internal status only; it does not interact with the device.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_SUCCESS	Errors cleared
USB_UNKNOWN_DEVICE	Device not found
USB_ENDPOINT_NOT_FOUND	Specified endpoint not found

Function

```
uint8_t USBHostClearEndpointErrors( uint8_t deviceAddress, uint8_t endpoint )
```

1.4.2.1.1.5 USBHostDeviceSpecificClientDriver Function

This function indicates if the specified device has explicit client driver support specified in the TPL.

File

usb_host.h

Syntax

```
bool USBHostDeviceSpecificClientDriver(uint8_t deviceAddress);
```

Description

This function indicates if the specified device has explicit client driver support specified in the TPL. It is used in client drivers' USB_CLIENT_INIT routines to indicate that the client driver should be used even though the class, subclass, and protocol values may not match those normally required by the class. For example, some printing devices do not fulfill all of the requirements of the printer class, so their class, subclass, and protocol fields indicate a custom driver rather than the printer class. But the printer class driver can still be used, with minor limitations.

Remarks

This function is used so client drivers can allow certain devices to enumerate. For example, some printer devices indicate a custom class rather than the printer class, even though the device has only minor limitations from the full printer class. The printer client driver will fail to initialize the device if it does not indicate printer class support in its interface descriptor. The printer client driver could allow any device with an interface that matches the printer class endpoint configuration, but both printer and mass storage devices utilize one bulk IN and one bulk OUT endpoint. So a mass storage device would be erroneously initialized as a printer device. This function allows a client driver to know that the client driver support was specified explicitly in the TPL, so for this particular device only, the class, subclass, and protocol fields can be safely ignored.

Preconditions

None

Return Values

Return Values	Description
true	This device is listed in the TPL by VID andPID, and has explicit client driver support.
false	This device is not listed in the TPL by VID and PID.

Function

```
bool USBHostDeviceSpecificClientDriver( uint8_t deviceAddress )
```

1.4.2.1.1.6 USBHostDeviceStatus Function

This function returns the current status of a device.

File

usb_host.h

Syntax

```
uint8_t USBHostDeviceStatus(uint8_t deviceAddress);
```

Description

This function returns the current status of a device. If the device is in a holding state due to an error, the error is returned.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_DEVICE_ATTACHED	Device is attached and running
USB_DEVICE_DETACHED	No device is attached
USB_DEVICE_ENUMERATING	Device is enumerating
USB_HOLDING_OUT_OF_MEMORY	Not enough heap space available
USB_HOLDING_UNSUPPORTED_DEVICE	Invalid configuration or unsupported class
USB_HOLDING_UNSUPPORTED_HUB	Hubs are not supported
USB_HOLDING_INVALID_CONFIGURATION	Invalid configuration requested
USB_HOLDING_PROCESSING_CAPACITY	Processing requirement excessive
USB_HOLDING_POWER_REQUIREMENT	Power requirement excessive
USB_HOLDING_CLIENT_INIT_ERROR	Client driver failed to initialize
USB_DEVICE_SUSPENDED	Device is suspended
Other	Device is holding in an error state. The return value indicates the error.

Function

```
uint8_t USBHostDeviceStatus( uint8_t deviceAddress )
```

1.4.2.1.1.7 USBHostInit Function

This function initializes the variables of the USB host stack.

File

usb_host.h

Syntax

```
bool USBHostInit(unsigned long flags);
```

Description

This function initializes the variables of the USB host stack. It does not initialize the hardware. The peripheral itself is initialized in one of the state machine states. Therefore, USBHostTasks() should be called soon after this function.

Remarks

If the endpoint list is empty, an entry is created in the endpoint list for EP0. If the list is not empty, free all allocated memory other than the EP0 node. This allows the routine to be called multiple times by the application.

Preconditions

None

Parameters

Parameters	Description
unsigned long flags	reserved

Return Values

Return Values	Description
true	Initialization successful
false	Could not allocate memory.

Function

```
bool USBHostInit( unsigned long flags )
```

1.4.2.1.1.8 USBHostIsochronousBuffersCreate Function**File**

usb_host.h

Syntax

```
bool USBHostIsochronousBuffersCreate( ISOCHRONOUS_DATA * isocData, uint8_t numberOfBuffers,
uint16_t bufferSize );
```

Description

This function initializes the isochronous data buffer information and allocates memory for each buffer. This function will not allocate memory if the buffer pointer is not NULL.

Remarks

This function is available only if USB_SUPPORT_ISOCHRONOUS_TRANSFERS is defined in usb_config.h.

Preconditions

None

Return Values

Return Values	Description
true	All buffers are allocated successfully.
false	Not enough heap space to allocate all buffers - adjust the project to provide more heap space.

Function

```
bool USBHostIsochronousBuffersCreate( ISOCHRONOUS_DATA * isocData,
uint8_t numberOfBuffers, uint16_t bufferSize )
```

1.4.2.1.1.9 USBHostIsochronousBuffersDestroy Function**File**

usb_host.h

Syntax

```
void USBHostIsochronousBuffersDestroy( ISOCHRONOUS_DATA * isocData, uint8_t numberOfBuffers );
```

Returns

None

Description

This function releases all of the memory allocated for the isochronous data buffers. It also resets all other information about the buffers.

Remarks

This function is available only if USB_SUPPORT_ISOCHRONOUS_TRANSFERS is defined in usb_config.h.

Preconditions

None

Function

```
void USBHostIsochronousBuffersDestroy( ISOCHRONOUS_DATA * isocData, uint8_t numberOfBuffers )
```

1.4.2.1.1.10 USBHostIsochronousBuffersReset Function

File

usb_host.h

Syntax

```
void USBHostIsochronousBuffersReset( ISOCHRONOUS_DATA * isocData, uint8_t numberOfBuffers );
```

Returns

None

Description

This function resets all the isochronous data buffers. It does not do anything with the space allocated for the buffers.

Remarks

This function is available only if USB_SUPPORT_ISOCHRONOUS_TRANSFERS is defined in usb_config.h.

Preconditions

None

Function

```
void USBHostIsochronousBuffersReset( ISOCHRONOUS_DATA * isocData, uint8_t numberOfBuffers )
```

1.4.2.1.1.11 USBHostIssueDeviceRequest Function

This function sends a standard device request to the attached device.

File

usb_host.h

Syntax

```
uint8_t USBHostIssueDeviceRequest(uint8_t deviceAddress, uint8_t bmRequestType, uint8_t bRequest, uint16_t wValue, uint16_t wIndex, uint16_t wLength, uint8_t * data, uint8_t dataDirection, uint8_t clientDriverID);
```

Description

This function sends a standard device request to the attached device. The user must pass in the parameters of the device request. If there is input or output data associated with the request, a pointer to the data must be provided. The direction of the associated data (input or output) must also be indicated.

This function does no special processing in regards to the request except for three requests. If SET INTERFACE is sent, then DTS is reset for all endpoints. If CLEAR FEATURE (ENDPOINT HALT) is sent, then DTS is reset for that endpoint. If SET CONFIGURATION is sent, the request is aborted with a failure. The function USBHostSetDeviceConfiguration() must be called to change the device configuration, since endpoint definitions may change.

Remarks

DTS reset is done before the command is issued.

Preconditions

The host state machine should be in the running state, and no reads or writes to EP0 should be in progress.

Return Values

Return Values	Description
USB_SUCCESS	Request processing started
USB_UNKNOWN_DEVICE	Device not found
USB_INVALID_STATE	The host must be in a normal running state to do this request
USB_ENDPOINT_BUSY	A read or write is already in progress
USB_ILLEGAL_REQUEST	SET CONFIGURATION cannot be performed with this function.

Function

```
uint8_t USBHostIssueDeviceRequest( uint8_t deviceAddress, uint8_t bmRequestType,
                                  uint8_t bRequest, uint16_t wValue, uint16_t wIndex, uint16_t wLength,
                                  uint8_t *data, uint8_t dataDirection, uint8_t clientDriverID )
```

1.4.2.1.1.12 USBHostRead Function

This function initiates a read from the attached device.

File

usb_host.h

Syntax

```
uint8_t USBHostRead(uint8_t deviceAddress, uint8_t endpoint, uint8_t * data, uint32_t size);
```

Description

This function initiates a read from the attached device.

If the endpoint is isochronous, special conditions apply. The pData and size parameters have slightly different meanings, since multiple buffers are required. Once started, an isochronous transfer will continue with no upper layer intervention until USBHostTerminateTransfer() is called. The ISOCHRONOUS_DATA_BUFFERS structure should not be manipulated until the transfer is terminated.

To clarify parameter usage and to simplify casting, use the macro USBHostReadIsochronous() when reading from an isochronous endpoint.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_SUCCESS	Read started successfully.
USB_UNKNOWN_DEVICE	Device with the specified address not found.
USB_INVALID_STATE	We are not in a normal running state.
USB_ENDPOINT_ILLEGAL_TYPE	Must use USBHostControlRead to read from a control endpoint.
USB_ENDPOINT_ILLEGAL_DIRECTION	Must read from an IN endpoint.
USB_ENDPOINT_STALLED	Endpoint is stalled. Must be cleared by the application.
USB_ENDPOINT_ERROR	Endpoint has too many errors. Must be cleared by the application.

USB_ENDPOINT_BUSY	A Read is already in progress.
USB_ENDPOINT_NOT_FOUND	Invalid endpoint.

Function

```
uint8_t USBHostRead( uint8_t deviceAddress, uint8_t endpoint, uint8_t *pData,
                     uint32_t size )
```

1.4.2.1.1.13 USBHostResetDevice Function

This function resets an attached device.

File

usb_host.h

Syntax

```
uint8_t USBHostResetDevice(uint8_t deviceAddress);
```

Description

This function places the device back in the RESET state, to issue RESET signaling. It can be called only if the state machine is not in the DETACHED state.

Remarks

In order to do a full clean-up, the state is set back to STATE_DETACHED rather than a reset state. The ATTACH interrupt will automatically be triggered when the module is re-enabled, and the proper reset will be performed.

Preconditions

None

Return Values

Return Values	Description
USB_SUCCESS	Success
USB_UNKNOWN_DEVICE	Device not found
USB_ILLEGAL_REQUEST	Device cannot RESUME unless it is suspended

Function

```
uint8_t USBHostResetDevice( uint8_t deviceAddress )
```

1.4.2.1.1.14 USBHostResumeDevice Function

This function issues a RESUME to the attached device.

File

usb_host.h

Syntax

```
uint8_t USBHostResumeDevice(uint8_t deviceAddress);
```

Description

This function issues a RESUME to the attached device. It can be called only if the state machine is in the suspend state.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_SUCCESS	Success
USB_UNKNOWN_DEVICE	Device not found
USB_ILLEGAL_REQUEST	Device cannot RESUME unless it is suspended

Function

```
uint8_t USBHostResumeDevice( uint8_t deviceAddress )
```

1.4.2.1.1.15 USBHostSetDeviceConfiguration Function

This function changes the device's configuration.

File

usb_host.h

Syntax

```
uint8_t USBHostSetDeviceConfiguration(uint8_t deviceAddress, uint8_t configuration);
```

Description

This function is used by the application to change the device's Configuration. This function must be used instead of USBHostIssueDeviceRequest(), because the endpoint definitions may change.

To see when the reconfiguration is complete, use the USBHostDeviceStatus() function. If configuration is still in progress, this function will return USB_DEVICE_ENUMERATING.

Remarks

If an invalid configuration is specified, this function cannot return an error. Instead, the event USB_UNSUPPORTED_DEVICE will be sent to the application layer and the device will be placed in a holding state with a USB_HOLDING_UNSUPPORTED_DEVICE error returned by USBHostDeviceStatus().

Preconditions

The host state machine should be in the running state, and no reads or writes should be in progress.

Example

```
rc = USBHostSetDeviceConfiguration( attachedDevice, configuration );
if (rc)
{
    // Error - cannot set configuration.
}
else
{
    while (USBHostDeviceStatus( attachedDevice ) == USB_DEVICE_ENUMERATING)
    {
        USBHostTasks();
    }
}
if (USBHostDeviceStatus( attachedDevice ) != USB_DEVICE_ATTACHED)
{
    // Error - cannot set configuration.
}
```

Return Values

Return Values	Description
USB_SUCCESS	Process of changing the configuration was started successfully.
USB_UNKNOWN_DEVICE	Device not found
USB_INVALID_STATE	This function cannot be called during enumeration or while performing a device request.

USB_BUSY	No IN or OUT transfers may be in progress.
----------	--

Function

```
uint8_t USBHostSetDeviceConfiguration( uint8_t deviceAddress, uint8_t configuration )
```

1.4.2.1.1.16 USBHostSetNAKTimeout Function

This function specifies NAK timeout capability.

File

usb_host.h

Syntax

```
uint8_t USBHostSetNAKTimeout( uint8_t deviceAddress, uint8_t endpoint, uint16_t flags,
                             uint16_t timeoutCount );
```

Description

This function is used to set whether or not an endpoint on a device should time out a transaction based on the number of NAKs received, and if so, how many NAKs are allowed before the timeout.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_SUCCESS	NAK timeout was configured successfully.
USB_UNKNOWN_DEVICE	Device not found.
USB_ENDPOINT_NOT_FOUND	The specified endpoint was not found.

Function

```
uint8_t USBHostSetNAKTimeout( uint8_t deviceAddress, uint8_t endpoint, uint16_t flags,
                             uint16_t timeoutCount )
```

1.4.2.1.1.17 USBHostShutdown Function**File**

usb_host.h

Syntax

```
void USBHostShutdown( );
```

Returns

None

Description

This function turns off the USB module and frees all unnecessary memory. This routine can be called by the application layer to shut down all USB activity, which effectively detaches all devices. The event EVENT_DETACH will be sent to the client drivers for the attached device, and the event EVENT_VBUS_RELEASE_POWER will be sent to the application layer.

Remarks

None

Preconditions

None

Parameters

Parameters	Description
None	None

Function

```
void USBHostShutdown( void )
```

1.4.2.1.1.18 USBHostSuspendDevice Function

This function suspends a device.

File

```
usb_host.h
```

Syntax

```
uint8_t USBHostSuspendDevice( uint8_t deviceAddress );
```

Description

This function put a device into an IDLE state. It can only be called while the state machine is in normal running mode. After 3ms, the attached device should go into SUSPEND mode.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_SUCCESS	Success
USB_UNKNOWN_DEVICE	Device not found
USB_ILLEGAL_REQUEST	Cannot suspend unless device is in normal run mode

Function

```
uint8_t USBHostSuspendDevice( uint8_t deviceAddress )
```

1.4.2.1.1.19 USBHostTasks Function

This function executes the host tasks for USB host operation.

File

```
usb_host.h
```

Syntax

```
void USBHostTasks( );
```

Returns

None

Description

This function executes the host tasks for USB host operation. It must be executed on a regular basis to keep everything functioning.

The primary purpose of this function is to handle device attach/detach and enumeration. It does not handle USB packet transmission or reception; that must be done in the USB interrupt handler to ensure timely operation.

This routine should be called on a regular basis, but there is no specific time requirement. Devices will still be able to attach, enumerate, and detach, but the operations will occur more slowly as the calling interval increases.

Remarks

None

Preconditions

USBHostInit() has been called.

Function

```
void USBHostTasks( void )
```

1.4.2.1.1.20 USBHostTerminateTransfer Function

This function terminates the current transfer for the given endpoint.

File

usb_host.h

Syntax

```
void USBHostTerminateTransfer(uint8_t deviceAddress, uint8_t endpoint);
```

Returns

None

Description

This function terminates the current transfer for the given endpoint. It can be used to terminate reads or writes that the device is not responding to. It is also the only way to terminate an isochronous transfer.

Remarks

None

Preconditions

None

Function

```
void USBHostTerminateTransfer( uint8_t deviceAddress, uint8_t endpoint )
```

1.4.2.1.1.21 USBHostTransferIsComplete Function

This function initiates whether or not the last endpoint transaction is complete.

File

usb_host.h

Syntax

```
bool USBHostTransferIsComplete(uint8_t deviceAddress, uint8_t endpoint, uint8_t *  
errorCode, uint32_t * byteCount);
```

Description

This function initiates whether or not the last endpoint transaction is complete. If it is complete, an error code and the number of bytes transferred are returned.

For isochronous transfers, byteCount is not valid. Instead, use the returned byte counts for each EVENT_TRANSFER event that was generated during the transfer.

Remarks

Possible values for errorCode are:

- USB_SUCCESS - Transfer successful

- USB_UNKNOWN_DEVICE - Device not attached
- USB_ENDPOINT_STALLED - Endpoint STALL'd
- USB_ENDPOINT_ERROR_ILLEGAL_PID - Illegal PID returned
- USB_ENDPOINT_ERROR_BIT_STUFF
- USB_ENDPOINT_ERROR_DMA
- USB_ENDPOINT_ERROR_TIMEOUT
- USB_ENDPOINT_ERROR_DATA_FIELD
- USB_ENDPOINT_ERROR_CRC16
- USB_ENDPOINT_ERROR_END_OF_FRAME
- USB_ENDPOINT_ERROR_PID_CHECK
- USB_ENDPOINT_ERROR - Other error

Preconditions

None

Return Values

Return Values	Description
true	Transfer is complete.
false	Transfer is not complete.

Function

```
bool USBHostTransferIsComplete( uint8_t deviceAddress, uint8_t endpoint,
                               uint8_t *errorCode, uint32_t *byteCount )
```

1.4.2.1.1.22 USBHostVbusEvent Function

This function handles Vbus events that are detected by the application.

File

usb_host.h

Syntax

```
uint8_t USBHostVbusEvent(USB_EVENT vbusEvent, uint8_t hubAddress, uint8_t portNumber);
```

Description

This function handles Vbus events that are detected by the application. Since Vbus management is application dependent, the application is responsible for monitoring Vbus and detecting overcurrent conditions and removal of the overcurrent condition. If the application detects an overcurrent condition, it should call this function with the event EVENT_VBUS_OVERCURRENT with the address of the hub and port number that has the condition. When a port returns to normal operation, the application should call this function with the event EVENT_VBUS_POWER_AVAILABLE so the stack knows that it can allow devices to attach to that port.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_SUCCESS	Event handled
USB_ILLEGAL_REQUEST	Invalid event, hub, or port

Function

```
uint8_t USBHostVbusEvent( USB_EVENT vbusEvent, uint8_t hubAddress,
                          uint8_t portNumber)
```

1.4.2.1.1.23 USBHostWrite Function

This function initiates a write to the attached device.

File

usb_host.h

Syntax

```
uint8_t USBHostWrite(uint8_t deviceAddress, uint8_t endpoint, uint8_t * data, uint32_t size);
```

Description

This function initiates a write to the attached device. The data buffer pointed to by **data* must remain valid during the entire time that the write is taking place; the data is not buffered by the stack.

If the endpoint is isochronous, special conditions apply. The *pData* and *size* parameters have slightly different meanings, since multiple buffers are required. Once started, an isochronous transfer will continue with no upper layer intervention until *USBHostTerminateTransfer()* is called. The ISOCHRONOUS_DATA_BUFFERS structure should not be manipulated until the transfer is terminated.

To clarify parameter usage and to simplify casting, use the macro *USBHostWriteIsochronous()* when writing to an isochronous endpoint.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_SUCCESS	Write started successfully.
USB_UNKNOWN_DEVICE	Device with the specified address not found.
USB_INVALID_STATE	We are not in a normal running state.
USB_ENDPOINT_ILLEGAL_TYPE	Must use <i>USBHostControlWrite</i> to write to a control endpoint.
USB_ENDPOINT_ILLEGAL_DIRECTION	Must write to an OUT endpoint.
USB_ENDPOINT_STALLED	Endpoint is stalled. Must be cleared by the application.
USB_ENDPOINT_ERROR	Endpoint has too many errors. Must be cleared by the application.
USB_ENDPOINT_BUSY	A Write is already in progress.
USB_ENDPOINT_NOT_FOUND	Invalid endpoint.

Function

```
uint8_t USBHostWrite( uint8_t deviceAddress, uint8_t endpoint, uint8_t *data,
                      uint32_t size )
```

1.4.2.1.1.24 USBHostGetCurrentConfigurationDescriptor Macro**File**

usb_host.h

Syntax

```
#define USBHostGetCurrentConfigurationDescriptor( deviceAddress ) ( pCurrentConfigurationDescriptor )
```

Returns

uint8_t * - Pointer to the Configuration Descriptor.

Description

This function returns a pointer to the current configuration descriptor of the requested device.

Remarks

This will need to be expanded to a full function when multiple device support is added.

Preconditions

None

Function

```
uint8_t * USBHostGetCurrentConfigurationDescriptor( uint8_t deviceAddress )
```

1.4.2.1.1.25 USBHostGetDeviceDescriptor Macro**File**

usb_host.h

Syntax

```
#define USBHostGetDeviceDescriptor( deviceAddress ) ( pDeviceDescriptor )
```

Returns

uint8_t * - Pointer to the Device Descriptor.

Description

This function returns a pointer to the device descriptor of the requested device.

Remarks

This will need to be expanded to a full function when multiple device support is added.

Preconditions

None

Function

```
uint8_t * USBHostGetDeviceDescriptor( uint8_t deviceAddress )
```

1.4.2.1.1.26 USBHostGetStringDescriptor Macro

This routine initiates a request to obtains the requested string descriptor.

File

usb_host.h

Syntax

```
#define USBHostGetStringDescriptor( deviceAddress, stringNumber, LangID, stringDescriptor,
stringLength, clientDriverID ) \
    USBHostIssueDeviceRequest( deviceAddress, USB_SETUP_DEVICE_TO_HOST | \
    USB_SETUP_TYPE_STANDARD | USB_SETUP_RECIPIENT_DEVICE, \
        \ \
        USB_REQUEST_GET_DESCRIPTOR, (USB_DESCRIPTOR_STRING << 8) | \
        stringNumber, \
            \ \
            LangID, stringLength, stringDescriptor, USB_DEVICE_REQUEST_GET, \
            clientDriverID )
```

Description

This routine initiates a request to obtain the requested string descriptor. If the request cannot be started, the routine returns an error. Otherwise, the request is started, and the requested string descriptor is stored in the designated location.

Example Usage:

```
USBHostGetStringDescriptor(
    deviceAddress,
    stringDescriptorNum,
    LangID,
    stringDescriptorBuffer,
    sizeof(stringDescriptorBuffer),
    0xFF
);

while(1)
{
    if(USBHostTransferIsComplete( deviceAddress , 0, &errorCode, &byteCount ))
    {
        if(errorCode)
        {
            //There was an error reading the string, bail out of loop
        }
        else
        {
            //String is located in specified buffer, do something with it.

            //The length of the string is both in the byteCount variable
            // as well as the first byte of the string itself
        }
        break;
    }
    USBTasks( );
}
```

Remarks

The returned string descriptor will be in the exact format as obtained from the device. The length of the entire descriptor will be in the first byte, and the descriptor type will be in the second. The string itself is represented in UNICODE. Refer to the USB 2.0 Specification for more information about the format of string descriptors.

Preconditions

None

Parameters

Parameters	Description
deviceAddress	Address of the device
stringNumber	Index of the desired string descriptor
LangID	The Language ID of the string to read (should be 0 if trying to read the language ID list *stringDescriptor - Pointer to where to store the string).
stringLength	Maximum length of the returned string.
clientDriverID	Client driver to return the completion event to.

Return Values

Return Values	Description
USB_SUCCESS	The request was started successfully.
USB_UNKNOWN_DEVICE	Device not found
USB_INVALID_STATE	We must be in a normal running state.
USB_ENDPOINT_BUSY	The endpoint is currently processing a request.

Function

```
uint8_t USBHostGetStringDescriptor ( uint8_t deviceAddress, uint8_t stringNumber,
```

```
uint8_t LangID, uint8_t *stringDescriptor, uint8_t stringLength,
uint8_t clientDriverID )
```

1.4.2.1.1.27 **USBHostReadIsochronous Macro**

This function initiates a read from an isochronous endpoint on the attached device.

File

usb_host.h

Syntax

```
#define USBHostReadIsochronous( a, e, p ) USBHostRead( a, e, (uint8_t *)p, (uint32_t)0 );
```

Description

This function initiates a read from an isochronous endpoint on the attached device. If the endpoint is not isochronous, use USBHostRead().

Once started, an isochronous transfer will continue with no upper layer intervention until USBHostTerminateTransfer() is called.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_SUCCESS	Read started successfully.
USB_UNKNOWN_DEVICE	Device with the specified address not found.
USB_INVALID_STATE	We are not in a normal running state.
USB_ENDPOINT_ILLEGAL_TYPE	Must use USBHostControlRead to read from a control endpoint.
USB_ENDPOINT_ILLEGAL_DIRECTION	Must read from an IN endpoint.
USB_ENDPOINT_STALLED	Endpoint is stalled. Must be cleared by the application.
USB_ENDPOINT_ERROR	Endpoint has too many errors. Must be cleared by the application.
USB_ENDPOINT_BUSY	A Read is already in progress.
USB_ENDPOINT_NOT_FOUND	Invalid endpoint.

Function

```
uint8_t USBHostReadIsochronous( uint8_t deviceAddress, uint8_t endpoint,
ISOCHRONOUS_DATA *pIsochronousData )
```

1.4.2.1.1.28 **USBHostWriteIsochronous Macro**

This function initiates a write to an isochronous endpoint on the attached device.

File

usb_host.h

Syntax

```
#define USBHostWriteIsochronous( a, e, p ) USBHostWrite( a, e, (uint8_t *)p, (uint32_t)0 );
```

Description

This function initiates a write to an isochronous endpoint on the attached device. If the endpoint is not isochronous, use

`USBHostWrite()`.

Once started, an isochronous transfer will continue with no upper layer intervention until `USBHostTerminateTransfer()` is called.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
<code>USB_SUCCESS</code>	Write started successfully.
<code>USB_UNKNOWN_DEVICE</code>	Device with the specified address not found.
<code>USB_INVALID_STATE</code>	We are not in a normal running state.
<code>USB_ENDPOINT_ILLEGAL_TYPE</code>	Must use <code>USBHostControlWrite</code> to write to a control endpoint.
<code>USB_ENDPOINT_ILLEGAL_DIRECTION</code>	Must write to an OUT endpoint.
<code>USB_ENDPOINT_STALLED</code>	Endpoint is stalled. Must be cleared by the application.
<code>USB_ENDPOINT_ERROR</code>	Endpoint has too many errors. Must be cleared by the application.
<code>USB_ENDPOINT_BUSY</code>	A Write is already in progress.
<code>USB_ENDPOINT_NOT_FOUND</code>	Invalid endpoint.

Function

```
uint8_t USBHostWriteIsochronous( uint8_t deviceAddress, uint8_t endpoint,
ISOCHRONOUS_DATA *pIsochronousData )
```

1.4.2.1.2 Data Types and Constants

Macros

Name	Description
<code>INIT_CL_SC_P</code>	Set class support in the TPL (non-OTG only).
<code>INIT_VID_PID</code>	Set VID/PID support in the TPL.
<code>TPL_ALLOW_HNP</code>	Bitmask for Host Negotiation Protocol.
<code>TPL_CLASS_DRV</code>	Bitmask for class driver support.
<code>TPL_EP0_ONLY_CUSTOM_DRIVER</code>	Bitmask to let a custom driver gain EP0 only and allow other interfaces to use standard drivers
<code>TPL_IGNORE_CLASS</code>	Bitmask for ignoring the class of a CL/SC/P driver
<code>TPL_IGNORE_PID</code>	Bitmask for ignoring the PID of a VID/PID driver
<code>TPL_IGNORE_PROTOCOL</code>	Bitmask for ignoring the protocol of a CL/SC/P driver
<code>TPL_IGNORE_SUBCLASS</code>	Bitmask for ignoring the subclass of a CL/SC/P driver
<code>TPL_SET_CONFIG</code>	Bitmask for setting the configuration.
<code>USB_HOST_APP_DATA_EVENT_HANDLER</code>	If the application does not provide an event handler, then we will assume that all events function without error.
<code>USB_HOST_APP_EVENT_HANDLER</code>	If the application does not provide an event handler, then we will assume that all events function without error.
<code>USB_NUM_BULK_NAKS</code>	Define how many NAK's are allowed during a bulk transfer before erroring.
<code>USB_NUM_COMMAND_TRIES</code>	During enumeration, define how many times each command will be tried before giving up and resetting the device.
<code>USB_NUM_CONTROL_NAKS</code>	Define how many NAK's are allowed during a control transfer before erroring.

USB_NUM_ENUMERATION_TRIES	Define how many times the host will try to enumerate the device before giving up and setting the state to DETACHED.
USB_NUM_INTERRUPT_NAKS	Define how many NAK's are allowed during an interrupt OUT transfer before erroring. Interrupt IN transfers that are NAK'd are terminated without error.

Module

Embedded Host Stack

Structures

	Name	Description
❖	_CLIENT_DRIVER_TABLE	Client Driver Table Structure This structure is used to define an entry in the client-driver table. Each entry provides the information that the Host layer needs to manage a particular USB client driver, including pointers to the interface routines that the Client Driver must implement.
❖	_HOST_TRANSFER_DATA	Host Transfer Information This structure is used when the event handler is used to notify the upper layer of transfer completion.
	CLIENT_DRIVER_TABLE	Client Driver Table Structure This structure is used to define an entry in the client-driver table. Each entry provides the information that the Host layer needs to manage a particular USB client driver, including pointers to the interface routines that the Client Driver must implement.
	HOST_TRANSFER_DATA	Host Transfer Information This structure is used when the event handler is used to notify the upper layer of transfer completion.

Types

Name	Description
TRANSFER_ATTRIBUTES	This is type TRANSFER_ATTRIBUTES.
USB_CLIENT_EVENT_HANDLER	This is a typedef to use when defining a client driver event handler.
USB_CLIENT_INIT	This is a typedef to use when defining a client driver initialization handler.
USB_TPL	Targeted Peripheral List This structure is used to define the devices that this host can support. If the host is a USB Embedded Host or Dual Role Device that does not support OTG, the TPL may contain both specific devices and generic classes. If the host supports OTG, then the TPL may contain ONLY specific devices.

Description**1.4.2.1.2.1 CLIENT_DRIVER_TABLE Structure****File**

usb_host.h

Syntax

```
typedef struct _CLIENT_DRIVER_TABLE {
    USB_CLIENT_INIT Initialize;
    USB_CLIENT_EVENT_HANDLER EventHandler;
    USB_CLIENT_EVENT_HANDLER DataEventHandler;
    uint32_t flags;
} CLIENT_DRIVER_TABLE;
```

Members

Members	Description
USB_CLIENT_INIT Initialize;	Initialization routine
USB_CLIENT_EVENT_HANDLER EventHandler;	Event routine
USB_CLIENT_EVENT_HANDLER DataEventHandler;	Data Event routine
uint32_t flags;	Initialization flags

Description

Client Driver Table Structure

This structure is used to define an entry in the client-driver table. Each entry provides the information that the Host layer needs to manage a particular USB client driver, including pointers to the interface routines that the Client Driver must implement.

1.4.2.1.2.2 HOST_TRANSFER_DATA Structure**File**

usb_host.h

Syntax

```
typedef struct _HOST_TRANSFER_DATA {
    uint32_t dataCount;
    uint8_t * pUserData;
    uint8_t bEndpointAddress;
    uint8_t bErrorCode;
    TRANSFER_ATTRIBUTES bmAttributes;
    uint8_t clientDriver;
} HOST_TRANSFER_DATA;
```

Members

Members	Description
uint32_t dataCount;	Count of bytes transferred.
uint8_t * pUserData;	Pointer to transfer data.
uint8_t bEndpointAddress;	Transfer endpoint.
uint8_t bErrorCode;	Transfer error code.
TRANSFER_ATTRIBUTES bmAttributes;	INTERNAL USE ONLY - Endpoint transfer attributes.
uint8_t clientDriver;	INTERNAL USE ONLY - Client driver index for sending the event.

Description

Host Transfer Information

This structure is used when the event handler is used to notify the upper layer of transfer completion.

1.4.2.1.2.3 TRANSFER_ATTRIBUTES Type**File**

usb_host.h

Syntax

```
typedef #warning USB_INITIAL_VBUS_CURRENT is in violation of the USB specification.
#warning USB_INITIAL_VBUS_CURRENT is in violation of the USB specification. union
TRANSFER_ATTRIBUTES@1 TRANSFER_ATTRIBUTES;
```

Description

This is type TRANSFER_ATTRIBUTES.

1.4.2.1.2.4 USB_CLIENT_EVENT_HANDLER Type

This is a typedef to use when defining a client driver event handler.

File

usb_host.h

Syntax

```
typedef bool (* USB_CLIENT_EVENT_HANDLER)(uint8_t address, USB_EVENT event, void *data,  
uint32_t size);
```

Description

This data type defines a pointer to a call-back function that must be implemented by a client driver if it needs to be aware of events on the USB. When an event occurs, the Host layer will call the client driver via this pointer to handle the event. Events are identified by the "event" parameter and may have associated data. If the client driver was able to handle the event, it should return true. If not (or if additional processing is required), it should return false.

Remarks

The application may also implement an event handling routine if it requires knowledge of events. To do so, it must implement a routine that matches this function signature and define the **USB_HOST_APP_EVENT_HANDLER** macro as the name of that function.

Preconditions

The client must have been initialized.

Return Values

Return Values	Description
true	The event was handled
false	The event was not handled

Function

```
bool (*USB_CLIENT_EVENT_HANDLER) ( uint8_t address, USB_EVENT event,  
void *data, uint32_t size )
```

1.4.2.1.2.5 USB_CLIENT_INIT Type

This is a typedef to use when defining a client driver initialization handler.

File

usb_host.h

Syntax

```
typedef bool (* USB_CLIENT_INIT)(uint8_t address, uint32_t flags, uint8_t clientDriverID);
```

Description

This routine is a call out from the host layer to a USB client driver. It is called when the system has been configured as a USB host and a new device has been attached to the bus. Its purpose is to initialize and activate the client driver.

Remarks

There may be multiple client drivers. If so, the USB host layer will call the initialize routine for each of the clients that are in the selected configuration.

Preconditions

The device has been configured.

Return Values

Return Values	Description
true	Successful
false	Not successful

Function

```
bool (*USB_CLIENT_INIT) ( uint8_t address, uint32_t flags, uint8_t clientDriverID )
```

1.4.2.1.2.6 USB_TPL Type**File**

usb_host.h

Syntax

```
typedef struct _USB_TPL USB_TPL;
```

Description

Targeted Peripheral List

This structure is used to define the devices that this host can support. If the host is a USB Embedded Host or Dual Role Device that does not support OTG, the TPL may contain both specific devices and generic classes. If the host supports OTG, then the TPL may contain ONLY specific devices.

1.4.2.1.2.7 INIT_CL_SC_P Macro**File**

usb_host.h

Syntax

```
#define INIT_CL_SC_P(c,s,p) {((c)|(((s)<<8)|((p)<<16))) // Set class support in the TPL  
(non-OTG only).
```

Description

Set class support in the TPL (non-OTG only).

1.4.2.1.2.8 INIT_VID_PID Macro**File**

usb_host.h

Syntax

```
#define INIT_VID_PID(v,p) {((v)|(((p)<<16))) // Set VID/PID support in the TPL.
```

Description

Set VID/PID support in the TPL.

1.4.2.1.2.9 TPL_ALLOW_HNP Macro**File**

usb_host.h

Syntax

```
#define TPL_ALLOW_HNP 0x01 // Bitmask for Host Negotiation Protocol.
```

Description

Bitmask for Host Negotiation Protocol.

1.4.2.1.2.10 TPL_CLASS_DRV Macro

File

usb_host.h

Syntax

```
#define TPL_CLASS_DRV 0x02 // Bitmask for class driver support.
```

Description

Bitmask for class driver support.

1.4.2.1.2.11 TPL_EP0_ONLY_CUSTOM_DRIVER Macro

File

usb_host.h

Syntax

```
#define TPL_EP0_ONLY_CUSTOM_DRIVER 0x80 // Bitmask to let a custom driver gain EP0 only and allow other interfaces to use standard drivers
```

Description

Bitmask to let a custom driver gain EP0 only and allow other interfaces to use standard drivers

1.4.2.1.2.12 TPL_IGNORE_CLASS Macro

File

usb_host.h

Syntax

```
#define TPL_IGNORE_CLASS 0x20 // Bitmask for ignoring the class of a CL/SC/P driver
```

Description

Bitmask for ignoring the class of a CL/SC/P driver

1.4.2.1.2.13 TPL_IGNORE_PID Macro

File

usb_host.h

Syntax

```
#define TPL_IGNORE_PID 0x40 // Bitmask for ignoring the PID of a VID/PID driver
```

Description

Bitmask for ignoring the PID of a VID/PID driver

1.4.2.1.2.14 TPL_IGNORE_PROTOCOL Macro

File

usb_host.h

Syntax

```
#define TPL_IGNORE_PROTOCOL 0x08 // Bitmask for ignoring the protocol of a CL/SC/P driver
```

Description

Bitmask for ignoring the protocol of a CL/SC/P driver

1.4.2.1.2.15 TPL_IGNORE_SUBCLASS Macro

File

usb_host.h

Syntax

```
#define TPL_IGNORE_SUBCLASS 0x10          // Bitmask for ignoring the subclass of  
a CL/SC/P driver
```

Description

Bitmask for ignoring the subclass of a CL/SC/P driver

1.4.2.1.2.16 TPL_SET_CONFIG Macro

File

usb_host.h

Syntax

```
#define TPL_SET_CONFIG 0x04           // Bitmask for setting the configuration.
```

Description

Bitmask for setting the configuration.

1.4.2.1.2.17 USB_HOST_APP_DATA_EVENT_HANDLER Macro

File

usb_host.h

Syntax

```
#define USB_HOST_APP_DATA_EVENT_HANDLER(a,e,d,s) true
```

Description

If the application does not provide an event handler, then we will assume that all events function without error.

1.4.2.1.2.18 USB_HOST_APP_EVENT_HANDLER Macro

File

usb_host.h

Syntax

```
#define USB_HOST_APP_EVENT_HANDLER(a,e,d,s)  
( (e==EVENT_OVERRIDE_CLIENT_DRIVER_SELECTION)?false:true)
```

Description

If the application does not provide an event handler, then we will assume that all events function without error.

1.4.2.1.2.19 USB_NUM_BULK_NAKS Macro

File

usb_host.h

Syntax

```
#define USB_NUM_BULK_NAKS 10000    // Define how many NAK's are allowed
```

Description

Define how many NAK's are allowed during a bulk transfer before erroring.

1.4.2.1.2.20 USB_NUM_COMMAND_TRIES Macro

File

usb_host.h

Syntax

```
#define USB_NUM_COMMAND_TRIES 3 // During enumeration, define how many
```

Description

During enumeration, define how many times each command will be tried before giving up and resetting the device.

1.4.2.1.2.21 USB_NUM_CONTROL_NAKS Macro

File

usb_host.h

Syntax

```
#define USB_NUM_CONTROL_NAKS 20 // Define how many NAK's are allowed
```

Description

Define how many NAK's are allowed during a control transfer before erroring.

1.4.2.1.2.22 USB_NUM_ENUMERATION_TRIES Macro

File

usb_host.h

Syntax

```
#define USB_NUM_ENUMERATION_TRIES 3 // Define how many times the host will try
```

Description

Define how many times the host will try to enumerate the device before giving up and setting the state to DETACHED.

1.4.2.1.2.23 USB_NUM_INTERRUPT_NAKS Macro

File

usb_host.h

Syntax

```
#define USB_NUM_INTERRUPT_NAKS 3 // Define how many NAK's are allowed
```

Description

Define how many NAK's are allowed during an interrupt OUT transfer before erroring. Interrupt IN transfers that are NAK'd are terminated without error.

1.4.2.1.3 usb_host.h

Functions

	Name	Description
💡	USB_HOST_APP_DATA_EVENT_HANDLER	This is a typedef to use when defining the application level data events handler.
💡	USB_HOST_APP_EVENT_HANDLER	This is a typedef to use when defining the application level events handler.
💡	USB_HostInterruptHandler	This function handles the interrupts when the USB module is running in host mode.

	USBHostClearEndpointErrors	This function clears an endpoint's internal error condition.
	USBHostDeviceSpecificClientDriver	This function indicates if the specified device has explicit client driver support specified in the TPL.
	USBHostDeviceStatus	This function returns the current status of a device.
	USBHostInit	This function initializes the variables of the USB host stack.
	USBHostIsochronousBuffersCreate	This function initializes the isochronous data buffer information and allocates memory for each buffer. This function will not allocate memory if the buffer pointer is not NULL.
	USBHostIsochronousBuffersDestroy	This function releases all of the memory allocated for the isochronous data buffers. It also resets all other information about the buffers.
	USBHostIsochronousBuffersReset	This function resets all the isochronous data buffers. It does not do anything with the space allocated for the buffers.
	USBHostIssueDeviceRequest	This function sends a standard device request to the attached device.
	USBHostRead	This function initiates a read from the attached device.
	USBHostResetDevice	This function resets an attached device.
	USBHostResumeDevice	This function issues a RESUME to the attached device.
	USBHostSetDeviceConfiguration	This function changes the device's configuration.
	USBHostSetNAKTimeout	This function specifies NAK timeout capability.
	USBHostShutdown	This function turns off the USB module and frees all unnecessary memory. This routine can be called by the application layer to shut down all USB activity, which effectively detaches all devices. The event EVENT_DETACH will be sent to the client drivers for the attached device, and the event EVENT_VBUS_RELEASE_POWER will be sent to the application layer.
	USBHostSuspendDevice	This function suspends a device.
	USBHostTasks	This function executes the host tasks for USB host operation.
	USBHostTerminateTransfer	This function terminates the current transfer for the given endpoint.
	USBHostTransferIsComplete	This function initiates whether or not the last endpoint transaction is complete.
	USBHostVbusEvent	This function handles Vbus events that are detected by the application.
	USBHostWrite	This function initiates a write to the attached device.

Macros

Name	Description
<code>__USBHOST_H__</code>	DOM-IGNORE-END
<code>INIT_CL_SC_P</code>	Set class support in the TPL (non-OTG only).
<code>INIT_VID_PID</code>	Set VID/PID support in the TPL.
<code>TPL_ALLOW_HNP</code>	Bitmask for Host Negotiation Protocol.
<code>TPL_CLASS_DRV</code>	Bitmask for class driver support.
<code>TPL_EP0_ONLY_CUSTOM_DRIVER</code>	Bitmask to let a custom driver gain EP0 only and allow other interfaces to use standard drivers
<code>TPL_IGNORE_CLASS</code>	Bitmask for ignoring the class of a CL/SC/P driver
<code>TPL_IGNORE_PID</code>	Bitmask for ignoring the PID of a VID/PID driver
<code>TPL_IGNORE_PROTOCOL</code>	Bitmask for ignoring the protocol of a CL/SC/P driver
<code>TPL_IGNORE_SUBCLASS</code>	Bitmask for ignoring the subclass of a CL/SC/P driver
<code>TPL_SET_CONFIG</code>	Bitmask for setting the configuration.
<code>USB_HOST_APP_DATA_EVENT_HANDLER</code>	If the application does not provide an event handler, then we will assume that all events function without error.

USB_HOST_APP_EVENT_HANDLER	If the application does not provide an event handler, then we will assume that all events function without error.
USB_NUM_BULK_NAKS	Define how many NAK's are allowed during a bulk transfer before erroring.
USB_NUM_COMMAND_TRIES	During enumeration, define how many times each command will be tried before giving up and resetting the device.
USB_NUM_CONTROL_NAKS	Define how many NAK's are allowed during a control transfer before erroring.
USB_NUM_ENUMERATION_TRIES	Define how many times the host will try to enumerate the device before giving up and setting the state to DETACHED.
USB_NUM_INTERRUPT_NAKS	Define how many NAK's are allowed during an interrupt OUT transfer before erroring. Interrupt IN transfers that are NAK'd are terminated without error.
USBHostGetCurrentConfigurationDescriptor	This function returns a pointer to the current configuration descriptor of the requested device.
USBHostGetDeviceDescriptor	This function returns a pointer to the device descriptor of the requested device.
USBHostGetStringDescriptor	This routine initiates a request to obtain the requested string descriptor.
USBHostReadIsochronous	This function initiates a read from an isochronous endpoint on the attached device.
USBHostWriteIsochronous	This function initiates a write to an isochronous endpoint on the attached device.

Module

Embedded Host Stack

Structures

	Name	Description
	_CLIENT_DRIVER_TABLE	Client Driver Table Structure This structure is used to define an entry in the client-driver table. Each entry provides the information that the Host layer needs to manage a particular USB client driver, including pointers to the interface routines that the Client Driver must implement.
	_HOST_TRANSFER_DATA	Host Transfer Information This structure is used when the event handler is used to notify the upper layer of transfer completion.
	CLIENT_DRIVER_TABLE	Client Driver Table Structure This structure is used to define an entry in the client-driver table. Each entry provides the information that the Host layer needs to manage a particular USB client driver, including pointers to the interface routines that the Client Driver must implement.
	HOST_TRANSFER_DATA	Host Transfer Information This structure is used when the event handler is used to notify the upper layer of transfer completion.

Types

Name	Description
TRANSFER_ATTRIBUTES	This is type TRANSFER_ATTRIBUTES.
USB_CLIENT_EVENT_HANDLER	This is a typedef to use when defining a client driver event handler.
USB_CLIENT_INIT	This is a typedef to use when defining a client driver initialization handler.
USB_TPL	Targeted Peripheral List This structure is used to define the devices that this host can support. If the host is a USB Embedded Host or Dual Role Device that does not support OTG, the TPL may contain both specific devices and generic classes. If the host supports OTG, then the TPL may contain ONLY specific devices.

Description

This is file usb_host.h.

1.4.2.1.4 __USBHOST_H__ Macro**File**

usb_host.h

Syntax

```
#define __USBHOST_H__
```

Module

Embedded Host Stack

Description

DOM-IGNORE-END

1.4.2.2 CDC Client Driver

This is a CDC client driver for use with the USB Embedded Host driver.

Files

Name	Description
usb_host_cdc.h	This is file usb_host_cdc.h.
usb_host_cdc_interface.h	This is file usb_host_cdc_interface.h.

Macros

Name	Description
_USB_HOST_CDC_H_	This is macro _USB_HOST_CDC_H_.
_USB_HOST_CDC_INTERFACE_H_	This is macro _USB_HOST_CDC_INTERFACE_H_.

Description**Communication Device Class (CDC) Host****CDC - Overview**

Several type of communication can benefit from USB. Communication Device Class specification provides common specification for communication devices. There are three classes that make up the definition for communications devices:

- * Communications Device Class
- * Communications Interface Class
- * Data Interface Class.

The Communications Device Class is a device-level definition and is used by the host to properly identify a communications device that may present several different types of interfaces.

The Communications Interface Class defines a general-purpose mechanism that can be used to enable all types of communications services on the Universal Serial Bus (USB). This interface consist of two elements, a management element and a notification element. The management element configures and controls the device, it consist of endpoint 0. Notification element is optional and is used to handle transport events. In the current stack notification element is not implemented.

The Data Interface Class defines a general-purpose mechanism to enable bulk or isochronous transfer on the USB when the data does not meet the requirements for any other class. This interface is used to transmit/receive data to/from the device.

The type of endpoints belonging to a Data Class interface are restricted to being either isochronous or bulk, and are expected to exist in pairs of the same type (one In and one Out). Current version of the stack is tested for Bulk transfers.

Class-Specific Codes

This section lists the codes for the Communications Device Class, Communications Interface Class and Data Interface Class, including subclasses and protocols supported in the current version of the stack. The current version of the stack supports RS232 emulation over USB. Below is the list of codes to support this functionality.

The following table defines the Communications Device Class code:

Code	Class
0x02	Communications Device Class

Communication Interface Codes

The following table defines the Communications Class code:

Code	Class
0x02	Communications Interface Class

CDC specification mentions various subclass , current version of the Microchip CDC host stack supports below mentioned subclasses. The following table defines the currently supported Subclass codes for the Communications Interface Class:

Code	SubClass
0x02	Abstract Control Model

The following table defines supported Communications Class Protocol Codes:

Code	Protocol
0x01	AT Commands: V.250 etc.

Data Interface Code

The following table defines the Data Interface Class code:

Code	Class
0x0A	Data Interface Class

No specific Subclass and Protocol codes are required to achieve RS232 functionality over USB.

Communication and Data Transfer Handling

Communication Management : The CDC client driver takes care of enumerating the device connected on the bus. The application must define Line Coding parameters in file `usb_config.h` . `USBConfig` utility can be used to set these parameters. If the connected device complies with the setting then the device is successfully attached else the device is not attached onto the bus. If the application needs to change the setting dynamically after the device has been successfully enumerated , interface function `USBHostCDC_Api_ACN_Request()`can be used to do so. Following standard requests are currently implemented:

Request	Summary
SendEncapsulatedCommand	Issues a command in the format of the supported control protocol.
GetEncapsulatedResponse	Requests a response in the format of the supported control protocol.
SetLineCoding	Configures DTE rate, stop-bits, parity, and number-of-character bits.
GetLineCoding	Requests current DTE rate, stop-bits, parity, and number-of-character bits.

SetControlLineState	[V24] signal used to tell the DCE device the DTE device is now present.
---------------------	---

Data transfers : Once the device is attached the application is ready to start data transfers. Usually two endpoints one in each direction are supported by the device.

- * To receive data from the device the application must set up a IN request at the rate depending on the baudrate settings. Application can use a timer interrupt to precisely set up the request. Function USBHostCDC_Api_Get_IN_Data() is used to setup the request. Maximum of 64 bytes can be received in single transfer.
- * To transmit data to the device application must set up a OUT request. Function USBHostCDC_Api_Send_OUT_Data() is used to setup out request. Any amount of data can be transferred to the device. The Client driver takes care of sending the data in 64 bytes packet.
- * USBHostCDC_ApiTransferIsComplete() is used to poll for the status of previous transfer.
- * USBHostCDC_ApiDeviceDetect() is used to get the status of the device. If the device is ready for new transfer then the function returns TRUE.

1.4.2.2.1 Functions

Functions

Name	Description
USBHostCDC_Api_ACN_Request	This function can be used by application code to dynamically access ACM specific requests. This function should be used only if application intends to modify for example the Baudrate from previously configured rate. Data transmitted/received to/from device is a array of bytes. Application must take extra care of understanding the data format before using this function.
USBHostCDC_Api_Get_IN_Data	This function is called by application to receive Input data over DATA interface. This function sets up the request to receive data from the device.
USBHostCDC_Api_Send_OUT_Data	This function is called by application to transmit out data over DATA interface. This function sets up the request to transmit data to the device.
USBHostCDC_ApiDeviceDetect	This function determines if a CDC device is attached and ready to use.
USBHostCDC_ApiTransferIsComplete	This function is called by application to poll for transfer status. This function returns true in the transfer is over. To check whether the transfer was successfull or not , application must check the error code returned by reference.
USBHostCDCDeviceStatus	This function determines the status of a CDC device.
USBHostCDCEventHandler	This function is the event handler for this client driver.
USBHostCDCInitAddress	This function initializes the address of the attached CDC device.
USBHostCDCInitialize	This function is the initialization routine for this client driver.
USBHostCDCResetDevice	This function starts a CDC reset.
USBHostCDCTasks	This function performs the maintenance tasks required by CDC class
USBHostCDCTransfer	This function starts a CDC transfer.
USBHostCDCTransferIsComplete	This function indicates whether or not the last transfer is complete.

Module

CDC Client Driver

Description

1.4.2.2.1.1 USBHostCDC_Api_ACN_Request Function

File

usb_host_cdc_interface.h

Syntax

```
uint8_t USBHostCDC_Api_ACN_Request(uint8_t requestType, uint8_t size, uint8_t* data);
```

Description

This function can be used by application code to dynamically access ACM specific requests. This function should be used only if application intends to modify for example the Baudrate from previously configured rate. Data transmitted/received to/from device is a array of bytes. Application must take extra care of understanding the data format before using this function.

Remarks

None

Preconditions

Device must be enumerated and attached successfully.

Return Values

Return Values	Description
USB_SUCCESS	Request started successfully
USB_CDC_DEVICE_NOT_FOUND	No device with specified address
USB_CDC_DEVICE_BUSY	Device not in proper state for performing a transfer
USB_CDC_COMMAND_FAILED	Request is not supported.
USB_CDC_ILLEGAL_REQUEST	Requested ID is invalid.

Function

```
uint8_t USBHostCDC_Api_ACN_Request(uint8_t requestType, uint8_t size, uint8_t* data)
```

1.4.2.2.1.2 USBHostCDC_Api_Get_IN_Data Function

File

usb_host_cdc_interface.h

Syntax

```
bool USBHostCDC_Api_Get_IN_Data(uint8_t no_of_bytes, uint8_t* data);
```

Description

This function is called by application to receive Input data over DATA interface. This function sets up the request to receive data from the device.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
TRUE	Transfer request is placed successfully.
FALSE	Transfer request failed.

Function

```
bool USBHostCDC_Api_Get_IN_Data(uint8_t no_of_bytes, uint8_t* data)
```

1.4.2.2.1.3 USBHostCDC_Api_Send_OUT_Data Function

File

usb_host_cdc_interface.h

Syntax

```
bool USBHostCDC_Api_Send_OUT_Data(uint16_t no_of_bytes, uint8_t* data);
```

Description

This function is called by application to transmit out data over DATA interface. This function sets up the request to transmit data to the device.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
TRUE	Transfer request is placed successfully.
FALSE	Transfer request failed.

Function

```
bool USBHostCDC_Api_Send_OUT_Data(uint16_t no_of_bytes, uint8_t* data)
```

1.4.2.2.1.4 USBHostCDC_ApiDeviceDetect Function

File

usb_host_cdc_interface.h

Syntax

```
bool USBHostCDC_ApiDeviceDetect();
```

Description

This function determines if a CDC device is attached and ready to use.

Remarks

Since this will often be called in a loop while waiting for a device, we'll make sure the tasks are executed.

Preconditions

None

Return Values

Return Values	Description
TRUE	CDC present and ready
FALSE	CDC not present or not ready

Function

```
bool USBHostCDC_ApiDeviceDetect( void )
```

1.4.2.2.1.5 USBHostCDC_ApiTransferIsComplete Function

File

usb_host_cdc_interface.h

Syntax

```
bool USBHostCDC_ApiTransferIsComplete(uint8_t* errorCodeDriver, uint8_t* byteCount);
```

Description

This function is called by application to poll for transfer status. This function returns true in the transfer is over. To check whether the transfer was successfull or not , application must check the error code returned by reference.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
TRUE	Transfer is has completed.
FALSE	Transfer is pending.

Function

```
bool USBHostCDC_ApiTransferIsComplete(uint8_t* errorCodeDriver,uint8_t* byteCount)
```

1.4.2.2.1.6 USBHostCDCDeviceStatus Function

This function determines the status of a CDC device.

File

usb_host_cdc.h

Syntax

```
uint8_t USBHostCDCDeviceStatus(uint8_t deviceAddress);
```

Description

This function determines the status of a CDC device.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_CDC_DEVICE_NOT_FOUND	Illegal device address, or the device is not an CDC
USB_CDC_INITIALIZING	CDC is attached and in the process of initializing
USB_PROCESSING_REPORT_DESCRIPTOR	CDC device is detected and report descriptor is being parsed
USB_CDC_NORMAL_RUNNING	CDC Device is running normal, ready to send and receive reports
USB_CDC_DEVICE_HOLDING	Device is holding due to error
USB_CDC_DEVICE_DETACHED	CDC detached.

Function

```
uint8_t USBHostCDCDeviceStatus( uint8_t deviceAddress )
```

1.4.2.2.1.7 USBHostCDCEventHandler Function

This function is the event handler for this client driver.

File

usb_host_cdc.h

Syntax

```
bool USBHostCDCEventHandler(uint8_t address, USB_EVENT event, void * data, uint32_t size);
```

Description

This function is the event handler for this client driver. It is called by the host layer when various events occur.

Remarks

None

Preconditions

The device has been initialized.

Return Values

Return Values	Description
TRUE	Event was handled
FALSE	Event was not handled

Function

```
bool USBHostCDCEventHandler( uint8_t address, USB_EVENT event,  
    void *data, uint32_t size )
```

1.4.2.2.1.8 USBHostCDCInitAddress Function

This function initializes the address of the attached CDC device.

File

usb_host_cdc.h

Syntax

```
bool USBHostCDCInitAddress(uint8_t address, uint32_t flags, uint8_t clientDriverID);
```

Description

This function initializes the address of the attached CDC device. Once the device is enumerated without any errors, the CDC client call this function. For all the transfer requests this address is used to identify the CDC device.

Remarks

None

Preconditions

The device has been enumerated without any errors.

Return Values

Return Values	Description
TRUE	We can support the device.
FALSE	We cannot support the device.

Function

```
bool USBHostCDCInitAddress( uint8_t address, uint32_t flags, uint8_t clientDriverID )
```

1.4.2.2.1.9 USBHostCDCInitialize Function

This function is the initialization routine for this client driver.

File

usb_host_cdc.h

Syntax

```
bool USBHostCDCInitialize(uint8_t address, uint32_t flags, uint8_t clientDriverID);
```

Description

This function is the initialization routine for this client driver. It is called by the host layer when the USB device is being enumerated. For a CDC device we need to look into CDC descriptor, interface descriptor and endpoint descriptor.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
TRUE	We can support the device.
FALSE	We cannot support the device.

Function

```
bool USBHostCDCInitialize( uint8_t address, uint32_t flags, uint8_t clientDriverID )
```

1.4.2.2.1.10 USBHostCDCResetDevice Function

This function starts a CDC reset.

File

usb_host_cdc.h

Syntax

```
uint8_t USBHostCDCResetDevice(uint8_t deviceAddress);
```

Description

This function starts a CDC reset. A reset can be issued only if the device is attached and not being initialized.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_SUCCESS	Reset started
USB_MSD_DEVICE_NOT_FOUND	No device with specified address
USB_MSD_ILLEGAL_REQUEST	Device is in an illegal state for reset

Function

```
uint8_t USBHostCDCResetDevice( uint8_t deviceAddress )
```

1.4.2.2.1.11 USBHostCDCTasks Function

This function performs the maintenance tasks required by CDC class

File

usb_host_cdc.h

Syntax

```
void USBHostCDCTasks( );
```

Returns

None

Description

This function performs the maintenance tasks required by the CDC class. If transfer events from the host layer are not being used, then it should be called on a regular basis by the application. If transfer events from the host layer are being used, this function is compiled out, and does not need to be called.

Remarks

None

Preconditions

USBHostCDCInitialize() has been called.

Parameters

Parameters	Description
None	None

Function

```
void USBHostCDCTasks( void )
```

1.4.2.2.1.12 USBHostCDCTransfer Function

This function starts a CDC transfer.

File

usb_host_cdc.h

Syntax

```
uint8_t USBHostCDCTransfer(uint8_t deviceAddress, uint8_t request, uint8_t direction,
                           uint8_t interfaceNum, uint16_t size, uint8_t * data, uint8_t endpointDATA);
```

Description

This function starts a CDC transfer. A read/write wrapper is provided in application interface file to access this function.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_SUCCESS	Request started successfully
USB_CDC_DEVICE_NOT_FOUND	No device with specified address
USB_CDC_DEVICE_BUSY	Device not in proper state for performing a transfer

Function

```
USBHostCDCTransfer( uint8_t deviceAddress, uint8_t direction, uint8_t reportid, uint8_t size, uint8_t *data)
```

1.4.2.2.1.13 USBHostCDCTransferIsComplete Function

This function indicates whether or not the last transfer is complete.

File

usb_host_cdc.h

Syntax

```
bool USBHostCDCTransferIsComplete(uint8_t deviceAddress, uint8_t * errorCode, uint8_t * uint8_tCount);
```

Description

This function indicates whether or not the last transfer is complete. If the functions returns TRUE, the returned uint8_t count and error code are valid. Since only one transfer can be performed at once and only one endpoint can be used, we only need to know the device address.

Preconditions

None

Return Values

Return Values	Description
TRUE	Transfer is complete, errorCode is valid
FALSE	Transfer is not complete, errorCode is not valid

Function

```
bool USBHostCDCTransferIsComplete( uint8_t deviceAddress,
                                    uint8_t *errorCode, uint32_t *uint8_tCount )
```

1.4.2.2 Data Types and Constants

Macros

Name	Description
DEVICE_CLASS_CDC	CDC Interface Class Code
EVENT_CDC_ATTACH	No event occurred (NULL event)
EVENT_CDC_COMM_READ_DONE	A CDC Communication Read transfer has completed
EVENT_CDC_COMM_WRITE_DONE	A CDC Communication Write transfer has completed
EVENT_CDC_DATA_READ_DONE	A CDC Data Read transfer has completed
EVENT_CDC_DATA_WRITE_DONE	A CDC Data Write transfer has completed
EVENT_CDC_NAK_TIMEOUT	CDC device NAK timeout has occurred
EVENT_CDC_NONE	No event occurred (NULL event)
EVENT_CDC_OFFSET	If the application has not defined an offset for CDC events, set it to 0.
EVENT_CDC_RESET	CDC reset complete
USB_CDC_ABSTRACT_CONTROL_MODEL	Abstract Control Model
USB_CDC_ATM_NETWORKING_CONTROL_MODEL	ATM Networking Control Model
USB_CDC_CAPI_CONTROL_MODEL	CAPI Control Model
USB_CDC_CLASS_ERROR	CDC Class Error Codes
USB_CDC_COMM_INTF	Communication Interface Class Code
USB_CDC_COMMAND_FAILED	Command failed at the device.
USB_CDC_COMMAND_PASSED	Command was successful.
USB_CDC_CONTROL_LINE_LENGTH	Number of uint8_ts Control line transfer
USB_CDC_CS_ENDPOINT	This is macro USB_CDC_CS_ENDPOINT.

USB_CDC_CS_INTERFACE	Functional Descriptor Details Type Values for the bDscType Field
USB_CDC_DATA_INTF	Data Interface Class Codes
USB_CDC_DEVICE_BUSY	A transfer is currently in progress.
USB_CDC_DEVICE_DETACHED	Device is detached.
USB_CDC_DEVICE_HOLDING	Device is holding due to error
USB_CDC_DEVICE_MANAGEMENT	Device Management
USB_CDC_DEVICE_NOT_FOUND	Device with the specified address is not available.
USB_CDC_DIRECT_LINE_CONTROL_MODEL	Direct Line Control Model
USB_CDC_DSC_FN_ACM	ACM - Abstract Control Management
USB_CDC_DSC_FN_CALL_MGT	This is macro USB_CDC_DSC_FN_CALL_MGT.
USB_CDC_DSC_FN_COUNTRY_SELECTION	This is macro USB_CDC_DSC_FN_COUNTRY_SELECTION.
USB_CDC_DSC_FN_DLM	DLM - Direct Line Management
USB_CDC_DSC_FN_HEADER	bDscSubType in Functional Descriptors
USB_CDC_DSC_FN_RPT_CAPABILITIES	This is macro USB_CDC_DSC_FN_RPT_CAPABILITIES.
USB_CDC_DSC_FN_TEL_OP_MODES	This is macro USB_CDC_DSC_FN_TEL_OP_MODES.
USB_CDC_DSC_FN_TELEPHONE_RINGER	This is macro USB_CDC_DSC_FN_TELEPHONE_RINGER.
USB_CDC_DSC_FN_UNION	This is macro USB_CDC_DSC_FN_UNION.
USB_CDC_DSC_FN_USB_TERMINAL	This is macro USB_CDC_DSC_FN_USB_TERMINAL.
USB_CDC_ETHERNET_EMULATION_MODEL	Ethernet Emulation Model
USB_CDC_ETHERNET_NETWORKING_CONTROL_MODEL	Ethernet Networking Control Model
USB_CDC_GET_COMM_FEATURE	Returns the current settings for the communications feature.
USB_CDC_GET_ENCAPSULATED_REQUEST	Requests a response in the format of the supported control protocol.
USB_CDC_GET_LINE_CODING	Requests current DTE rate, stop-bits, parity, and number-of-character bits.
USB_CDC_ILLEGAL_REQUEST	Cannot perform requested operation.
USB_CDC_INITIALIZING	Device is initializing.
USB_CDC_INTERFACE_ERROR	The interface layer cannot support the device.
USB_CDC_LINE_CODING_LENGTH	Number of uint8_ts Line Coding transfer
USB_CDC_MAX_PACKET_SIZE	Max transfer size is 64 uint8_ts for Full Speed USB
USB_CDC_MOBILE_DIRECT_LINE_MODEL	Mobile Direct Line Model
USB_CDC_MULTI_CHANNEL_CONTROL_MODEL	Multi-Channel Control Model
USB_CDC_NO_PROTOCOL	No class specific protocol required For more.... see Table 7 in USB CDC Specification 1.2
USB_CDC_NO_REPORT_DESCRIPTOR	No report descriptor found
USB_CDC_NORMAL_RUNNING	Device is running and available for data transfers.
USB_CDC_OBEX	OBEX
USB_CDC_PHASE_ERROR	Command had a phase error at the device.
USB_CDC_REPORT_DESCRIPTOR_BAD	Report Descriptor for not proper
USB_CDC_RESET_ERROR	An error occurred while resetting the device.
USB_CDC_RESETTING_DEVICE	Device is being reset.
USB_CDC_SEND_BREAK	Sends special carrier modulation used to specify [V24] style break.
USB_CDC_SEND_ENCAPSULATED_COMMAND	Issues a command in the format of the supported control protocol.
USB_CDC_SET_COMM_FEATURE	Controls the settings for a particular communications feature.

USB_CDC_SET_CONTROL_LINE_STATE	V24] signal used to tell the DCE device the DTE device is now present.
USB_CDC_SET_LINE_CODING	Configures DTE rate, stop-bits, parity, and number-of-character bits.
USB_CDC_TELEPHONE_CONTROL_MODEL	Telephone Control Model
USB_CDC_V25TER	Common AT commands ("Hayes(TM)")
USB_CDC_WIRELESS_HANDSET_CONTROL_MODEL	Wireless Handset Control Model

Module

CDC Client Driver

Structures

	Name	Description
◆	_COMM_INTERFACE_DETAILS	This structure stores communication interface details of the attached CDC device
◆	_DATA_INTERFACE_DETAILS	This structure stores data interface details of the attached CDC device
	COMM_INTERFACE_DETAILS	This structure stores communication interface details of the attached CDC device
	DATA_INTERFACE_DETAILS	This structure stores data interface details of the attached CDC device
	USB_CDC_ACM_FN_DSC	Abstract Control Management Functional Descriptor
	USB_CDC_CALL_MGT_FN_DSC	Call Management Functional Descriptor
	USB_CDC_DEVICE_INFO	This structure is used to hold information about an attached CDC device
◆	_USB_CDC_ACM_FN_DSC	Abstract Control Management Functional Descriptor
	USB_CDC_HEADER_FN_DSC	Header Functional Descriptor
◆	_USB_CDC_CALL_MGT_FN_DSC	Call Management Functional Descriptor
	USB_CDC_UNION_FN_DSC	Union Functional Descriptor
◆	_USB_CDC_DEVICE_INFO	This structure is used to hold information about an attached CDC device
◆	_USB_CDC_HEADER_FN_DSC	Header Functional Descriptor
◆	_USB_CDC_UNION_FN_DSC	Union Functional Descriptor

Unions

	Name	Description
	USB_CDC_CONTROL_SIGNAL_BITMAP	This is type USB_CDC_CONTROL_SIGNAL_BITMAP.
	USB_CDC_LINE_CODING	This is type USB_CDC_LINE_CODING.
◆	_USB_CDC_CONTROL_SIGNAL_BITMAP	This is type USB_CDC_CONTROL_SIGNAL_BITMAP.
◆	_USB_CDC_LINE_CODING	This is type USB_CDC_LINE_CODING.

Description**1.4.2.2.2.1 COMM_INTERFACE_DETAILS Structure****File**

usb_host_cdc.h

Syntax

```
typedef struct _COMM_INTERFACE_DETAILS {
    uint8_t interfaceNum;
    uint8_t noOfEndpoints;
    USB_CDC_HEADER_FN_DSC Header_Fn_Dsc;
    USB_CDC_ACM_FN_DSC ACM_Fn_Desc;
    USB_CDC_UNION_FN_DSC Union_Fn_Desc;
    USB_CDC_CALL_MGT_FN_DSC Call_Mgt_Fn_Desc;
    uint16_t endpointMaxDataSize;
    uint16_t endpointInDataSize;
```

```

    uint16_t endpointOutDataSize;
    uint8_t endpointPollInterval;
    uint8_t endpointType;
    uint8_t endpointIN;
    uint8_t endpointOUT;
} COMM_INTERFACE_DETAILS;

```

Members

Members	Description
uint8_t interfaceNum;	communication interface number
uint8_t noOfEndpoints;	Number endpoints for communication interface Functional Descriptor Details
USB_CDC_HEADER_FN_DSC Header_Fn_Dsc;	Header Function Descriptor
USB_CDC_ACM_FN_DSC ACM_Fn_Desc;	Abstract Control Model Function Descriptor
USB_CDC_UNION_FN_DSC Union_Fn_Desc;	Union Function Descriptor
USB_CDC_CALL_MGT_FN_DSC Call_Mgt_Fn_Desc;	Call Management Function Descriptor Endpoint Descriptor Details
uint16_t endpointMaxDataSize;	Max data size for a interface.
uint16_t endpointInDataSize;	Max data size for a interface.
uint16_t endpointOutDataSize;	Max data size for a interface.
uint8_t endpointPollInterval;	Polling rate of corresponding interface.
uint8_t endpointType;	Endpoint type - either Isochronous or Bulk
uint8_t endpointIN;	IN endpoint for comm interface.
uint8_t endpointOUT;	IN endpoint for comm interface.

Description

This structure stores communication interface details of the attached CDC device

1.4.2.2.2 DATA_INTERFACE_DETAILS Structure**File**

usb_host_cdc.h

Syntax

```

typedef struct _DATA_INTERFACE_DETAILS {
    uint8_t interfaceNum;
    uint8_t noOfEndpoints;
    uint16_t endpointInDataSize;
    uint16_t endpointOutDataSize;
    uint8_t endpointType;
    uint8_t endpointIN;
    uint8_t endpointOUT;
} DATA_INTERFACE_DETAILS;

```

Members

Members	Description
uint8_t interfaceNum;	Data interface number
uint8_t noOfEndpoints;	number of endpoints associated with data interface
uint16_t endpointInDataSize;	Max data size for a interface.
uint16_t endpointOutDataSize;	Max data size for a interface.
uint8_t endpointType;	Endpoint type - either Isochronous or Bulk
uint8_t endpointIN;	IN endpoint for comm interface.
uint8_t endpointOUT;	IN endpoint for comm interface.

Description

This structure stores data interface details of the attached CDC device

1.4.2.2.2.3 USB_CDC_ACN_FN_DSC Structure

File

usb_host_cdc.h

Syntax

```
typedef struct _USB_CDC_ACN_FN_DSC {
    uint8_t bFNLength;
    uint8_t bDscType;
    uint8_t bDscSubType;
    uint8_t bmCapabilities;
} USB_CDC_ACN_FN_DSC;
```

Members

Members	Description
uint8_t bFNLength;	Size of this functional descriptor, in uint8_ts.
uint8_t bDscType;	CS_INTERFACE
uint8_t bDscSubType;	Abstract Control Management functional descriptor subtype as defined in [USBCDC1.2].
uint8_t bmCapabilities;	The capabilities that this configuration supports. (A bit value of zero means that the request is not supported.)

Description

Abstract Control Management Functional Descriptor

1.4.2.2.2.4 USB_CDC_CALL_MGT_FN_DSC Structure

File

usb_host_cdc.h

Syntax

```
typedef struct _USB_CDC_CALL_MGT_FN_DSC {
    uint8_t bFNLength;
    uint8_t bDscType;
    uint8_t bDscSubType;
    uint8_t bmCapabilities;
    uint8_t bDataInterface;
} USB_CDC_CALL_MGT_FN_DSC;
```

Members

Members	Description
uint8_t bFNLength;	Size of this functional descriptor, in uint8_ts.
uint8_t bDscType;	CS_INTERFACE
uint8_t bDscSubType;	Call Management functional descriptor subtype, as defined in [USBCDC1.2].
uint8_t bmCapabilities;	The capabilities that this configuration supports:
uint8_t bDataInterface;	Interface number of Data Class interface optionally used for call management.

Description

Call Management Functional Descriptor

1.4.2.2.2.5 USB_CDC_CONTROL_SIGNAL_BITMAP Union

File

usb_host_cdc.h

Syntax

```
typedef union _USB_CDC_CONTROL_SIGNAL_BITMAP {
    uint8_t _uint8_t;
    struct {
        unsigned DTE_PRESENT : 1;
        unsigned CARRIER_CONTROL : 1;
    }
} USB_CDC_CONTROL_SIGNAL_BITMAP;
```

Members

Members	Description
unsigned DTE_PRESENT : 1;	0] Not Present [1] Present
unsigned CARRIER_CONTROL : 1;	0] Deactivate [1] Activate

Description

This is type USB_CDC_CONTROL_SIGNAL_BITMAP.

1.4.2.2.2.6 USB_CDC_DEVICE_INFO Structure**File**

usb_host_cdc.h

Syntax

```
typedef struct _USB_CDC_DEVICE_INFO {
    uint8_t* userData;
    uint16_t reportSize;
    uint16_t remainingBytes;
    uint16_t bytesTransferred;
    union {
        struct {
            uint8_t bfDirection : 1;
            uint8_t bfReset : 1;
            uint8_t bfClearDataIN : 1;
            uint8_t bfClearDataOUT : 1;
        }
        uint8_t val;
    } flags;
    uint8_t driverSupported;
    uint8_t deviceAddress;
    uint8_t errorCode;
    uint8_t state;
    uint8_t returnState;
    uint8_t noOfInterfaces;
    uint8_t interface;
    uint8_t endpointDATA;
    uint8_t commRequest;
    uint8_t clientDriverID;
    COMM_INTERFACE_DETAILS commInterface;
    DATA_INTERFACE_DETAILS dataInterface;
} USB_CDC_DEVICE_INFO;
```

Members

Members	Description
uint8_t* userData;	Data pointer to application buffer.
uint16_t reportSize;	Total length of user data
uint16_t remainingBytes;	Number uint8_ts remaining to be transferred in case user data length is more than 64 uint8_ts
uint16_t bytesTransferred;	Number of uint8_ts transferred to/from the user's data buffer.
uint8_t bfDirection : 1;	Direction of current transfer (0=OUT, 1=IN).
uint8_t bfReset : 1;	Flag indicating to perform CDC Reset.
uint8_t bfClearDataIN : 1;	Flag indicating to clear the IN endpoint.

uint8_t bfClearDataOUT : 1;	Flag indicating to clear the OUT endpoint.
uint8_t driverSupported;	If CDC driver supports requested Class,Subclass & Protocol.
uint8_t deviceAddress;	Address of the device on the bus.
uint8_t errorCode;	Error code of last error.
uint8_t state;	State machine state of the device.
uint8_t returnState;	State to return to after performing error handling.
uint8_t noOfInterfaces;	Total number of interfaces in the device.
uint8_t interface;	Interface number of current transfer.
uint8_t endpointDATA;	Endpoint to use for the current transfer.
uint8_t commRequest;	Current Communication code
uint8_t clientDriverID;	Client driver ID for device requests.
COMM_INTERFACE_DETAILS commInterface;	This structure stores communication interface details.
DATA_INTERFACE_DETAILS dataInterface;	This structure stores data interface details.

Description

This structure is used to hold information about an attached CDC device

1.4.2.2.7 USB_CDC_HEADER_FN_DSC Structure**File**

usb_host_cdc.h

Syntax

```
typedef struct _USB_CDC_HEADER_FN_DSC {
    uint8_t bFNLength;
    uint8_t bDscType;
    uint8_t bDscSubType;
    uint8_t bcdCDC[2];
} USB_CDC_HEADER_FN_DSC;
```

Members

Members	Description
uint8_t bFNLength;	Size of this functional descriptor, in uint8_ts.
uint8_t bDscType;	CS_INTERFACE
uint8_t bDscSubType;	Header. This is defined in [USBCDC1.2], which defines this as a header.
uint8_t bcdCDC[2];	USB Class Definitions for Communications Devices Specification release number in binary-coded decimal.

Description

Header Functional Descriptor

1.4.2.2.8 USB_CDC_LINE_CODING Union**File**

usb_host_cdc.h

Syntax

```
typedef union _USB_CDC_LINE_CODING {
    struct {
        uint8_t _uint8_t[USB_CDC_LINE_CODING_LENGTH];
    }
    struct {
        uint32_t dwDTERate;
        uint8_t bCharFormat;
        uint8_t bParityType;
        uint8_t bDataBits;
    }
}
```

```

    }
} USB_CDC_LINE_CODING;

```

Members

Members	Description
uint32_t dwDTERate;	Data terminal rate, in bits per second.
uint8_t bCharFormat;	Stop bits 0:1 Stop bit, 1:1.5 Stop bits, 2:2 Stop bits
uint8_t bParityType;	Parity 0:None, 1:Odd, 2:Even, 3:Mark, 4:Space
uint8_t bDataBits;	Data bits (5, 6, 7, 8 or 16)

Description

This is type USB_CDC_LINE_CODING.

1.4.2.2.2.9 USB_CDC_UNION_FN_DSC Structure**File**

usb_host_cdc.h

Syntax

```

typedef struct _USB_CDC_UNION_FN_DSC {
    uint8_t bFNLength;
    uint8_t bDscType;
    uint8_t bDscSubType;
    uint8_t bMasterIntf;
    uint8_t bSaveIntf0;
} USB_CDC_UNION_FN_DSC;

```

Members

Members	Description
uint8_t bFNLength;	Size of this functional descriptor, in uint8_ts.
uint8_t bDscType;	CS_INTERFACE
uint8_t bDscSubType;	Union Descriptor Functional Descriptor subtype as defined in [USBCDC1.2].
uint8_t bMasterIntf;	Interface number of the control (Communications Class) interface
uint8_t bSaveIntf0;	Interface number of the subordinate (Data Class) interface

Description

Union Functional Descriptor

1.4.2.2.2.10 DEVICE_CLASS_CDC Macro**File**

usb_host_cdc.h

Syntax

```
#define DEVICE_CLASS_CDC 0x02 // CDC Interface Class Code
```

Description

CDC Interface Class Code

1.4.2.2.2.11 EVENT_CDC_ATTACH Macro**File**

usb_host_cdc.h

Syntax

```
#define EVENT_CDC_ATTACH EVENT_CDC_BASE + EVENT_CDC_OFFSET + 1 // No event occurred (NULL event)
```

Description

No event occurred (NULL event)

1.4.2.2.12 EVENT_CDC_COMM_READ_DONE Macro

File

usb_host_cdc.h

Syntax

```
#define EVENT_CDC_COMM_READ_DONE EVENT_CDC_BASE + EVENT_CDC_OFFSET + 2 // A CDC Communication Read transfer has completed
```

Description

A CDC Communication Read transfer has completed

1.4.2.2.13 EVENT_CDC_COMM_WRITE_DONE Macro

File

usb_host_cdc.h

Syntax

```
#define EVENT_CDC_COMM_WRITE_DONE EVENT_CDC_BASE + EVENT_CDC_OFFSET + 3 // A CDC Communication Write transfer has completed
```

Description

A CDC Communication Write transfer has completed

1.4.2.2.14 EVENT_CDC_DATA_READ_DONE Macro

File

usb_host_cdc.h

Syntax

```
#define EVENT_CDC_DATA_READ_DONE EVENT_CDC_BASE + EVENT_CDC_OFFSET + 4 // A CDC Data Read transfer has completed
```

Description

A CDC Data Read transfer has completed

1.4.2.2.15 EVENT_CDC_DATA_WRITE_DONE Macro

File

usb_host_cdc.h

Syntax

```
#define EVENT_CDC_DATA_WRITE_DONE EVENT_CDC_BASE + EVENT_CDC_OFFSET + 5 // A CDC Data Write transfer has completed
```

Description

A CDC Data Write transfer has completed

1.4.2.2.2.16 EVENT_CDC_NAK_TIMEOUT Macro

File

usb_host_cdc.h

Syntax

```
#define EVENT_CDC_NAK_TIMEOUT EVENT_CDC_BASE + EVENT_CDC_OFFSET + 7 // CDC device NAK  
timeout has occurred
```

Description

CDC device NAK timeout has occurred

1.4.2.2.2.17 EVENT_CDC_NONE Macro

File

usb_host_cdc.h

Syntax

```
#define EVENT_CDC_NONE EVENT_CDC_BASE + EVENT_CDC_OFFSET + 0 // No event occurred (NULL  
event)
```

Description

No event occurred (NULL event)

1.4.2.2.2.18 EVENT_CDC_OFFSET Macro

File

usb_host_cdc.h

Syntax

```
#define EVENT_CDC_OFFSET 0
```

Description

If the application has not defined an offset for CDC events, set it to 0.

1.4.2.2.2.19 EVENT_CDC_RESET Macro

File

usb_host_cdc.h

Syntax

```
#define EVENT_CDC_RESET EVENT_CDC_BASE + EVENT_CDC_OFFSET + 6 // CDC reset complete
```

Description

CDC reset complete

1.4.2.2.2.20 USB_CDC_ABSTRACT_CONTROL_MODEL Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_ABSTRACT_CONTROL_MODEL 0x02 // Abstract Control Model
```

Description

Abstract Control Model

1.4.2.2.2.21 USB_CDC_ATM_NETWORKING_CONTROL_MODEL Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_ATM_NETWORKING_CONTROL_MODEL 0x07 // ATM Networking Control Model
```

Description

ATM Networking Control Model

1.4.2.2.2.22 USB_CDC_CAPI_CONTROL_MODEL Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_CAPI_CONTROL_MODEL 0x05 // CAPI Control Model
```

Description

CAPI Control Model

1.4.2.2.2.23 USB_CDC_CLASS_ERROR Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_CLASS_ERROR USB_ERROR_CLASS_DEFINED
```

Description

CDC Class Error Codes

1.4.2.2.2.24 USB_CDC_COMM_INTF Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_COMM_INTF 0x02 // Communication Interface Class Code
```

Description

Communication Interface Class Code

1.4.2.2.2.25 USB_CDC_COMMAND_FAILED Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_COMMAND_FAILED (USB_CDC_CLASS_ERROR | 0x01) // Command failed at the device.
```

Description

Command failed at the device.

1.4.2.2.26 USB_CDC_COMMAND_PASSED Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_COMMAND_PASSED USB_SUCCESS           // Command was successful.
```

Description

Command was successful.

1.4.2.2.27 USB_CDC_CONTROL_LINE_LENGTH Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_CONTROL_LINE_LENGTH 0x00           // Number of uint8_ts Control line transfer
```

Description

Number of uint8_ts Control line transfer

1.4.2.2.28 USB_CDC_CS_ENDPOINT Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_CS_ENDPOINT 0x25
```

Description

This is macro USB_CDC_CS_ENDPOINT.

1.4.2.2.29 USB_CDC_CS_INTERFACE Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_CS_INTERFACE 0x24
```

Description

Functional Descriptor Details Type Values for the bDscType Field

1.4.2.2.30 USB_CDC_DATA_INTF Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_DATA_INTF 0x0A
```

Description

Data Interface Class Codes

1.4.2.2.2.31 USB_CDC_DEVICE_BUSY Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_DEVICE_BUSY (USB_CDC_CLASS_ERROR | 0x04) // A transfer is currently in progress.
```

Description

A transfer is currently in progress.

1.4.2.2.2.32 USB_CDC_DEVICE_DETACHED Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_DEVICE_DETACHED 0x50 // Device is detached.
```

Description

Device is detached.

1.4.2.2.2.33 USB_CDC_DEVICE_HOLDING Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_DEVICE_HOLDING 0x54 // Device is holding due to error
```

Description

Device is holding due to error

1.4.2.2.2.34 USB_CDC_DEVICE_MANAGEMENT Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_DEVICE_MANAGEMENT 0x09 // Device Management
```

Description

Device Management

1.4.2.2.2.35 USB_CDC_DEVICE_NOT_FOUND Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_DEVICE_NOT_FOUND (USB_CDC_CLASS_ERROR | 0x03) // Device with the specified address is not available.
```

Description

Device with the specified address is not available.

1.4.2.2.2.36 USB_CDC_DIRECT_LINE_CONTROL_MODEL Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_DIRECT_LINE_CONTROL_MODEL 0x01 // Direct Line Control Model
```

Description

Direct Line Control Model

1.4.2.2.2.37 USB_CDC_DSC_FN_ACM Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_DSC_FN_ACM 0x02 // ACM - Abstract Control Management
```

Description

ACM - Abstract Control Management

1.4.2.2.2.38 USB_CDC_DSC_FN_CALL_MGT Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_DSC_FN_CALL_MGT 0x01
```

Description

This is macro USB_CDC_DSC_FN_CALL_MGT.

1.4.2.2.2.39 USB_CDC_DSC_FN_COUNTRY_SELECTION Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_DSC_FN_COUNTRY_SELECTION 0x07
```

Description

This is macro USB_CDC_DSC_FN_COUNTRY_SELECTION.

1.4.2.2.2.40 USB_CDC_DSC_FN_DLM Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_DSC_FN_DLM 0x03 // DLM - Direct Line Management
```

Description

DLM - Direct Line Management

1.4.2.2.2.41 USB_CDC_DSC_FN_HEADER Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_DSC_FN_HEADER 0x00
```

Description

bDscSubType in Functional Descriptors

1.4.2.2.2.42 USB_CDC_DSC_FN_RPT_CAPABILITIES Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_DSC_FN_RPT_CAPABILITIES 0x05
```

Description

This is macro USB_CDC_DSC_FN_RPT_CAPABILITIES.

1.4.2.2.2.43 USB_CDC_DSC_FN_TEL_OP_MODES Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_DSC_FN_TEL_OP_MODES 0x08
```

Description

This is macro USB_CDC_DSC_FN_TEL_OP_MODES.

1.4.2.2.2.44 USB_CDC_DSC_FN_TELEPHONE_RINGER Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_DSC_FN_TELEPHONE_RINGER 0x04
```

Description

This is macro USB_CDC_DSC_FN_TELEPHONE_RINGER.

1.4.2.2.2.45 USB_CDC_DSC_FN_UNION Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_DSC_FN_UNION 0x06
```

Description

This is macro USB_CDC_DSC_FN_UNION.

1.4.2.2.2.46 USB_CDC_DSC_FN_USB_TERMINAL Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_DSC_FN_USB_TERMINAL 0x09
```

Description

This is macro USB_CDC_DSC_FN_USB_TERMINAL.

1.4.2.2.2.47 USB_CDC_ETHERNET_EMULATION_MODEL Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_ETHERNET_EMULATION_MODEL 0x0C // Ethernet Emulation Model
```

Description

Ethernet Emulation Model

1.4.2.2.2.48 USB_CDC_ETHERNET_NETWORKING_CONTROL_MODEL Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_ETHERNET_NETWORKING_CONTROL_MODEL 0x06 // Ethernet Networking Control Model
```

Description

Ethernet Networking Control Model

1.4.2.2.2.49 USB_CDC_GET_COMM_FEATURE Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_GET_COMM_FEATURE 0x03 // Returns the current settings for the communications feature.
```

Description

Returns the current settings for the communications feature.

1.4.2.2.2.50 USB_CDC_GET_ENCAPSULATED_REQUEST Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_GET_ENCAPSULATED_REQUEST 0x01 // Requests a response in the format of the supported control protocol.
```

Description

Requests a response in the format of the supported control protocol.

1.4.2.2.2.51 USB_CDC_GET_LINE_CODING Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_GET_LINE_CODING 0x21      // Requests current DTE rate, stop-bits, parity,  
and number-of-character bits.
```

Description

Requests current DTE rate, stop-bits, parity, and number-of-character bits.

1.4.2.2.2.52 USB_CDC_ILLEGAL_REQUEST Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_ILLEGAL_REQUEST (USB_CDC_CLASS_ERROR | 0x0B) // Cannot perform requested  
operation.
```

Description

Cannot perform requested operation.

1.4.2.2.2.53 USB_CDC_INITIALIZING Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_INITIALIZING 0x51      // Device is initializing.
```

Description

Device is initializing.

1.4.2.2.2.54 USB_CDC_INTERFACE_ERROR Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_INTERFACE_ERROR (USB_CDC_CLASS_ERROR | 0x06) // The interface layer cannot  
support the device.
```

Description

The interface layer cannot support the device.

1.4.2.2.2.55 USB_CDC_LINE_CODING_LENGTH Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_LINE_CODING_LENGTH 0x07      // Number of uint8_ts Line Coding transfer
```

Description

Number of uint8_ts Line Coding transfer

1.4.2.2.2.56 USB_CDC_MAX_PACKET_SIZE Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_MAX_PACKET_SIZE 0x200 // Max transfer size is 64 uint8_ts for Full Speed  
USB
```

Description

Max transfer size is 64 uint8_ts for Full Speed USB

Data Structures

1.4.2.2.2.57 USB_CDC_MOBILE_DIRECT_LINE_MODEL Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_MOBILE_DIRECT_LINE_MODEL 0x0A // Mobile Direct Line Model
```

Description

Mobile Direct Line Model

1.4.2.2.2.58 USB_CDC_MULTI_CHANNEL_CONTROL_MODEL Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_MULTI_CHANNEL_CONTROL_MODEL 0x04 // Multi-Channel Control Model
```

Description

Multi-Channel Control Model

1.4.2.2.2.59 USB_CDC_NO_PROTOCOL Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_NO_PROTOCOL 0x00 // No class specific protocol required
```

Description

No class specific protocol required For more.... see Table 7 in USB CDC Specification 1.2

1.4.2.2.2.60 USB_CDC_NO_REPORT_DESCRIPTOR Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_NO_REPORT_DESCRIPTOR (USB_CDC_CLASS_ERROR | 0x05) // No report descriptor found
```

Description

No report descriptor found

1.4.2.2.61 USB_CDC_NORMAL_RUNNING Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_NORMAL_RUNNING 0x53 // Device is running and available for data transfers.
```

Description

Device is running and available for data transfers.

1.4.2.2.62 USB_CDC_OBEX Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_OBEX 0x0B // OBEX
```

Description

OBEX

1.4.2.2.63 USB_CDC_PHASE_ERROR Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_PHASE_ERROR (USB_CDC_CLASS_ERROR | 0x02) // Command had a phase error at the device.
```

Description

Command had a phase error at the device.

1.4.2.2.64 USB_CDC_REPORT_DESCRIPTOR_BAD Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_REPORT_DESCRIPTOR_BAD (USB_CDC_CLASS_ERROR | 0x05) // Report Descriptor for not proper
```

Description

Report Descriptor for not proper

1.4.2.2.65 USB_CDC_RESET_ERROR Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_RESET_ERROR (USB_CDC_CLASS_ERROR | 0x0A) // An error occurred while  
resetting the device.
```

Description

An error occurred while resetting the device.

1.4.2.2.66 USB_CDC_RESETTING_DEVICE Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_RESETTING_DEVICE 0x55 // Device is being reset.
```

Description

Device is being reset.

1.4.2.2.67 USB_CDC_SEND_BREAK Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_SEND_BREAK 0x23 // Sends special carrier modulation used to specify  
[V24] style break.
```

Description

Sends special carrier modulation used to specify [V24] style break.

1.4.2.2.68 USB_CDC_SEND_ENCAPSULATED_COMMAND Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_SEND_ENCAPSULATED_COMMAND 0x00 // Issues a command in the format of the  
supported control protocol.
```

Description

Issues a command in the format of the supported control protocol.

1.4.2.2.69 USB_CDC_SET_COMM_FEATURE Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_SET_COMM_FEATURE 0x02 // Controls the settings for a particular  
communications feature.
```

Description

Controls the settings for a particular communications feature.

1.4.2.2.2.70 USB_CDC_SET_CONTROL_LINE_STATE Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_SET_CONTROL_LINE_STATE 0x22      // [V24] signal used to tell the DCE device  
the DTE device is now present.
```

Description

V24] signal used to tell the DCE device the DTE device is now present.

1.4.2.2.2.71 USB_CDC_SET_LINE_CODING Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_SET_LINE_CODING 0x20      // Configures DTE rate, stop-bits, parity, and  
number-of-character bits.
```

Description

Configures DTE rate, stop-bits, parity, and number-of-character bits.

1.4.2.2.2.72 USB_CDC_TELEPHONE_CONTROL_MODEL Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_TELEPHONE_CONTROL_MODEL 0x03 // Telephone Control Model
```

Description

Telephone Control Model

1.4.2.2.2.73 USB_CDC_V25TER Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_V25TER 0x01      // Common AT commands ("Hayes(TM)")
```

Description

Common AT commands ("Hayes(TM)")

1.4.2.2.2.74 USB_CDC_WIRELESS_HANDSET_CONTROL_MODEL Macro

File

usb_host_cdc.h

Syntax

```
#define USB_CDC_WIRELESS_HANDSET_CONTROL_MODEL 0x08 // Wireless Handset Control Model
```

Description

Wireless Handset Control Model

1.4.2.2.3 usb_host_cdc.h

Functions

	Name	Description
USBHostCDCDeviceStatus	This function determines the status of a CDC device.	
USBHostCDCEventHandler	This function is the event handler for this client driver.	
USBHostCDCInitAddress	This function initializes the address of the attached CDC device.	
USBHostCDCInitialize	This function is the initialization routine for this client driver.	
USBHostCDCResetDevice	This function starts a CDC reset.	
USBHostCDCTasks	This function performs the maintenance tasks required by CDC class	
USBHostCDCTransfer	This function starts a CDC transfer.	
USBHostCDCTransferIsComplete	This function indicates whether or not the last transfer is complete.	

Macros

Name	Description
_USB_HOST_CDC_H_	This is macro _USB_HOST_CDC_H_.
DEVICE_CLASS_CDC	CDC Interface Class Code
EVENT_CDC_ATTACH	No event occurred (NULL event)
EVENT_CDC_COMM_READ_DONE	A CDC Communication Read transfer has completed
EVENT_CDC_COMM_WRITE_DONE	A CDC Communication Write transfer has completed
EVENT_CDC_DATA_READ_DONE	A CDC Data Read transfer has completed
EVENT_CDC_DATA_WRITE_DONE	A CDC Data Write transfer has completed
EVENT_CDC_NAK_TIMEOUT	CDC device NAK timeout has occurred
EVENT_CDC_NONE	No event occurred (NULL event)
EVENT_CDC_OFFSET	If the application has not defined an offset for CDC events, set it to 0.
EVENT_CDC_RESET	CDC reset complete
USB_CDC_ABSTRACT_CONTROL_MODEL	Abstract Control Model
USB_CDC_ATM_NETWORKING_CONTROL_MODEL	ATM Networking Control Model
USB_CDC_CAPI_CONTROL_MODEL	CAPI Control Model
USB_CDC_CLASS_ERROR	CDC Class Error Codes
USB_CDC_COMM_INTF	Communication Interface Class Code
USB_CDC_COMMAND_FAILED	Command failed at the device.
USB_CDC_COMMAND_PASSED	Command was successful.
USB_CDC_CONTROL_LINE_LENGTH	Number of uint8_ts Control line transfer
USB_CDC_CS_ENDPOINT	This is macro USB_CDC_CS_ENDPOINT.
USB_CDC_CS_INTERFACE	Functional Descriptor Details Type Values for the bDscType Field
USB_CDC_DATA_INTF	Data Interface Class Codes
USB_CDC_DEVICE_BUSY	A transfer is currently in progress.
USB_CDC_DEVICE_DETACHED	Device is detached.
USB_CDC_DEVICE_HOLDING	Device is holding due to error
USB_CDC_DEVICE_MANAGEMENT	Device Management
USB_CDC_DEVICE_NOT_FOUND	Device with the specified address is not available.
USB_CDC_DIRECT_LINE_CONTROL_MODEL	Direct Line Control Model
USB_CDC_DSC_FN_ACM	ACM - Abstract Control Management
USB_CDC_DSC_FN_CALL_MGT	This is macro USB_CDC_DSC_FN_CALL_MGT.
USB_CDC_DSC_FN_COUNTRY_SELECTION	This is macro USB_CDC_DSC_FN_COUNTRY_SELECTION.

USB_CDC_DSC_FN_DLM	DLM - Direct Line Management
USB_CDC_DSC_FN_HEADER	bDscSubType in Functional Descriptors
USB_CDC_DSC_FN_RPT_CAPABILITIES	This is macro USB_CDC_DSC_FN_RPT_CAPABILITIES.
USB_CDC_DSC_FN_TEL_OP_MODES	This is macro USB_CDC_DSC_FN_TEL_OP_MODES.
USB_CDC_DSC_FN_TELEPHONE_RINGER	This is macro USB_CDC_DSC_FN_TELEPHONE_RINGER.
USB_CDC_DSC_FN_UNION	This is macro USB_CDC_DSC_FN_UNION.
USB_CDC_FN_USB_TERMINAL	This is macro USB_CDC_FN_USB_TERMINAL.
USB_CDC_ETHERNET_EMULATION_MODEL	Ethernet Emulation Model
USB_CDC_ETHERNET_NETWORKING_CONTROL_MODEL	Ethernet Networking Control Model
USB_CDC_GET_COMM_FEATURE	Returns the current settings for the communications feature.
USB_CDC_GET_ENCAPSULATED_REQUEST	Requests a response in the format of the supported control protocol.
USB_CDC_GET_LINE_CODING	Requests current DTE rate, stop-bits, parity, and number-of-character bits.
USB_CDC_ILLEGAL_REQUEST	Cannot perform requested operation.
USB_CDC_INITIALIZING	Device is initializing.
USB_CDC_INTERFACE_ERROR	The interface layer cannot support the device.
USB_CDC_LINE_CODING_LENGTH	Number of uint8_ts Line Coding transfer
USB_CDC_MAX_PACKET_SIZE	Max transfer size is 64 uint8_ts for Full Speed USB
USB_CDC_MOBILE_DIRECT_LINE_MODEL	Mobile Direct Line Model
USB_CDC_MULTI_CHANNEL_CONTROL_MODEL	Multi-Channel Control Model
USB_CDC_NO_PROTOCOL	No class specific protocol required For more.... see Table 7 in USB CDC Specification 1.2
USB_CDC_NO_REPORT_DESCRIPTOR	No report descriptor found
USB_CDC_NORMAL_RUNNING	Device is running and available for data transfers.
USB_CDC_OBEX	OBEX
USB_CDC_PHASE_ERROR	Command had a phase error at the device.
USB_CDC_REPORT_DESCRIPTOR_BAD	Report Descriptor for not proper
USB_CDC_RESET_ERROR	An error occurred while resetting the device.
USB_CDC_RESETTING_DEVICE	Device is being reset.
USB_CDC_SEND_BREAK	Sends special carrier modulation used to specify [V24] style break.
USB_CDC_SEND_ENCAPSULATED_COMMAND	Issues a command in the format of the supported control protocol.
USB_CDC_SET_COMM_FEATURE	Controls the settings for a particular communications feature.
USB_CDC_SET_CONTROL_LINE_STATE	V24] signal used to tell the DCE device the DTE device is now present.
USB_CDC_SET_LINE_CODING	Configures DTE rate, stop-bits, parity, and number-of-character bits.
USB_CDC_TELEPHONE_CONTROL_MODEL	Telephone Control Model
USB_CDC_V25TER	Common AT commands ("Hayes(TM)")
USB_CDC_WIRELESS_HANDSET_CONTROL_MODEL	Wireless Handset Control Model

Module

CDC Client Driver

Structures

	Name	Description
	_COMM_INTERFACE_DETAILS	This structure stores communication interface details of the attached CDC device

	_DATA_INTERFACE_DETAILS	This structure stores data interface details of the attached CDC device
	_USB_CDC_ACM_FN_DSC	Abstract Control Management Functional Descriptor
	_USB_CDC_CALL_MGT_FN_DSC	Call Management Functional Descriptor
	_USB_CDC_DEVICE_INFO	This structure is used to hold information about an attached CDC device
	_USB_CDC_HEADER_FN_DSC	Header Functional Descriptor
	_USB_CDC_UNION_FN_DSC	Union Functional Descriptor
	COMM_INTERFACE_DETAILS	This structure stores communication interface details of the attached CDC device
	DATA_INTERFACE_DETAILS	This structure stores data interface details of the attached CDC device
	USB_CDC_ACM_FN_DSC	Abstract Control Management Functional Descriptor
	USB_CDC_CALL_MGT_FN_DSC	Call Management Functional Descriptor
	USB_CDC_DEVICE_INFO	This structure is used to hold information about an attached CDC device
	USB_CDC_HEADER_FN_DSC	Header Functional Descriptor
	USB_CDC_UNION_FN_DSC	Union Functional Descriptor

Unions

	Name	Description
	_USB_CDC_CONTROL_SIGNAL_BITMAP	This is type USB_CDC_CONTROL_SIGNAL_BITMAP.
	_USB_CDC_LINE_CODING	This is type USB_CDC_LINE_CODING.
	USB_CDC_CONTROL_SIGNAL_BITMAP	This is type USB_CDC_CONTROL_SIGNAL_BITMAP.
	USB_CDC_LINE_CODING	This is type USB_CDC_LINE_CODING.

Description

This is file `usb_host_cdc.h`.

1.4.2.2.4 usb_host_cdc_interface.h**Functions**

	Name	Description
	USBHostCDC_Api_ACM_Request	This function can be used by application code to dynamically access ACM specific requests. This function should be used only if application intends to modify for example the Baudrate from previously configured rate. Data transmitted/received to/from device is a array of bytes. Application must take extra care of understanding the data format before using this function.
	USBHostCDC_Api_Get_IN_Data	This function is called by application to receive Input data over DATA interface. This function sets up the request to receive data from the device.
	USBHostCDC_Api_Send_OUT_Data	This function is called by application to transmit out data over DATA interface. This function sets up the request to transmit data to the device.
	USBHostCDC_ApiDeviceDetect	This function determines if a CDC device is attached and ready to use.
	USBHostCDC_ApiTransferIsComplete	This function is called by application to poll for transfer status. This function returns true in the transfer is over. To check whether the transfer was successfull or not , application must check the error code returned by reference.

Macros

Name	Description
<code>_USB_HOST_CDC_INTERFACE_H_</code>	This is macro <code>_USB_HOST_CDC_INTERFACE_H_</code> .

Module

CDC Client Driver

Description

This is file `usb_host_cdc_interface.h`.

1.4.2.2.5 _USB_HOST_CDC_H_ Macro**File**

`usb_host_cdc.h`

Syntax

```
#define _USB_HOST_CDC_H_
```

Module

CDC Client Driver

Description

This is macro `_USB_HOST_CDC_H_`.

1.4.2.2.6 _USB_HOST_CDC_INTERFACE_H_ Macro**File**

`usb_host_cdc_interface.h`

Syntax

```
#define _USB_HOST_CDC_INTERFACE_H_
```

Module

CDC Client Driver

Description

This is macro `_USB_HOST_CDC_INTERFACE_H_`.

1.4.2.3 HID Client Driver

This client driver provides USB Embedded Host support for HID devices.

Enumerations

Name	Description
<code>USB_HOST_HID_RETURN_CODES</code>	This is type <code>USB_HOST_HID_RETURN_CODES</code> .

Files

Name	Description
<code>usb_host_hid.h</code>	This is file <code>usb_host_hid.h</code> .
<code>usb_host_hid_parser.h</code>	This is file <code>usb_host_hid_parser.h</code> .

Macros

Name	Description
<code>_USB_HOST_HID_PARSER_H_</code>	<code>usb_host_hid_parser.h</code>
<code>DSC_RPT_wValue</code>	Report Descriptor Code, used for <code>USBHostIssueDeviceRequest</code>
<code>USB_HID_TRANSFER_IN</code>	
<code>USB_HID_TRANSFER_OUT</code>	This is macro <code>USB_HID_TRANSFER_OUT</code> .

Description

This client driver provides USB Embedded Host support for HID devices. Common HID devices include mice, keyboards,

and bar code scanners. Many other USB peripherals also use the HID class to transfer data, since it provides a simple, flexible interface and does not require a custom Windows driver when used with a PC.

See [AN1144 - USB HID Class on an Embedded Host](#) and [AN1212 - Using USB Keyboard with an Embedded Host](#) for more information.

1.4.2.3.1 Functions

Functions

	Name	Description
≡	USBHostHID_ApiFindBit	This function is used to locate a specific button or indicator. Once the report descriptor is parsed by the HID layer without any error, data from the report descriptor is stored in pre defined dat structures. This function traverses these data structure and extract data required by application
≡	USBHostHID_ApiFindValue	Find a specific Usage Value. Once the report descriptor is parsed by the HID layer without any error, data from the report descriptor is stored in pre defined dat structures. This function traverses these data structure and extract data required by application.
≡	USBHostHID_ApiGetCurrentInterfaceNum	This function returns the interface number of the cuurent report descriptor parsed. This function must be called to fill data interface detail data structure and passed as parameter when requesinf for report transfers.
≡	USBHostHID_ApilImportData	This function can be used by application to extract data from the input reports. On receiving the input report from the device application can call the function with required inputs 'HID_DATA_DETAILS'.
≡	USBHostHID_HasUsage	This function is used to locate the usage in a report descriptor. Function will look into the data structures created by the HID parser and return the appropriate location.
≡	USBHostHIDDeviceDetect	This function determines if a HID device is attached and ready to use.
≡	USBHostHIDDeviceStatus	
≡	USBHostHIDEEventHandler	This function is the event handler for this client driver.
≡	USBHostHIDInitialize	This function is the initialization routine for this client driver.
≡	USBHostHIDRead	This function starts a Get report transfer reuest from the device, utilizing the function USBHostHIDTransfer();
≡	USBHostHIDReadIsComplete	This function indicates whether or not the last read request is complete.
≡	USBHostHIDReadTerminate	This function terminates a read request that is in progress.
≡	USBHostHIDResetDevice	This function starts a HID reset.
≡	USBHostHIDTasks	This function performs the maintenance tasks required by HID class
≡	USBHostHIDWrite	This function starts a Set report transfer request to the device, utilizing the function USBHostHIDTransfer();
≡	USBHostHIDWriteIsComplete	This function indicates whether or not the last write request is complete.
≡	USBHostHIDWriteTerminate	This function terminates a write request that is in progress.

Macros

Name	Description
USBHostHID_GetCurrentReportInfo	This function returns a pointer to the current report info structure.
USBHostHID_GetItemListPointers	This function returns a pointer to list of item pointers stored in a structure.

Module

HID Client Driver

Description**1.4.2.3.1.1 USBHostHID_ApiFindBit Function****File**

usb_host_hid.h

Syntax

```
bool USBHostHID_ApiFindBit(uint16_t usagePage, uint16_t usage, HIDReportTypeEnum type,  
uint8_t* Report_ID, uint8_t* Report_Length, uint8_t* Start_Bit);
```

Description

This function is used to locate a specific button or indicator. Once the report descriptor is parsed by the HID layer without any error, data from the report descriptor is stored in pre defined dat structures. This function traverses these data structure and extract data required by application

Remarks

Application event handler with event 'EVENT_HID_RPT_DESC_PARSED' is called. Application is suppose to fill in data details in structure 'HID_DATA_DETAILS'. This function can be used to the get the details of the required usages.

Preconditions

None

Return Values

Return Values	Description
true	If the required usage is located in the report descriptor
false	If the application required usage is not supported by the device(i.e report descriptor).

Function

```
bool USBHostHID_ApiFindBit(uint16_t usagePage, uint16_t usage, HIDReportTypeEnum type,  
uint8_t* Report_ID, uint8_t* Report_Length, uint8_t* Start_Bit)
```

1.4.2.3.1.2 USBHostHID_ApiFindValue Function**File**

usb_host_hid.h

Syntax

```
bool USBHostHID_ApiFindValue(uint16_t usagePage, uint16_t usage, HIDReportTypeEnum type,  
uint8_t* Report_ID, uint8_t* Report_Length, uint8_t* Start_Bit, uint8_t* Bit_Length);
```

Description

Find a specific Usage Value. Once the report descriptor is parsed by the HID layer without any error, data from the report descriptor is stored in pre defined dat structures. This function traverses these data structure and extract data required by application.

Remarks

Application event handler with event 'EVENT_HID_RPT_DESC_PARSED' is called. Application is suppose to fill in data details structure 'HID_DATA_DETAILS' This function can be used to the get the details of the required usages.

Preconditions

None

Return Values

Return Values	Description
true	If the required usage is located in the report descriptor
false	If the application required usage is not supported by the device(i.e report descriptor).

Function

```
bool USBHostHID_ApiFindValue(uint16_t usagePage,uint16_t usage,
    HIDReportTypeEnum type,uint8_t* Report_ID,uint8_t* Report_Length,uint8_t*
    Start_Bit, uint8_t* Bit_Length)
```

1.4.2.3.1.3 USBHostHID_ApiGetCurrentInterfaceNum Function**File**

usb_host_hid.h

Syntax

```
uint8_t USBHostHID_ApiGetCurrentInterfaceNum( );
```

Description

This function returns the interface number of the current report descriptor parsed. This function must be called to fill data interface detail data structure and passed as parameter when requesting for report transfers.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
true	Transfer is complete, errorCode is valid
false	Transfer is not complete, errorCode is not valid

Function

```
uint8_t USBHostHID_ApiGetCurrentInterfaceNum(void)
```

1.4.2.3.1.4 USBHostHID_ApiImportData Function**File**

usb_host_hid.h

Syntax

```
bool USBHostHID_ApiImportData(uint8_t * report, uint16_t reportLength, HID_USER_DATA_SIZE *
buffer, HID_DATA_DETAILS * pDataDetails);
```

Description

This function can be used by application to extract data from the input reports. On receiving the input report from the device application can call the function with required inputs 'HID_DATA_DETAILS'.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
true	If the required data is retrieved from the report
false	If required data is not found.

Function

```
bool USBHostHID_ApilImportData(uint8_t *report, uint16_t reportLength,
                                HID_USER_DATA_SIZE *buffer,HID_DATA_DETAILS *pDataDetails)
```

1.4.2.3.1.5 USBHostHID_HasUsage Function**File**

usb_host_hid_parser.h

Syntax

```
bool USBHostHID_HasUsage(HID_REPORTITEM * reportItem, uint16_t usagePage, uint16_t usage,
                         uint16_t * pindex, uint8_t* count);
```

Description

This function is used to locate the usage in a report descriptor. Function will look into the data structures created by the HID parser and return the appropriate location.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
bool	FALSE - If requested usage is not found
TRUE	if requested usage is found

Function

```
bool USBHostHID_HasUsage( HID_REPORTITEM *reportItem, uint16_t usagePage,
                           uint16_t usage, uint16_t *pindex, uint8_t* count)
```

1.4.2.3.1.6 USBHostHID_GetCurrentReportInfo Macro**File**

usb_host_hid.h

Syntax

```
#define USBHostHID_GetCurrentReportInfo (&deviceRptInfo)
```

Returns

uint8_t * - Pointer to the report Info structure.

Description

This function returns a pointer to the current report info structure.

Remarks

None

Preconditions

None

Function

```
uint8_t* USBHostHID_GetCurrentReportInfo(void)
```

1.4.2.3.1.7 USBHostHID_GetItemListPointers Macro

File

usb_host_hid.h

Syntax

```
#define USBHostHID_GetItemListPointers (&itemListPtrs)
```

Returns

uint8_t * - Pointer to list of item pointers structure.

Description

This function returns a pointer to list of item pointers stored in a structure.

Remarks

None

Preconditions

None

Function

```
uint8_t* USBHostHID_GetItemListPointers()
```

1.4.2.3.1.8 USBHostHIDDeviceDetect Function

File

usb_host_hid.h

Syntax

```
uint8_t USBHostHIDDeviceDetect();
```

Description

This function determines if a HID device is attached and ready to use.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
uint8_t deviceAddress	Address of the newly attached device or 0 if there are no newly attached devices.

Function

```
bool USBHostHIDDeviceDetect( uint8_t deviceAddress )
```

1.4.2.3.1.9 USBHostHIDDeviceStatus Function

File

`usb_host_hid.h`

Syntax

```
uint8_t USBHostHIDDeviceStatus(uint8_t deviceAddress);
```

Description

This function determines the status of a HID device.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
<code>USB_HID_DEVICE_NOT_FOUND</code>	Illegal device address, or the device is not an HID
<code>USB_HID_INITIALIZING</code>	HID is attached and in the process of initializing
<code>USB_PROCESSING_REPORT_DESCRIPTOR</code>	HID device is detected and report descriptor is being parsed
<code>USB_HID_NORMAL_RUNNING</code>	HID Device is running normal, ready to send and receive reports
<code>USB_HID_DEVICE_HOLDING</code>	Driver has encountered error and could not recover
<code>USB_HID_DEVICE_DETACHED</code>	HID detached.

Function

```
uint8_t USBHostHIDDeviceStatus( uint8_t deviceAddress )
```

1.4.2.3.1.10 USBHostHIDEEventHandler Function

This function is the event handler for this client driver.

File

`usb_host_hid.h`

Syntax

```
bool USBHostHIDEEventHandler(uint8_t address, USB_EVENT event, void * data, uint32_t size);
```

Description

This function is the event handler for this client driver. It is called by the host layer when various events occur.

Remarks

None

Preconditions

The device has been initialized.

Return Values

Return Values	Description
<code>true</code>	Event was handled
<code>false</code>	Event was not handled

Function

```
bool USBHostHIDEEventHandler( uint8_t address, USB_EVENT event,
```

```
void *data, uint32_t size )
```

1.4.2.3.1.11 USBHostHIDInitialize Function

This function is the initialization routine for this client driver.

File

usb_host_hid.h

Syntax

```
bool USBHostHIDInitialize(uint8_t address, uint32_t flags, uint8_t clientDriverID);
```

Description

This function is the initialization routine for this client driver. It is called by the host layer when the USB device is being enumerated. For a HID device we need to look into HID descriptor, interface descriptor and endpoint descriptor.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
true	We can support the device.
false	We cannot support the device.

Function

```
bool USBHostHIDInitialize( uint8_t address, uint16_t flags, uint8_t clientDriverID )
```

1.4.2.3.1.12 USBHostHIDRead Function

This function starts a Get report transfer request from the device, utilizing the function USBHostHIDTransfer();

File

usb_host_hid.h

Syntax

```
uint8_t USBHostHIDRead(uint8_t deviceAddress, uint8_t reportid, uint8_t interface, uint8_t size, uint8_t * data);
```

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_SUCCESS	Request started successfully
USB_HID_DEVICE_NOT_FOUND	No device with specified address
USB_HID_DEVICE_BUSY	Device not in proper state for performing a transfer
Others	Return values from USBHostRead()

Function

```
uint8_t USBHostHIDRead( uint8_t deviceAddress, uint8_t reportid, uint8_t interface,
                        uint8_t size, uint8_t *data)
```

1.4.2.3.1.13 USBHostHIDReadIsComplete Function

This function indicates whether or not the last read request is complete.

File

usb_host_hid.h

Syntax

```
bool USBHostHIDReadIsComplete(uint8_t deviceAddress, uint8_t * errorCode, uint8_t * byteCount);
```

Description

This function indicates whether or not the last read request is complete. If the functions returns true, the returned byte count and error code are valid. Since only one read request can be performed at once and only one endpoint can be used, we only need to know the device address.

Preconditions

None

Return Values

Return Values	Description
true	Transfer is complete, errorCode is valid
false	Transfer is not complete, errorCode is not valid

Function

```
bool USBHostHIDReadIsComplete( uint8_t deviceAddress, uint8_t *errorCode, uint32_t *byteCount )
```

1.4.2.3.1.14 USBHostHIDReadTerminate Function

This function terminates a read request that is in progress.

File

usb_host_hid.h

Syntax

```
uint8_t USBHostHIDReadTerminate(uint8_t deviceAddress, uint8_t interfaceNum);
```

Description

This function terminates a read request that is in progress.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_SUCCESS	read request terminated
USB_HID_DEVICE_NOT_FOUND	No device with specified address

Function

```
uint8_t USBHostHIDReadTerminate( uint8_t deviceAddress, uint8_t interfaceNum )
```

1.4.2.3.1.15 USBHostHIDResetDevice Function

This function starts a HID reset.

File

usb_host_hid.h

Syntax

```
uint8_t USBHostHIDResetDevice(uint8_t deviceAddress);
```

Description

This function starts a HID reset. A reset can be issued only if the device is attached and not being initialized.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_SUCCESS	Reset started
USB_MSD_DEVICE_NOT_FOUND	No device with specified address
USB_MSD_ILLEGAL_REQUEST	Device is in an illegal state for reset

Function

```
uint8_t USBHostHIDResetDevice( uint8_t deviceAddress )
```

1.4.2.3.1.16 USBHostHIDTasks Function

This function performs the maintenance tasks required by HID class

File

usb_host_hid.h

Syntax

```
void USBHostHIDTasks();
```

Returns

None

Description

This function performs the maintenance tasks required by the HID class. If transfer events from the host layer are not being used, then it should be called on a regular basis by the application. If transfer events from the host layer are being used, this function is compiled out, and does not need to be called.

Remarks

None

Preconditions

USBHostHIDInitialize() has been called.

Parameters

Parameters	Description
None	None

Function

```
void USBHostHIDTasks( void )
```

1.4.2.3.1.17 USBHostHIDWrite Function

This function starts a Set report transfer request to the device, utilizing the function USBHostHIDTransfer();

File

usb_host_hid.h

Syntax

```
uint8_t USBHostHIDWrite(uint8_t deviceAddress, uint8_t reportid, uint8_t interface, uint8_t size, uint8_t * data);
```

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_SUCCESS	Request started successfully
USB_HID_DEVICE_NOT_FOUND	No device with specified address
USB_HID_DEVICE_BUSY	Device not in proper state for performing a transfer
Others	Return values from USBHostIssueDeviceRequest(), and USBHostWrite()

Function

```
uint8_t USBHostHIDWrite( uint8_t deviceAddress, uint8_t reportid, uint8_t interface, uint8_t size, uint8_t * data)
```

1.4.2.3.1.18 USBHostHIDWriteIsComplete Function

This function indicates whether or not the last write request is complete.

File

usb_host_hid.h

Syntax

```
bool USBHostHIDWriteIsComplete(uint8_t deviceAddress, uint8_t * errorCode, uint8_t * byteCount);
```

Description

This function indicates whether or not the last write request is complete. If the functions returns true, the returned byte count and error code are valid. Since only one write can be performed at once and only one endpoint can be used, we only need to know the device address.

Preconditions

None

Return Values

Return Values	Description
true	Transfer is complete, errorCode is valid
false	Transfer is not complete, errorCode is not valid

Function

```
bool USBHostHIDWriteIsComplete( uint8_t deviceAddress, uint8_t *errorCode, uint32_t *byteCount )
```

1.4.2.3.1.19 USBHostHIDWriteTerminate Function

This function terminates a write request that is in progress.

File

usb_host_hid.h

Syntax

```
uint8_t USBHostHIDWriteTerminate(uint8_t deviceAddress, uint8_t interfaceNum);
```

Description

This function terminates a write request that is in progress.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_SUCCESS	write request terminated
USB_HID_DEVICE_NOT_FOUND	No device with specified address

Function

```
uint8_t USBHostHIDWriteTerminate( uint8_t deviceAddress, uint8_t interfaceNum )
```

1.4.2.3.2 Data Types and Constants

Enumerations

Name	Description
HIDReportTypeEnum	This is type HIDReportTypeEnum.
USB_HID_RPT_DESC_ERROR	HID parser error codes This enumerates the error encountered during the parsing of report descriptor. In case of any error parsing is stopped and the error is flagged. Device is not attached successfully.

Macros

Name	Description
DEVICE_CLASS_HID	HID Interface Class Code
DSC_HID	HID Descriptor Code
DSC_PHY	Physical Descriptor Code
EVENT_HID_ATTACH	A HID device has attached. The returned data pointer points to a USB_HID_DEVICE_ID structure.
EVENT_HID_BAD_REPORT_DESCRIPTOR	There was a problem parsing the report descriptor of the attached device. Communication with the device is not allowed, and the device should be detached.
EVENT_HID_DETACH	A HID device has detached. The returned data pointer points to a byte with the previous address of the detached device.
EVENT_HID_NONE	No event occurred (NULL event)
EVENT_HID_OFFSET	If the application has not defined an offset for HID events, set it to 0.
EVENT_HID_READ_DONE	define EVENT_HID_TRANSFER EVENT_HID_BASE + EVENT_HID_OFFSET + 3 // Unused - value retained for legacy. A HID Read transfer has completed. The returned data pointer points to a HID_TRANSFER_DATA structure, with information about the transfer.

EVENT_HID_RESET	HID reset complete. The returned data pointer is NULL.
EVENT_HID_RESET_ERROR	An error occurred while trying to do a HID reset. The returned data pointer is NULL.
EVENT_HID_RPT_DESC_PARSED	A Report Descriptor has been parsed. The returned data pointer is NULL. The application must collect details, or simply return true if the application is already aware of the data format.
EVENT_HID_WRITE_DONE	A HID Write transfer has completed. The returned data pointer points to a HID_TRANSFER_DATA structure, with information about the transfer.
USB_HID_CLASS_ERROR	

Module

HID Client Driver

Structures

	Name	Description
◆	_HID_COLLECTION	HID Collection Details This structure contains information about each collection encountered in the report descriptor.
◆	_HID_DATA_DETAILS	HID Data Details This structure defines the objects used by the application to access required report. Application must use parser interface functions to fill these details. e.g. USBHostHID_ApiFindValue
◆	_HID_GLOBALS	HID Global Item Information This structure contains information about each Global Item of the report descriptor.
◆	_HID_ITEM_INFO	HID Item Information This structure contains information about each Item of the report descriptor.
◆	_HID_REPORT	HID Report details This structure contains information about each report exchanged with the device.
◆	_HID_REPORTITEM	HID Report Details This structure contains information about each Report encountered in the report descriptor.
◆	_HID_STRINGITEM	HID String Item Details This structure contains information about each Report encountered in the report descriptor.
◆	_HID_TRANSFER_DATA	HID Transfer Information This structure is used when the event handler is used to notify the upper layer of transfer completion (EVENT_HID_READ_DONE or EVENT_HID_WRITE_DONE).
◆	_HID_USAGEITEM	HID Report Details This structure contains information about each Usage Item encountered in the report descriptor.
	HID_COLLECTION	HID Collection Details This structure contains information about each collection encountered in the report descriptor.
	HID_DATA_DETAILS	HID Data Details This structure defines the objects used by the application to access required report. Application must use parser interface functions to fill these details. e.g. USBHostHID_ApiFindValue
	HID_DESIGITEM	HID String Item Details This structure contains information about each Report encountered in the report descriptor.

	HID_GLOBALS	HID Global Item Information This structure contains information about each Global Item of the report descriptor.
	HID_ITEM_INFO	HID Item Information This structure contains information about each Item of the report descriptor.
	HID_REPORT	HID Report details This structure contains information about each report exchanged with the device.
	HID_REPORTITEM	HID Report Details This structure contains information about each Report encountered in the report descriptor.
	HID_STRINGITEM	HID String Item Details This structure contains information about each Report encountered in the report descriptor.
	HID_TRANSFER_DATA	HID Transfer Information This structure is used when the event handler is used to notify the upper layer of transfer completion (EVENT_HID_READ_DONE or EVENT_HID_WRITE_DONE).
	HID_USAGEITEM	HID Report Details This structure contains information about each Usage Item encountered in the report descriptor.
❖	_USB_HID_DEVICE_ID	HID Device ID Information This structure contains identification information about an attached device.
❖	_USB_HID_DEVICE_RPT_INFO	Report Descriptor Information This structure contains top level information of the report descriptor. This information is important and is used to understand the information during the course of parsing. This structure also stores temporary data needed during parsing the report descriptor. All of this information may not be of much importance to the application.
	USB_HID_DEVICE_ID	HID Device ID Information This structure contains identification information about an attached device.
❖	_USB_HID_ITEM_LIST	List of Items This structure contains array of pointers to all the Items in the report descriptor. HID parser will populate the lists while parsing the report descriptor. This data is used by interface functions provided in file usb_host_hid_interface.c to retrieve data from the report received from the device. Application can also access these details to retrieve the intended information incase provided interface function fail to do so.
	USB_HID_DEVICE_RPT_INFO	Report Descriptor Information This structure contains top level information of the report descriptor. This information is important and is used to understand the information during the course of parsing. This structure also stores temporary data needed during parsing the report descriptor. All of this information may not be of much importance to the application.
	USB_HID_ITEM_LIST	List of Items This structure contains array of pointers to all the Items in the report descriptor. HID parser will populate the lists while parsing the report descriptor. This data is used by interface functions provided in file usb_host_hid_interface.c to retrieve data from the report received from the device. Application can also access these details to retrieve the intended information incase provided interface function fail to do so.

Types

Name	Description
HID_USER_DATA_SIZE	HID User Data Size This defines the data type required to hold the maximum field size data. Maximum size of data field within a report

Variables

Name	Description
deviceRptInfo	
itemListPtrs	This is variable itemListPtrs.

Description**1.4.2.3.2.1 HID_COLLECTION Structure****File**

usb_host_hid_parser.h

Syntax

```
typedef struct _HID_COLLECTION {
    uint32_t data;
    uint16_t usagePage;
    uint8_t firstUsageItem;
    uint8_t usageItems;
    uint8_t firstReportItem;
    uint8_t reportItems;
    uint8_t parent;
    uint8_t firstChild;
    uint8_t nextSibling;
} HID_COLLECTION;
```

Members

Members	Description
uint32_t data;	Collection raw data
uint16_t usagePage;	Usage page associated with current level of collection
uint8_t firstUsageItem;	Index of First Usage Item in the current collection
uint8_t usageItems;	Number of Usage Items in the current collection
uint8_t firstReportItem;	Index of First report Item in the current collection
uint8_t reportItems;	Number of report Items in the current collection
uint8_t parent;	Index to Parent collection
uint8_t firstChild;	Index to next child collection in the report descriptor
uint8_t nextSibling;	Index to next child collection in the report descriptor

Description

HID Collection Details

This structure contains information about each collection encountered in the report descriptor.

1.4.2.3.2.2 HID_DATA_DETAILS Structure**File**

usb_host_hid.h

Syntax

```
typedef struct _HID_DATA_DETAILS {
    uint16_t reportLength;
```

```

    uint16_t reportID;
    uint8_t bitOffset;
    uint8_t bitLength;
    uint8_t count;
    uint8_t signExtend;
    uint8_t interfaceNum;
} HID_DATA_DETAILS;

```

Members

Members	Description
uint16_t reportLength;	reportLength - the expected length of the parent report.
uint16_t reportID;	reportID - report ID - the first byte of the parent report.
uint8_t bitOffset;	BitOffset - bit offset within the report.
uint8_t bitLength;	bitlength - length of the data in bits.
uint8_t count;	count - what's left of the message after this data.
uint8_t signExtend;	extend - sign extend the data.
uint8_t interfaceNum;	interfaceNum - informs HID layer about interface number.

Description**HID Data Details**

This structure defines the objects used by the application to access required report. Application must use parser interface functions to fill these details. e.g. USBHostHID_ApiFindValue

1.4.2.3.2.3 HID_DESIGITEM Structure**File**

usb_host_hid_parser.h

Syntax

```

typedef struct _HID_STRINGITEM {
    bool isRange;
    uint16_t index;
    uint16_t minimum;
    uint16_t maximum;
} HID_STRINGITEM, HID_DESIGITEM;

```

Members

Members	Description
bool isRange;	If range of String Item is valid
uint16_t index;	String index for a String descriptor; allows a string to be associated with a particular item or control
uint16_t minimum;	Specifies the first string index when assigning a group of sequential strings to controls in an array or bitmap
uint16_t maximum;	Specifies the last string index when assigning a group of sequential strings to controls in an array or bitmap

Description**HID String Item Details**

This structure contains information about each Report encountered in the report descriptor.

1.4.2.3.2.4 HID_GLOBALS Structure**File**

usb_host_hid_parser.h

Syntax

```

typedef struct _HID_GLOBALS {

```

```

    uint16_t usagePage;
    int32_t logicalMinimum;
    int32_t logicalMaximum;
    int32_t physicalMinimum;
    int32_t physicalMaximum;
    int32_t unitExponent;
    int32_t unit;
    uint16_t reportIndex;
    uint8_t reportID;
    uint8_t reportSize;
    uint8_t reportCount;
} HID_GLOBALS;

```

Members

Members	Description
uint16_t usagePage;	Specifies current Usage Page
int32_t logicalMinimum;	This is the minimum value that a variable or array item will report
int32_t logicalMaximum;	This is the maximum value that a variable or array item will report
int32_t physicalMinimum;	Minimum value for the physical extent of a variable item
int32_t physicalMaximum;	Maximum value for the physical extent of a variable item
int32_t unitExponent;	Value of the unit exponent in base 10
int32_t unit;	Unit values
uint16_t reportIndex;	Counter to keep track of report being processed in the parser
uint8_t reportID;	Report ID. All the reports are preceded by a single byte report ID
uint8_t reportSize;	Size of current report in bytes
uint8_t reportCount;	This field determines number of fields in the report

Description

HID Global Item Information

This structure contains information about each Global Item of the report descriptor.

1.4.2.3.2.5 HID_ITEM_INFO Structure

File

usb_host_hid_parser.h

Syntax

```

typedef struct _HID_ITEM_INFO {
    union {
        struct {
            uint8_t ItemSize : 2;
            uint8_t ItemType : 2;
            uint8_t ItemTag : 4;
        }
        uint8_t val;
    } ItemDetails;
    union {
        int32_t sItemData;
        uint32_t uItemData;
        uint8_t bItemData[4];
    } Data;
} HID_ITEM_INFO;

```

Members

Members	Description
uint8_t ItemSize : 2;	Numeric expression specifying size of data
uint8_t ItemType : 2;	This field identifies type of item(Main, Global or Local)

<code>uint8_t ItemTag : 4;</code>	This field specifies the function of the item
<code>uint8_t val;</code>	to access the data in byte format
<code>int32_t sItemData;</code>	Item Data is stored in signed format
<code>uint32_t uItemData;</code>	Item Data is stored in unsigned format

Description

HID Item Information

This structure contains information about each Item of the report descriptor.

1.4.2.3.2.6 HID_REPORT Structure**File**

`usb_host_hid_parser.h`

Syntax

```
typedef struct _HID_REPORT {
    uint16_t reportID;
    uint16_t inputBits;
    uint16_t outputBits;
    uint16_t featureBits;
} HID_REPORT;
```

Members

Members	Description
<code>uint16_t reportID;</code>	Report ID of the associated report
<code>uint16_t inputBits;</code>	If input report then length of report in bits
<code>uint16_t outputBits;</code>	If output report then length of report in bits
<code>uint16_t featureBits;</code>	If feature report then length of report in bits

Description

HID Report details

This structure contains information about each report exchanged with the device.

1.4.2.3.2.7 HID_REPORTITEM Structure**File**

`usb_host_hid_parser.h`

Syntax

```
typedef struct _HID_REPORTITEM {
    HIDReportTypeEnum reportType;
    HID_GLOBALS globals;
    uint8_t startBit;
    uint8_t parent;
    uint32_t dataModes;
    uint8_t firstUsageItem;
    uint8_t usageItems;
    uint8_t firstStringItem;
    uint8_t stringItems;
    uint8_t firstDesignatorItem;
    uint8_t designatorItems;
} HID_REPORTITEM;
```

Members

Members	Description
<code>HIDReportTypeEnum reportType;</code>	Type of Report Input/Output/Feature
<code>HID_GLOBALS globals;</code>	Stores all the global items associated with the current report

uint8_t startBit;	Starting Bit Position of the report
uint8_t parent;	Index of parent collection
uint32_t dataModes;	this tells the data mode is array or not
uint8_t firstUsageItem;	Index to first usage item related to the report
uint8_t usageItems;	Number of usage items in the current report
uint8_t firstStringItem;	Index to first string item in the list
uint8_t stringItems;	Number of string items in the current report
uint8_t firstDesignatorItem;	Index to first designator item
uint8_t designatorItems;	Number of designator items in the current report

Description

HID Report Details

This structure contains information about each Report encountered in the report descriptor.

1.4.2.3.2.8 HID_STRINGITEM Structure**File**

usb_host_hid_parser.h

Syntax

```
typedef struct _HID_STRINGITEM {
    bool isRange;
    uint16_t index;
    uint16_t minimum;
    uint16_t maximum;
} HID_STRINGITEM, HID_DESIGITEM;
```

Members

Members	Description
bool isRange;	If range of String Item is valid
uint16_t index;	String index for a String descriptor; allows a string to be associated with a particular item or control
uint16_t minimum;	Specifies the first string index when assigning a group of sequential strings to controls in an array or bitmap
uint16_t maximum;	Specifies the last string index when assigning a group of sequential strings to controls in an array or bitmap

Description

HID String Item Details

This structure contains information about each Report encountered in the report descriptor.

1.4.2.3.2.9 HID_TRANSFER_DATA Structure**File**

usb_host_hid.h

Syntax

```
typedef struct _HID_TRANSFER_DATA {
    uint32_t dataCount;
    uint8_t bErrorCode;
} HID_TRANSFER_DATA;
```

Members

Members	Description
uint32_t dataCount;	Count of bytes transferred.

<code>uint8_t bErrorCode;</code>	Transfer error code.
----------------------------------	----------------------

Description

HID Transfer Information

This structure is used when the event handler is used to notify the upper layer of transfer completion (EVENT_HID_READ_DONE or EVENT_HID_WRITE_DONE).

1.4.2.3.2.10 HID_USAGEITEM Structure**File**

`usb_host_hid_parser.h`

Syntax

```
typedef struct _HID_USAGEITEM {
    bool isRange;
    uint16_t usagePage;
    uint16_t usage;
    uint16_t usageMinimum;
    uint16_t usageMaximum;
} HID_USAGEITEM;
```

Members

Members	Description
<code>bool isRange;</code>	True if Usage item has a valid MAX and MIN range
<code>uint16_t usagePage;</code>	Usage page ID associated with the Item
<code>uint16_t usage;</code>	Usage ID associated with the Item
<code>uint16_t usageMinimum;</code>	Defines the starting usage associated with an array or bitmap
<code>uint16_t usageMaximum;</code>	Defines the ending usage associated with an array or bitmap

Description

HID Report Details

This structure contains information about each Usage Item encountered in the report descriptor.

1.4.2.3.2.11 HID_USER_DATA_SIZE Type**File**

`usb_host_hid.h`

Syntax

```
typedef unsigned char HID_USER_DATA_SIZE;
```

Description

HID User Data Size

This defines the data type required to hold the maximum field size data.

Maximum size of data field within a report

1.4.2.3.2.12 HIDReportTypeEnum Enumeration**File**

`usb_host_hid_parser.h`

Syntax

```
typedef enum {
    hidReportInput,
    hidReportOutput,
```

```

    hidReportFeature,
    hidReportUnknown
} HIDReportTypeEnum;
}
```

Description

This is type HIDReportTypeEnum.

1.4.2.3.2.13 USB_HID_DEVICE_ID Structure**File**

usb_host_hid.h

Syntax

```

typedef struct _USB_HID_DEVICE_ID {
    uint16_t vid;
    uint16_t pid;
    uint8_t deviceAddress;
    uint8_t clientDriverID;
} USB_HID_DEVICE_ID;
```

Members

Members	Description
uint16_t vid;	Vendor ID of the device
uint16_t pid;	Product ID of the device
uint8_t deviceAddress;	Address of the device on the USB
uint8_t clientDriverID;	Client driver ID for device requests

Description

HID Device ID Information

This structure contains identification information about an attached device.

1.4.2.3.2.14 USB_HID_DEVICE_RPT_INFO Structure**File**

usb_host_hid_parser.h

Syntax

```

typedef struct _USB_HID_DEVICE_RPT_INFO {
    uint16_t reportPollingRate;
    uint8_t interfaceNumber;
    bool haveDesignatorMax;
    bool haveDesignatorMin;
    bool haveStringMax;
    bool haveStringMin;
    bool haveUsageMax;
    bool haveUsageMin;
    uint16_t designatorMaximum;
    uint16_t designatorMinimum;
    uint16_t designatorRanges;
    uint16_t designators;
    uint16_t rangeUsagePage;
    uint16_t stringMaximum;
    uint16_t stringMinimum;
    uint16_t stringRanges;
    uint16_t usageMaximum;
    uint16_t usageMinimum;
    uint16_t usageRanges;
    uint8_t collectionNesting;
    uint8_t collections;
    uint8_t designatorItems;
    uint8_t firstUsageItem;
    uint8_t firstDesignatorItem;
}
```

```

    uint8_t firstStringItem;
    uint8_t globalsNesting;
    uint8_t maxCollectionNesting;
    uint8_t maxGlobalsNesting;
    uint8_t parent;
    uint8_t reportItems;
    uint8_t reports;
    uint8_t sibling;
    uint8_t stringItems;
    uint8_t strings;
    uint8_t usageItems;
    uint8_t usages;
    HID_GLOBALS globals;
} USB_HID_DEVICE_RPT_INFO;

```

Members

Members	Description
uint16_t reportPollingRate;	This stores the pollrate for the input report. Application can use this to decide the rate of transfer
uint8_t interfaceNumber;	This stores the interface number for the current report descriptor
bool haveDesignatorMax;	True if report descriptor has a valid Designator Max
bool haveDesignatorMin;	True if report descriptor has a valid Designator Min
bool haveStringMax;	True if report descriptor has a valid String Max
bool haveStringMin;	True if report descriptor has a valid String Min
bool haveUsageMax;	True if report descriptor has a valid Usage Max
bool haveUsageMin;	True if report descriptor has a valid Usage Min
uint16_t designatorMaximum;	Last designator max value
uint16_t designatorMinimum;	Last designator min value
uint16_t designatorRanges;	Last designator range
uint16_t designators;	This tells total number of designator items
uint16_t rangeUsagePage;	current usage page during parsing
uint16_t stringMaximum;	current string maximum
uint16_t stringMinimum;	current string minimum
uint16_t stringRanges;	current string ranges
uint16_t usageMaximum;	current usage maximum
uint16_t usageMinimum;	current usage minimum
uint16_t usageRanges;	current usage ranges
uint8_t collectionNesting;	this number tells depth of collection nesting
uint8_t collections;	total number of collections
uint8_t designatorItems;	total number of designator items
uint8_t firstUsageItem;	index of first usage item for the current collection
uint8_t firstDesignatorItem;	index of first designator item for the current collection
uint8_t firstStringItem;	index of first string item for the current collection
uint8_t globalsNesting;	On encountering every PUSH item , this is incremented , keep track of current depth of Globals
uint8_t maxCollectionNesting;	Maximum depth of collections
uint8_t maxGlobalsNesting;	Maximum depth of Globals
uint8_t parent;	Parent collection
uint8_t reportItems;	total number of report items
uint8_t reports;	total number of reports
uint8_t sibling;	current sibling collection
uint8_t stringItems;	total number of string items , used to index the array of strings
uint8_t strings;	total sumber of strings

uint8_t usageItems;	total number of usage items , used to index the array of usage
uint8_t usages;	total sumber of usages
HID_GLOBALS globals;	holds cuurent globals items

Description

Report Descriptor Information

This structure contains top level information of the report descriptor. This information is important and is used to understand the information during th ecourse of parsing. This structure also stores temporary data needed during parsing the report descriptor. All of this information may not be of much importance to the application.

1.4.2.3.2.15 USB_HID_ITEM_LIST Structure**File**

usb_host_hid_parser.h

Syntax

```
typedef struct _USB_HID_ITEM_LIST {
    HID_COLLECTION * collectionList;
    HID_DESIGITEM * designatorItemList;
    HID_GLOBALS * globalsStack;
    HID_REPORTITEM * reportItemList;
    HID_REPORT * reportList;
    HID_STRINGITEM * stringItemList;
    HID_USAGEITEM * usageItemList;
    uint8_t * collectionStack;
} USB_HID_ITEM_LIST;
```

Members

Members	Description
HID_COLLECTION * collectionList;	List of collections, see HID_COLLECTION for details in the structure
HID_DESIGITEM * designatorItemList;	List of designator Items, see HID_DESIGITEM for details in the structure
HID_GLOBALS * globalsStack;	List of global Items, see HID_GLOBALS for details in the structure
HID_REPORTITEM * reportItemList;	List of report Items, see HID_REPORTITEM for details in the structure
HID_REPORT * reportList;	List of reports , see HID_REPORT for details in the structure
HID_STRINGITEM * stringItemList;	List of string item , see HID_STRINGITEM for details in the structure
HID_USAGEITEM * usageItemList;	List of Usage item , see HID_USAGEITEM for details in the structure
uint8_t * collectionStack;	stores the array of parents ids for the collection

Description

List of Items

This structure contains array of pointers to all the Items in the report descriptor. HID parser will populate the lists while parsing the report descriptor. This data is used by interface functions provided in file usb_host_hid_interface.c to retrive data from the report received from the device. Application can also access these details to retrive the intended information incase provided interface function fail to do so.

1.4.2.3.2.16 USB_HID_RPT_DESC_ERROR Enumeration**File**

usb_host_hid_parser.h

Syntax

```
typedef enum {
    HID_ERR = 0,
    HID_ERR_NotEnoughMemory,
    HID_ERR_NullPointer,
    HID_ERR_UnexpectedEndCollection,
    HID_ERR_UnexpectedPop,
    HID_ERR_MissingEndCollection,
    HID_ERR_MissingTopLevelCollection,
    HID_ERR_NoReports,
    HID_ERR_UnmatchedUsageRange,
    HID_ERR_UnmatchedStringRange,
    HID_ERR_UnmatchedDesignatorRange,
    HID_ERR_UnexpectedEndOfDescriptor,
    HID_ERR_BadLogicalMin,
    HID_ERR_BadLogicalMax,
    HID_ERR_BadLogical,
    HID_ERR_ZeroReportSize,
    HID_ERR_ZeroReportID,
    HID_ERR_ZeroReportCount,
    HID_ERR_BadUsageRangePage,
    HID_ERR_BadUsageRange
} USB_HID_RPT_DESC_ERROR;
```

Members

Members	Description
HID_ERR = 0	No error
HID_ERR_NotEnoughMemory	If not enough Heap can be allocated, make sure sufficient dynamic memory is allocated for the parser
HID_ERR_NullPointer	Pointer to report descriptor is NULL
HID_ERR_UnexpectedEndCollection	End of collection not expected
HID_ERR_UnexpectedPop	POP not expected
HID_ERR_MissingEndCollection	No end of collection found
HID_ERR_MissingTopLevelCollection	Atleast one collection must be present
HID_ERR_NoReports	atlest one report must be present
HID_ERR_UnmatchedUsageRange	Either Minimum or Maximum for usage range missing
HID_ERR_UnmatchedStringRange	Either Minimum or Maximum for string range missing
HID_ERR_UnmatchedDesignatorRange	Either Minimum or Maximum for designator range missing
HID_ERR_UnexpectedEndOfDescriptor	Report descriptor not formatted properly
HID_ERR_BadLogicalMin	Logical Min greater than report size
HID_ERR_BadLogicalMax	Logical Max greater than report size
HID_ERR_BadLogical	If logical Min is greater than Max
HID_ERR_ZeroReportSize	Report size is zero
HID_ERR_ZeroReportID	report ID is zero
HID_ERR_ZeroReportCount	Number of reports is zero
HID_ERR_BadUsageRangePage	Bad Usage page range
HID_ERR_BadUsageRange	Bad Usage range

Description

HID parser error codes

This enumerates the error encountered during the parsing of report descriptor. In case of any error parsing is stopped and the error is flagged. Device is not attached successfully.

1.4.2.3.2.17 deviceRptInfo Variable

File

usb_host_hid_parser.h

Syntax

```
USB_HID_DEVICE_RPT_INFO deviceRptInfo;
```

Section

External Variables

1.4.2.3.2.18 itemListPtrs Variable

File

usb_host_hid_parser.h

Syntax

```
USB_HID_ITEM_LIST itemListPtrs;
```

Description

This is variable itemListPtrs.

1.4.2.3.2.19 DEVICE_CLASS_HID Macro

File

usb_host_hid.h

Syntax

```
#define DEVICE_CLASS_HID 0x03 // HID Interface Class Code
```

Description

HID Interface Class Code

1.4.2.3.2.20 DSC_HID Macro

File

usb_host_hid.h

Syntax

```
#define DSC_HID 0x21 // HID Descriptor Code
```

Description

HID Descriptor Code

1.4.2.3.2.21 DSC_PHY Macro

File

usb_host_hid.h

Syntax

```
#define DSC_PHY 0x23 // Physical Descriptor Code
```

Description

Physical Descriptor Code

1.4.2.3.2.22 EVENT_HID_ATTACH Macro

File

usb_host_hid.h

Syntax

```
#define EVENT_HID_ATTACH EVENT_HID_BASE + EVENT_HID_OFFSET + 7
```

Description

A HID device has attached. The returned data pointer points to a USB_DEVICE_ID structure.

1.4.2.3.2.23 EVENT_HID_BAD_REPORT_DESCRIPTOR Macro

File

usb_host_hid.h

Syntax

```
#define EVENT_HID_BAD_REPORT_DESCRIPTOR EVENT_HID_BASE + EVENT_HID_OFFSET + 9
```

Description

There was a problem parsing the report descriptor of the attached device. Communication with the device is not allowed, and the device should be detached.

1.4.2.3.2.24 EVENT_HID_DETACH Macro

File

usb_host_hid.h

Syntax

```
#define EVENT_HID_DETACH EVENT_HID_BASE + EVENT_HID_OFFSET + 8
```

Description

A HID device has detached. The returned data pointer points to a byte with the previous address of the detached device.

1.4.2.3.2.25 EVENT_HID_NONE Macro

File

usb_host_hid.h

Syntax

```
#define EVENT_HID_NONE EVENT_HID_BASE + EVENT_HID_OFFSET + 0
```

Description

No event occurred (NULL event)

1.4.2.3.2.26 EVENT_HID_OFFSET Macro

File

usb_host_hid.h

Syntax

```
#define EVENT_HID_OFFSET 0
```

Description

If the application has not defined an offset for HID events, set it to 0.

1.4.2.3.2.27 EVENT_HID_READ_DONE Macro

File

usb_host_hid.h

Syntax

```
#define EVENT_HID_READ_DONE EVENT_HID_BASE + EVENT_HID_OFFSET + 4
```

Description

define EVENT_HID_TRANSFER EVENT_HID_BASE + EVENT_HID_OFFSET + 3 // Unused - value retained for legacy. A HID Read transfer has completed. The returned data pointer points to a HID_TRANSFER_DATA structure, with information about the transfer.

1.4.2.3.2.28 EVENT_HID_RESET Macro

File

usb_host_hid.h

Syntax

```
#define EVENT_HID_RESET EVENT_HID_BASE + EVENT_HID_OFFSET + 6
```

Description

HID reset complete. The returned data pointer is NULL.

1.4.2.3.2.29 EVENT_HID_RESET_ERROR Macro

File

usb_host_hid.h

Syntax

```
#define EVENT_HID_RESET_ERROR EVENT_HID_BASE + EVENT_HID_OFFSET + 10
```

Description

An error occurred while trying to do a HID reset. The returned data pointer is NULL.

1.4.2.3.2.30 EVENT_HID_RPT_DESC_PARSED Macro

File

usb_host_hid.h

Syntax

```
#define EVENT_HID_RPT_DESC_PARSED EVENT_HID_BASE + EVENT_HID_OFFSET + 1
```

Description

A Report Descriptor has been parsed. The returned data pointer is NULL. The application must collect details, or simply return true if the application is already aware of the data format.

1.4.2.3.2.31 EVENT_HID_WRITE_DONE Macro

File

usb_host_hid.h

Syntax

```
#define EVENT_HID_WRITE_DONE EVENT_HID_BASE + EVENT_HID_OFFSET + 5
```

Description

A HID Write transfer has completed. The returned data pointer points to a HID_TRANSFER_DATA structure, with information about the transfer.

1.4.2.3.2.32 USB_HID_CLASS_ERROR Macro

File

usb_host_hid.h

Syntax

```
#define USB_HID_CLASS_ERROR USB_ERROR_CLASS_DEFINED
```

Section

HID Class Error Codes

1.4.2.3.3 usb_host_hid.h

Enumerations

Name	Description
USB_HOST_HID_RETURN_CODES	This is type USB_HOST_HID_RETURN_CODES.

Functions

	Name	Description
💡	USBHostHID_ApiFindBit	This function is used to locate a specific button or indicator. Once the report descriptor is parsed by the HID layer without any error, data from the report descriptor is stored in pre defined dat structures. This function traverses these data structure and exract data required by application
💡	USBHostHID_ApiFindValue	Find a specific Usage Value. Once the report descriptor is parsed by the HID layer without any error, data from the report descriptor is stored in pre defined dat structures. This function traverses these data structure and exract data required by application.
💡	USBHostHID_ApiGetCurrentInterfaceNum	This function reurns the interface number of the cuurent report descriptor parsed. This function must be called to fill data interface detail data structure and passed as parameter when requesinf for report transfers.
💡	USBHostHID_ApiImportData	This function can be used by application to extract data from the input reports. On receiving the input report from the device application can call the function with required inputs 'HID_DATA_DETAILS'.
💡	USBHostHIDDeviceDetect	This function determines if a HID device is attached and ready to use.
💡	USBHostHIDDeviceStatus	
💡	USBHostHIDEEventHandler	This function is the event handler for this client driver.
💡	USBHostHIDInitialize	This function is the initialization routine for this client driver.
💡	USBHostHIDRead	This function starts a Get report transfer reuest from the device, utilizing the function USBHostHIDTransfer();
💡	USBHostHIDReadIsComplete	This function indicates whether or not the last read request is complete.
💡	USBHostHIDReadTerminate	This function terminates a read request that is in progress.
💡	USBHostHIDResetDevice	This function starts a HID reset.
💡	USBHostHIDTasks	This function performs the maintenance tasks required by HID class
💡	USBHostHIDWrite	This function starts a Set report transfer request to the device, utilizing the function USBHostHIDTransfer();
💡	USBHostHIDWritIsComplete	This function indicates whether or not the last write request is complete.
💡	USBHostHIDWriteTerminate	This function terminates a write request that is in progress.

Macros

Name	Description
DEVICE_CLASS_HID	HID Interface Class Code
DSC_HID	HID Descriptor Code
DSC_PHY	Physical Descriptor Code
DSC_RPT_wValue	Report Descriptor Code, used for USBHostIssueDeviceRequest
EVENT_HID_ATTACH	A HID device has attached. The returned data pointer points to a USB_HID_DEVICE_ID structure.
EVENT_HID_BAD_REPORT_DESCRIPTOR	There was a problem parsing the report descriptor of the attached device. Communication with the device is not allowed, and the device should be detached.
EVENT_HID_DETACH	A HID device has detached. The returned data pointer points to a byte with the previous address of the detached device.
EVENT_HID_NONE	No event occurred (NULL event)
EVENT_HID_OFFSET	If the application has not defined an offset for HID events, set it to 0.
EVENT_HID_READ_DONE	define EVENT_HID_TRANSFER EVENT_HID_BASE + EVENT_HID_OFFSET + 3 // Unused - value retained for legacy. A HID Read transfer has completed. The returned data pointer points to a HID_TRANSFER_DATA structure, with information about the transfer.
EVENT_HID_RESET	HID reset complete. The returned data pointer is NULL.
EVENT_HID_RESET_ERROR	An error occurred while trying to do a HID reset. The returned data pointer is NULL.
EVENT_HID_RPT_DESC_PARSED	A Report Descriptor has been parsed. The returned data pointer is NULL. The application must collect details, or simply return true if the application is already aware of the data format.
EVENT_HID_WRITE_DONE	A HID Write transfer has completed. The returned data pointer points to a HID_TRANSFER_DATA structure, with information about the transfer.
USB_HID_CLASS_ERROR	
USB_HID_TRANSFER_IN	
USB_HID_TRANSFER_OUT	This is macro USB_HID_TRANSFER_OUT.
USBHostHID_GetCurrentReportInfo	This function returns a pointer to the current report info structure.
USBHostHID_GetItemListPointers	This function returns a pointer to list of item pointers stored in a structure.

Module

HID Client Driver

Structures

	Name	Description
◆	_HID_DATA_DETAILS	HID Data Details This structure defines the objects used by the application to access required report. Application must use parser interface functions to fill these details. e.g. USBHostHID_ApiFindValue
◆	_HID_TRANSFER_DATA	HID Transfer Information This structure is used when the event handler is used to notify the upper layer of transfer completion (EVENT_HID_READ_DONE or EVENT_HID_WRITE_DONE).
◆	_USB_HID_DEVICE_ID	HID Device ID Information This structure contains identification information about an attached device.
	HID_DATA_DETAILS	HID Data Details This structure defines the objects used by the application to access required report. Application must use parser interface functions to fill these details. e.g. USBHostHID_ApiFindValue

	HID_TRANSFER_DATA	HID Transfer Information This structure is used when the event handler is used to notify the upper layer of transfer completion (EVENT_HID_READ_DONE or EVENT_HID_WRITE_DONE).
	USB_HID_DEVICE_ID	HID Device ID Information This structure contains identification information about an attached device.

Types

Name	Description
HID_USER_DATA_SIZE	HID User Data Size This defines the data type required to hold the maximum field size data. Maximum size of data field within a report

Description

This is file usb_host_hid.h.

1.4.2.3.4 USB_HOST_HID_RETURN_CODES Enumeration

File

usb_host_hid.h

Syntax

```
typedef enum {
    USB_HID_COMMAND_PASSED = USB_SUCCESS,
    USB_HID_COMMAND_FAILED = USB_HID_CLASS_ERROR,
    USB_HID_PHASE_ERROR,
    USB_HID_DEVICE_NOT_FOUND,
    USB_HID_DEVICE_BUSY,
    USB_HID_NO_REPORT_DESCRIPTOR,
    USB_HID_INTERFACE_ERROR,
    USB_HID_REPORT_DESCRIPTOR_BAD,
    USB_HID_RESET_ERROR,
    USB_HID_ILLEGAL_REQUEST,
    USB_HID_DEVICE_DETACHED,
    USB_HID_INITIALIZING,
    USB_PROCESSING_REPORT_DESCRIPTOR,
    USB_HID_NORMAL_RUNNING,
    USB_HID_DEVICE_HOLDING,
    USB_HID_RESETTING_DEVICE
} USB_HOST_HID_RETURN_CODES;
```

Members

Members	Description
USB_HID_COMMAND_PASSED = USB_SUCCESS	Command was successful.
USB_HID_COMMAND_FAILED = USB_HID_CLASS_ERROR	Command failed at the device.
USB_HID_PHASE_ERROR	Command had a phase error at the device.
USB_HID_DEVICE_NOT_FOUND	Device with the specified address is not available.
USB_HID_DEVICE_BUSY	A transfer is currently in progress.
USB_HID_NO_REPORT_DESCRIPTOR	No report descriptor found
USB_HID_INTERFACE_ERROR	The interface layer cannot support the device.
USB_HID_REPORT_DESCRIPTOR_BAD	Report Descriptor for not proper
USB_HID_RESET_ERROR	An error occurred while resetting the device.
USB_HID_ILLEGAL_REQUEST	Cannot perform requested operation.

Module

HID Client Driver

Description

This is type USB_HOST_HID_RETURN_CODES.

1.4.2.3.5 _USB_HOST_HID_PARSER_H_ Macro**File**

usb_host_hid_parser.h

Syntax

```
#define _USB_HOST_HID_PARSER_H_
```

Module

HID Client Driver

Description

usb_host_hid_parser.h

1.4.2.3.6 usb_host_hid_parser.h**Enumerations**

Name	Description
HIDReportTypeEnum	This is type HIDReportTypeEnum.
USB_HID_RPT_DESC_ERROR	HID parser error codes This enumerates the error encountered during the parsing of report descriptor. In case of any error parsing is stopped and the error is flagged. Device is not attached successfully.

Functions

	Name	Description
!	USBHostHID_HasUsage	This function is used to locate the usage in a report descriptor. Function will look into the data structures created by the HID parser and return the appropriate location.

Macros

Name	Description
_USB_HOST_HID_PARSER_H_	usb_host_hid_parser.h

Module

HID Client Driver

Structures

	Name	Description
!	_HID_COLLECTION	HID Collection Details This structure contains information about each collection encountered in the report descriptor.
!	_HID_GLOBALS	HID Global Item Information This structure contains information about each Global Item of the report descriptor.
!	_HID_ITEM_INFO	HID Item Information This structure contains information about each Item of the report descriptor.
!	_HID_REPORT	HID Report details This structure contains information about each report exchanged with the device.

	_HID_REPORTITEM	HID Report Details This structure contains information about each Report encountered in the report descriptor.
	_HID_STRINGITEM	HID String Item Details This structure contains information about each Report encountered in the report descriptor.
	_HID_USAGEITEM	HID Report Details This structure contains information about each Usage Item encountered in the report descriptor.
	_USB_HID_DEVICE_RPT_INFO	Report Descriptor Information This structure contains top level information of the report descriptor. This information is important and is used to understand the information during the course of parsing. This structure also stores temporary data needed during parsing the report descriptor. All of this information may not be of much importance to the application.
	_USB_HID_ITEM_LIST	List of Items This structure contains array of pointers to all the Items in the report descriptor. HID parser will populate the lists while parsing the report descriptor. This data is used by interface functions provided in file usb_host_hid_interface.c to retrieve data from the report received from the device. Application can also access these details to retrieve the intended information incase provided interface function fail to do so.
	HID_COLLECTION	HID Collection Details This structure contains information about each collection encountered in the report descriptor.
	HID_DESIGITEM	HID String Item Details This structure contains information about each Report encountered in the report descriptor.
	HID_GLOBALS	HID Global Item Information This structure contains information about each Global Item of the report descriptor.
	HID_ITEM_INFO	HID Item Information This structure contains information about each Item of the report descriptor.
	HID_REPORT	HID Report details This structure contains information about each report exchanged with the device.
	HID_REPORTITEM	HID Report Details This structure contains information about each Report encountered in the report descriptor.
	HID_STRINGITEM	HID String Item Details This structure contains information about each Report encountered in the report descriptor.
	HID_USAGEITEM	HID Report Details This structure contains information about each Usage Item encountered in the report descriptor.
	USB_HID_DEVICE_RPT_INFO	Report Descriptor Information This structure contains top level information of the report descriptor. This information is important and is used to understand the information during the course of parsing. This structure also stores temporary data needed during parsing the report descriptor. All of this information may not be of much importance to the application.

	USB_HID_ITEM_LIST	List of Items This structure contains array of pointers to all the items in the report descriptor. HID parser will populate the lists while parsing the report descriptor. This data is used by interface functions provided in file <code>usb_host_hid_interface.c</code> to retrieve data from the report received from the device. Application can also access these details to retrieve the intended information incase provided interface function fail to do so.
--	--------------------------	---

Variables

Name	Description
<code>deviceRptInfo</code>	
<code>itemListPtrs</code>	This is variable <code>itemListPtrs</code> .

Description

This is file `usb_host_hid_parser.h`.

1.4.2.3.7 DSC_RPT_wValue Macro**File**

`usb_host_hid.h`

Syntax

```
#define DSC_RPT_wValue 0x2200 // Report Descriptor Code, used for USBHostIssueDeviceRequest
```

Module

HID Client Driver

Description

Report Descriptor Code, used for `USBHostIssueDeviceRequest`

1.4.2.3.8 USB_HID_TRANSFER_IN Macro**File**

`usb_host_hid.h`

Syntax

```
#define USB_HID_TRANSFER_IN 1
```

Module

HID Client Driver

Section

Constants

1.4.2.3.9 USB_HID_TRANSFER_OUT Macro**File**

`usb_host_hid.h`

Syntax

```
#define USB_HID_TRANSFER_OUT 0
```

Module

HID Client Driver

Description

This is macro USB_HID_TRANSFER_OUT.

1.4.2.4 Mass Storage Client Driver

This client driver provides USB Embedded Host support for mass storage devices.

Files

Name	Description
usb_host_msd.h	This is file usb_host_msd.h.

Macros

Name	Description
_USBHOSTMSD_H_	This is macro _USBHOSTMSD_H_.

Description

This client driver provides USB Embedded Host support for mass storage devices. Mass storage devices use USB Bulk transfers to efficiently transfer large amounts of data. Bulk transfers may utilize all remaining bandwidth on the bus after all of the Control, Interrupt, and Isochronous transfers for the frame have completed. The exact amount of time required for a bulk transfer will depend on the amount of other traffic that is on the bus. Therefore, Bulk transfers should be used only for non-time critical operations.

This implementation of the Mass Storage Class provides support for the Bulk Only Transport.

See [AN1142 - USB Mass Storage Class on an Embedded Host](#) for more information about the Mass Storage Class and this client driver.

1.4.2.4.1 Functions

Functions

	Name	Description
✳️	USBHostMSDDeviceStatus	This function determines the status of a mass storage device.
✳️	USBHostMSDEventHandler	This function is the event handler for this client driver.
✳️	USBHostMSDInitialize	This function is the initialization routine for this client driver.
✳️	USBHostMSDResetDevice	This function starts a bulk-only mass storage reset.
✳️	USBHostMSDTasks	This function performs the maintenance tasks required by the mass storage class.
✳️	USBHostMSDTerminateTransfer	This function terminates a mass storage transfer.
✳️	USBHostMSDTransfer	This function starts a mass storage transfer.
✳️	USBHostMSDTransferIsComplete	This function indicates whether or not the last transfer is complete.

Macros

Name	Description
USBHostMSDRead	This function starts a mass storage read, utilizing the function USBHostMSDTransfer();
USBHostMSDWrite	This function starts a mass storage write, utilizing the function USBHostMSDTransfer();

Module

Mass Storage Client Driver

Description

1.4.2.4.1.1 USBHostMSDDeviceStatus Function

File

usb_host_msd.h

Syntax

```
uint8_t USBHostMSDDeviceStatus(uint8_t deviceAddress);
```

Description

This function determines the status of a mass storage device.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_MSD_DEVICE_NOT_FOUND	Illegal device address, or the device is not an MSD
USB_MSD_INITIALIZING	MSD is attached and in the process of initializing
USB_MSD_NORMAL_RUNNING	MSD is in normal running mode
USB_MSD_RESETTING_DEVICE	MSD is resetting
USB_MSD_DEVICE_DETACHED	MSD detached. Should not occur
USB_MSD_ERROR_STATE	MSD is holding due to an error. No communication is allowed.
Other	Return codes from USBHostDeviceStatus() will also be returned if the device is in the process of enumerating.

Function

```
uint8_t USBHostMSDDeviceStatus( uint8_t deviceAddress )
```

1.4.2.4.1.2 USBHostMSDEventHandler Function

This function is the event handler for this client driver.

File

usb_host_msd.h

Syntax

```
bool USBHostMSDEventHandler(uint8_t address, USB_EVENT event, void * data, uint32_t size);
```

Description

This function is the event handler for this client driver. It is called by the host layer when various events occur.

Remarks

None

Preconditions

The device has been initialized.

Return Values

Return Values	Description
true	Event was handled
false	Event was not handled

Function

```
bool USBHostMSDEventHandler( uint8_t address, USB_EVENT event,
void *data, uint32_t size )
```

1.4.2.4.1.3 USBHostMSDInitialize Function

This function is the initialization routine for this client driver.

File

`usb_host_msd.h`

Syntax

```
bool USBHostMSDInitialize(uint8_t address, uint32_t flags, uint8_t clientDriverID);
```

Description

This function is the initialization routine for this client driver. It is called by the host layer when the USB device is being enumerated. For a mass storage device, we need to make sure that we have room for a new device, and that the device has at least one bulk IN and one bulk OUT endpoint.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
true	We can support the device.
false	We cannot support the device.

Function

```
bool USBHostMSDInitialize( uint8_t address, uint32_t flags, uint8_t clientDriverID )
```

1.4.2.4.1.4 USBHostMSDResetDevice Function

This function starts a bulk-only mass storage reset.

File

`usb_host_msd.h`

Syntax

```
uint8_t USBHostMSDResetDevice(uint8_t deviceAddress);
```

Description

This function starts a bulk-only mass storage reset. A reset can be issued only if the device is attached and not being initialized.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_SUCCESS	Reset started

USB_MSD_DEVICE_NOT_FOUND	No device with specified address
USB_MSD_ILLEGAL_REQUEST	Device is in an illegal state for reset

Function

```
uint8_t USBHostMSDResetDevice( uint8_t deviceAddress )
```

1.4.2.4.1.5 USBHostMSDTasks Function

This function performs the maintenance tasks required by the mass storage class.

File

```
usb_host_msd.h
```

Syntax

```
void USBHostMSDTasks( );
```

Returns

None

Description

This function performs the maintenance tasks required by the mass storage class. If transfer events from the host layer are not being used, then it should be called on a regular basis by the application. If transfer events from the host layer are being used, this function is compiled out, and does not need to be called.

Remarks

None

Preconditions

USBHostMSDInitialize() has been called.

Parameters

Parameters	Description
None	None

Function

```
void USBHostMSDTasks( void )
```

1.4.2.4.1.6 USBHostMSDTerminateTransfer Function**File**

```
usb_host_msd.h
```

Syntax

```
void USBHostMSDTerminateTransfer(uint8_t deviceAddress);
```

Returns

None

Description

This function terminates a mass storage transfer.

Remarks

After executing this function, the application may have to reset the device in order for the device to continue working properly.

Preconditions

None

Function

```
void USBHostMSDTerminateTransfer( uint8_t deviceAddress )
```

1.4.2.4.1.7 USBHostMSDTransfer Function

This function starts a mass storage transfer.

File

```
usb_host_msd.h
```

Syntax

```
uint8_t USBHostMSDTransfer(uint8_t deviceAddress, uint8_t deviceLUN, uint8_t direction,
                           uint8_t * commandBlock, uint8_t commandBlockLength, uint8_t * data, uint32_t dataLength);
```

Description

This function starts a mass storage transfer. Usually, applications will probably utilize a read/write wrapper to access this function.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_SUCCESS	Request started successfully
USB_MSD_DEVICE_NOT_FOUND	No device with specified address
USB_MSD_DEVICE_BUSY	Device not in proper state for performing a transfer
USB_MSD_INVALID_LUN	Specified LUN does not exist

Function

```
uint8_t USBHostMSDTransfer( uint8_t deviceAddress, uint8_t deviceLUN,
                            uint8_t direction, uint8_t *commandBlock, uint8_t commandBlockLength,
                            uint8_t *data, uint32_t dataLength )
```

1.4.2.4.1.8 USBHostMSDTransferIsComplete Function

This function indicates whether or not the last transfer is complete.

File

```
usb_host_msd.h
```

Syntax

```
bool USBHostMSDTransferIsComplete(uint8_t deviceAddress, uint8_t * errorCode, uint32_t * byteCount);
```

Description

This function indicates whether or not the last transfer is complete. If the functions returns true, the returned byte count and error code are valid. Since only one transfer can be performed at once and only one endpoint can be used, we only need to know the device address.

Remarks

None

Preconditions

None

Return Values

Return Values	Description
true	Transfer is complete, errorCode is valid
false	Transfer is not complete, errorCode is not valid

Function

```
bool USBHostMSDTransferIsComplete( uint8_t deviceAddress,
                                    uint8_t *errorCode, uint32_t *byteCount )
```

1.4.2.4.1.9 USBHostMSDRead Macro**File**

usb_host_msd.h

Syntax

```
#define USBHostMSDRead(
    deviceAddress, deviceLUN, commandBlock, commandBlockLength, data, dataLength ) \
    USBHostMSDTransfer( deviceAddress, deviceLUN, 1, commandBlock, commandBlockLength,
                        data, dataLength )
```

Description

This function starts a mass storage read, utilizing the function USBHostMSDTransfer();

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_SUCCESS	Request started successfully
USB_MSD_DEVICE_NOT_FOUND	No device with specified address
USB_MSD_DEVICE_BUSY	Device not in proper state for performing a transfer
USB_MSD_INVALID_LUN	Specified LUN does not exist

Function

```
uint8_t USBHostMSDRead( uint8_t deviceAddress, uint8_t deviceLUN, uint8_t *commandBlock,
                        uint8_t commandBlockLength, uint8_t *data, uint32_t dataLength );
```

1.4.2.4.1.10 USBHostMSDWrite Macro**File**

usb_host_msd.h

Syntax

```
#define USBHostMSDWrite(
    deviceAddress, deviceLUN, commandBlock, commandBlockLength, data, dataLength ) \
    USBHostMSDTransfer( deviceAddress, deviceLUN, 0, commandBlock, commandBlockLength,
                        data, dataLength )
```

Description

This function starts a mass storage write, utilizing the function USBHostMSDTransfer();

Remarks

None

Preconditions

None

Return Values

Return Values	Description
USB_SUCCESS	Request started successfully
USB_MSD_DEVICE_NOT_FOUND	No device with specified address
USB_MSD_DEVICE_BUSY	Device not in proper state for performing a transfer
USB_MSD_INVALID_LUN	Specified LUN does not exist

Function

```
uint8_t USBHostMSDWrite( uint8_t deviceAddress, uint8_t deviceLUN, uint8_t *commandBlock,
                         uint8_t commandBlockLength, uint8_t *data, uint32_t dataLength );
```

1.4.2.4.2 Data Types and Constants**Macros**

Name	Description
DEVICE_CLASS_MASS_STORAGE	Class code for Mass Storage.
DEVICE_INTERFACE_PROTOCOL_BULK_ONLY	Protocol code for Bulk-only mass storage.
DEVICE_SUBCLASS_CD_DVD	SubClass code for a CD/DVD drive (not supported).
DEVICE_SUBCLASS_FLOPPY_INTERFACE	SubClass code for a floppy disk interface (not supported).
DEVICE_SUBCLASS_RBC	SubClass code for Reduced Block Commands (not supported).
DEVICE_SUBCLASS_REMOVABLE	SubClass code for removable media (not supported).
DEVICE_SUBCLASS_SCSI	SubClass code for a SCSI interface device (supported).
DEVICE_SUBCLASS_TAPE_DRIVE	SubClass code for a tape drive (not supported).
EVENT_MSD_ATTACH	MSD device has attached
EVENT_MSD_MAX_LUN	Set maximum LUN for the device
EVENT_MSD_NONE	No event occurred (NULL event)
EVENT_MSD_OFFSET	If the application has not defined an offset for MSD events, set it to 0.
EVENT_MSD_RESET	MSD reset complete
EVENT_MSD_TRANSFER	A MSD transfer has completed
MSD_COMMAND_FAILED	Transfer failed. Returned in dCSWStatus.
MSD_COMMAND_PASSED	Transfer was successful. Returned in dCSWStatus.
MSD_PHASE_ERROR	Transfer phase error. Returned in dCSWStatus.
USB_MSD_CBW_ERROR	The CBW was not transferred successfully.
USB_MSD_COMMAND_FAILED	Command failed at the device.
USB_MSD_COMMAND_PASSED	Command was successful.
USB_MSD_CSW_ERROR	The CSW was not transferred successfully.
USB_MSD_DEVICE_BUSY	A transfer is currently in progress.
USB_MSD_DEVICE_DETACHED	Device is detached.
USB_MSD_DEVICE_NOT_FOUND	Device with the specified address is not available.
USB_MSD_ERROR	Error code offset.
USB_MSD_ERROR_STATE	Device is holding due to a MSD error.
USB_MSD_ILLEGAL_REQUEST	Cannot perform requested operation.
USB_MSD_INITIALIZING	Device is initializing.

USB_MSD_INVALID_LUN	Invalid LUN specified.
USB_MSD_MEDIA_INTERFACE_ERROR	The media interface layer cannot support the device.
USB_MSD_NORMAL_RUNNING	Device is running and available for data transfers.
USB_MSD_OUT_OF_MEMORY	No dynamic memory is available.
USB_MSD_PHASE_ERROR	Command had a phase error at the device.
USB_MSD_RESET_ERROR	An error occurred while resetting the device.
USB_MSD_RESETTING_DEVICE	Device is being reset.

Module

Mass Storage Client Driver

Description**1.4.2.4.2.1 DEVICE_CLASS_MASS_STORAGE Macro****File**

usb_host_msd.h

Syntax

```
#define DEVICE_CLASS_MASS_STORAGE 0x08      // Class code for Mass Storage.
```

Description

Class code for Mass Storage.

1.4.2.4.2.2 DEVICE_INTERFACE_PROTOCOL_BULK_ONLY Macro**File**

usb_host_msd.h

Syntax

```
#define DEVICE_INTERFACE_PROTOCOL_BULK_ONLY 0x50      // Protocol code for Bulk-only mass storage.
```

Description

Protocol code for Bulk-only mass storage.

1.4.2.4.2.3 DEVICE_SUBCLASS_CD_DVD Macro**File**

usb_host_msd.h

Syntax

```
#define DEVICE_SUBCLASS_CD_DVD 0x02      // SubClass code for a CD/DVD drive (not supported).
```

Description

SubClass code for a CD/DVD drive (not supported).

1.4.2.4.2.4 DEVICE_SUBCLASS_FLOPPY_INTERFACE Macro**File**

usb_host_msd.h

Syntax

```
#define DEVICE_SUBCLASS_FLOPPY_INTERFACE 0x04      // SubClass code for a floppy disk interface (not supported).
```

Description

SubClass code for a floppy disk interface (not supported).

1.4.2.4.2.5 DEVICE_SUBCLASS_RBC Macro**File**

usb_host_msd.h

Syntax

```
#define DEVICE_SUBCLASS_RBC 0x01      // SubClass code for Reduced Block Commands (not supported).
```

Description

SubClass code for Reduced Block Commands (not supported).

1.4.2.4.2.6 DEVICE_SUBCLASS_REMOVABLE Macro**File**

usb_host_msd.h

Syntax

```
#define DEVICE_SUBCLASS_REMOVABLE 0x05      // SubClass code for removable media (not supported).
```

Description

SubClass code for removable media (not supported).

1.4.2.4.2.7 DEVICE_SUBCLASS_SCSI Macro**File**

usb_host_msd.h

Syntax

```
#define DEVICE_SUBCLASS_SCSI 0x06      // SubClass code for a SCSI interface device (supported).
```

Description

SubClass code for a SCSI interface device (supported).

1.4.2.4.2.8 DEVICE_SUBCLASS_TAPE_DRIVE Macro**File**

usb_host_msd.h

Syntax

```
#define DEVICE_SUBCLASS_TAPE_DRIVE 0x03      // SubClass code for a tape drive (not supported).
```

Description

SubClass code for a tape drive (not supported).

1.4.2.4.2.9 EVENT_MSD_ATTACH Macro**File**

usb_host_msd.h

Syntax

```
#define EVENT_MSD_ATTACH EVENT_MSD_BASE + EVENT_MSD_OFFSET + 4 // MSD device has attached
```

Description

MSD device has attached

1.4.2.4.2.10 EVENT_MSD_MAX_LUN Macro

File

usb_host_msd.h

Syntax

```
#define EVENT_MSD_MAX_LUN EVENT_MSD_BASE + EVENT_MSD_OFFSET + 3 // Set maximum LUN for  
the device
```

Description

Set maximum LUN for the device

1.4.2.4.2.11 EVENT_MSD_NONE Macro

File

usb_host_msd.h

Syntax

```
#define EVENT_MSD_NONE EVENT_MSD_BASE + EVENT_MSD_OFFSET + 0 // No event occurred (NULL  
event)
```

Description

No event occurred (NULL event)

1.4.2.4.2.12 EVENT_MSD_OFFSET Macro

File

usb_host_msd.h

Syntax

```
#define EVENT_MSD_OFFSET 0
```

Description

If the application has not defined an offset for MSD events, set it to 0.

1.4.2.4.2.13 EVENT_MSD_RESET Macro

File

usb_host_msd.h

Syntax

```
#define EVENT_MSD_RESET EVENT_MSD_BASE + EVENT_MSD_OFFSET + 2 // MSD reset complete
```

Description

MSD reset complete

1.4.2.4.2.14 EVENT_MSD_TRANSFER Macro

File

usb_host_msd.h

Syntax

```
#define EVENT_MSD_TRANSFER EVENT_MSD_BASE + EVENT_MSD_OFFSET + 1 // A MSD transfer has completed
```

Description

A MSD transfer has completed

1.4.2.4.2.15 MSD_COMMAND_FAILED Macro

File

usb_host_msd.h

Syntax

```
#define MSD_COMMAND_FAILED 0x01 // Transfer failed. Returned in dCSWStatus.
```

Description

Transfer failed. Returned in dCSWStatus.

1.4.2.4.2.16 MSD_COMMAND_PASSED Macro

File

usb_host_msd.h

Syntax

```
#define MSD_COMMAND_PASSED 0x00 // Transfer was successful. Returned in dCSWStatus.
```

Description

Transfer was successful. Returned in dCSWStatus.

1.4.2.4.2.17 MSD_PHASE_ERROR Macro

File

usb_host_msd.h

Syntax

```
#define MSD_PHASE_ERROR 0x02 // Transfer phase error. Returned in dCSWStatus.
```

Description

Transfer phase error. Returned in dCSWStatus.

1.4.2.4.2.18 USB_MSD_CBW_ERROR Macro

File

usb_host_msd.h

Syntax

```
#define USB_MSD_CBW_ERROR (USB_MSD_ERROR | 0x04) // The CBW was not transferred successfully.
```

Description

The CBW was not transferred successfully.

1.4.2.4.2.19 USB_MSD_COMMAND_FAILED Macro

File

usb_host_msd.h

Syntax

```
#define USB_MSD_COMMAND_FAILED (USB_MSD_ERROR | MSD_COMMAND_FAILED) // Command failed at the device.
```

Description

Command failed at the device.

1.4.2.4.2.20 USB_MSD_COMMAND_PASSED Macro

File

usb_host_msd.h

Syntax

```
#define USB_MSD_COMMAND_PASSED USB_SUCCESS // Command was successful.
```

Description

Command was successful.

1.4.2.4.2.21 USB_MSD_CSW_ERROR Macro

File

usb_host_msd.h

Syntax

```
#define USB_MSD_CSW_ERROR (USB_MSD_ERROR | 0x05) // The CSW was not transferred successfully.
```

Description

The CSW was not transferred successfully.

1.4.2.4.2.22 USB_MSD_DEVICE_BUSY Macro

File

usb_host_msd.h

Syntax

```
#define USB_MSD_DEVICE_BUSY (USB_MSD_ERROR | 0x07) // A transfer is currently in progress.
```

Description

A transfer is currently in progress.

1.4.2.4.2.23 USB_MSD_DEVICE_DETACHED Macro

File

usb_host_msd.h

Syntax

```
#define USB_MSD_DEVICE_DETACHED 0x50 // Device is detached.
```

Description

Device is detached.

1.4.2.4.2.24 USB_MSD_DEVICE_NOT_FOUND Macro

File

usb_host_msd.h

Syntax

```
#define USB_MSD_DEVICE_NOT_FOUND (USB_MSD_ERROR | 0x06)           // Device with the  
specified address is not available.
```

Description

Device with the specified address is not available.

1.4.2.4.2.25 USB_MSD_ERROR Macro

File

usb_host_msd.h

Syntax

```
#define USB_MSD_ERROR USB_ERROR_CLASS_DEFINED                  // Error code offset.
```

Description

Error code offset.

1.4.2.4.2.26 USB_MSD_ERROR_STATE Macro

File

usb_host_msd.h

Syntax

```
#define USB_MSD_ERROR_STATE 0x55      // Device is holding due to a MSD error.
```

Description

Device is holding due to a MSD error.

1.4.2.4.2.27 USB_MSD_ILLEGAL_REQUEST Macro

File

usb_host_msd.h

Syntax

```
#define USB_MSD_ILLEGAL_REQUEST (USB_MSD_ERROR | 0x0B)           // Cannot perform  
requested operation.
```

Description

Cannot perform requested operation.

1.4.2.4.2.28 USB_MSD_INITIALIZING Macro

File

usb_host_msd.h

Syntax

```
#define USB_MSD_INITIALIZING 0x51      // Device is initializing.
```

Description

Device is initializing.

1.4.2.4.2.29 USB_MSD_INVALID_LUN Macro

File

usb_host_msd.h

Syntax

```
#define USB_MSD_INVALID_LUN (USB_MSD_ERROR | 0x08)           // Invalid LUN specified.
```

Description

Invalid LUN specified.

1.4.2.4.2.30 USB_MSD_MEDIA_INTERFACE_ERROR Macro

File

usb_host_msd.h

Syntax

```
#define USB_MSD_MEDIA_INTERFACE_ERROR (USB_MSD_ERROR | 0x09)           // The media
interface layer cannot support the device.
```

Description

The media interface layer cannot support the device.

1.4.2.4.2.31 USB_MSD_NORMAL_RUNNING Macro

File

usb_host_msd.h

Syntax

```
#define USB_MSD_NORMAL_RUNNING 0x52           // Device is running and available for data
transfers.
```

Description

Device is running and available for data transfers.

1.4.2.4.2.32 USB_MSD_OUT_OF_MEMORY Macro

File

usb_host_msd.h

Syntax

```
#define USB_MSD_OUT_OF_MEMORY (USB_MSD_ERROR | 0x03)           // No dynamic memory is
available.
```

Description

No dynamic memory is available.

1.4.2.4.2.33 USB_MSD_PHASE_ERROR Macro

File

usb_host_msd.h

Syntax

```
#define USB_MSD_PHASE_ERROR (USB_MSD_ERROR | MSD_PHASE_ERROR)           // Command had a phase
error at the device.
```

Description

Command had a phase error at the device.

1.4.2.4.2.34 USB_MSD_RESET_ERROR Macro

File

usb_host_msd.h

Syntax

```
#define USB_MSD_RESET_ERROR (USB_MSD_ERROR | 0x0A)           // An error occurred while
resetting the device.
```

Description

An error occurred while resetting the device.

1.4.2.4.2.35 USB_MSD_RESETTING_DEVICE Macro

File

usb_host_msd.h

Syntax

```
#define USB_MSD_RESETTING_DEVICE 0x53      // Device is being reset.
```

Description

Device is being reset.

1.4.2.4.3 usb_host_msd.h

Functions

	Name	Description
♫	USBHostMSDDeviceStatus	This function determines the status of a mass storage device.
♫	USBHostMSDEventHandler	This function is the event handler for this client driver.
♫	USBHostMSDInitialize	This function is the initialization routine for this client driver.
♫	USBHostMSDResetDevice	This function starts a bulk-only mass storage reset.
♫	USBHostMSDTasks	This function performs the maintenance tasks required by the mass storage class.
♫	USBHostMSDTerminateTransfer	This function terminates a mass storage transfer.
♫	USBHostMSDTransfer	This function starts a mass storage transfer.
♫	USBHostMSDTransferIsComplete	This function indicates whether or not the last transfer is complete.

Macros

Name	Description
_USBHOSTMSD_H_	This is macro _USBHOSTMSD_H_.
DEVICE_CLASS_MASS_STORAGE	Class code for Mass Storage.
DEVICE_INTERFACE_PROTOCOL_BULK_ONLY	Protocol code for Bulk-only mass storage.
DEVICE_SUBCLASS_CD_DVD	SubClass code for a CD/DVD drive (not supported).
DEVICE_SUBCLASS_FLOPPY_INTERFACE	SubClass code for a floppy disk interface (not supported).
DEVICE_SUBCLASS_RBC	SubClass code for Reduced Block Commands (not supported).
DEVICE_SUBCLASS_REMOVABLE	SubClass code for removable media (not supported).
DEVICE_SUBCLASS_SCSI	SubClass code for a SCSI interface device (supported).
DEVICE_SUBCLASS_TAPE_DRIVE	SubClass code for a tape drive (not supported).
EVENT_MSD_ATTACH	MSD device has attached
EVENT_MSD_MAX_LUN	Set maximum LUN for the device
EVENT_MSD_NONE	No event occurred (NULL event)
EVENT_MSD_OFFSET	If the application has not defined an offset for MSD events, set it to 0.

EVENT_MSD_RESET	MSD reset complete
EVENT_MSD_TRANSFER	A MSD transfer has completed
MSD_COMMAND_FAILED	Transfer failed. Returned in dCSWStatus.
MSD_COMMAND_PASSED	Transfer was successful. Returned in dCSWStatus.
MSD_PHASE_ERROR	Transfer phase error. Returned in dCSWStatus.
USB_MSD_CBW_ERROR	The CBW was not transferred successfully.
USB_MSD_COMMAND_FAILED	Command failed at the device.
USB_MSD_COMMAND_PASSED	Command was successful.
USB_MSD_CSW_ERROR	The CSW was not transferred successfully.
USB_MSD_DEVICE_BUSY	A transfer is currently in progress.
USB_MSD_DEVICE_DETACHED	Device is detached.
USB_MSD_DEVICE_NOT_FOUND	Device with the specified address is not available.
USB_MSD_ERROR	Error code offset.
USB_MSD_ERROR_STATE	Device is holding due to a MSD error.
USB_MSD_ILLEGAL_REQUEST	Cannot perform requested operation.
USB_MSD_INITIALIZING	Device is initializing.
USB_MSD_INVALID_LUN	Invalid LUN specified.
USB_MSD_MEDIA_INTERFACE_ERROR	The media interface layer cannot support the device.
USB_MSD_NORMAL_RUNNING	Device is running and available for data transfers.
USB_MSD_OUT_OF_MEMORY	No dynamic memory is available.
USB_MSD_PHASE_ERROR	Command had a phase error at the device.
USB_MSD_RESET_ERROR	An error occurred while resetting the device.
USB_MSD_RESETTING_DEVICE	Device is being reset.
USBHostMSDRead	This function starts a mass storage read, utilizing the function USBHostMSDTransfer();
USBHostMSDWrite	This function starts a mass storage write, utilizing the function USBHostMSDTransfer();

Module

Mass Storage Client Driver

Description

This is file usb_host_msd.h.

1.4.2.4.4 _USBHOSTMSD_H_ Macro**File**

usb_host_msd.h

Syntax

#define _USBHOSTMSD_H_

Module

Mass Storage Client Driver

Description

This is macro _USBHOSTMSD_H_.

1.5 Demo Board Information

This section gives a brief introduction and links to more information for the USB demo boards.

Description

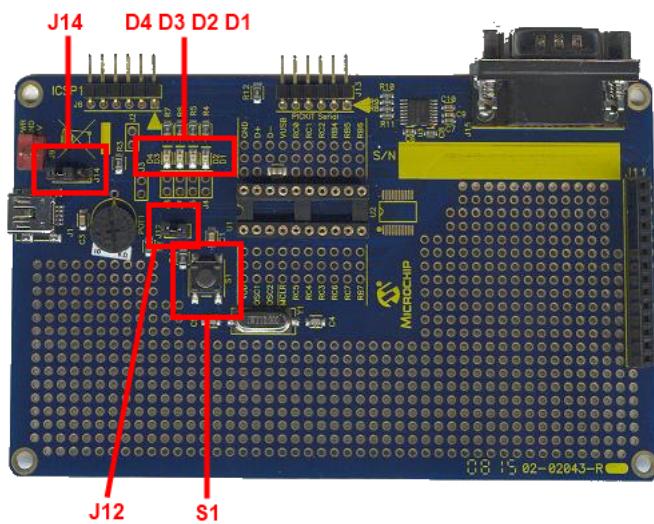
1.5.1 Low Pin Count USB Development Board

The low pin count USB development board serves as a base platform for the 20-pin USB products. This currently is the PIC18F14K50 family devices and the PIC16F145x family devices.

Description

Overview

This board features the [PIC18F14K50](#) microcontroller, but can also be used (preferably with minor modifications) with the [PIC16F145x](#) devices. The PIC18F14K50 controller has 20 pins, 16KB of flash, 768 bytes of RAM and an 8-bit core running up to 12MIPS.



J12 - Shorts the VUSB pin to Vdd rail. This jumper should always be left open, unless an 'LF' device is used, and the board VDD is externally supplied with a nominal 3.3V supply (ex: external power provided on J9, with the J14 jumpered in the leftmost position).

J14 - Selects the power source for the board. Short pins 1 and 2 to power from J9. Short pins 2 and 3 to power from the USB VBUS line.

S1 - Application button. Connected to RA3

D1 - Application LED. Connected to RC0

D2 - Application LED. Connected to RC1

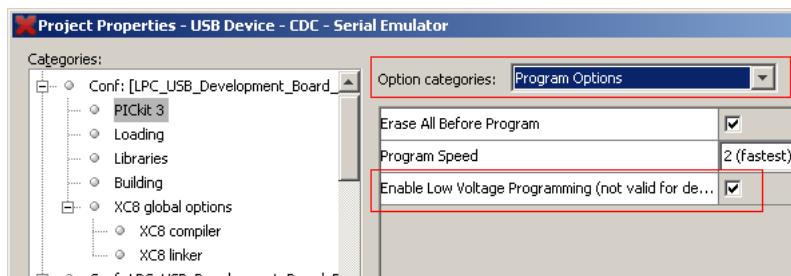
D3 - Application LED. Connected to RC2

D4 - Application LED. Connected to RC3

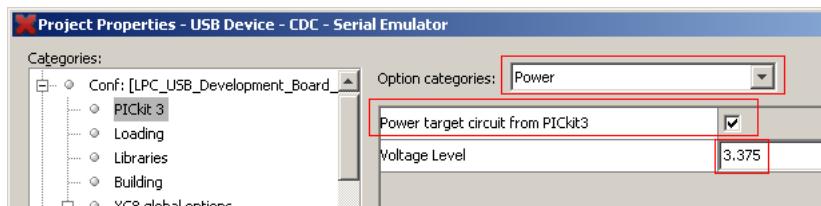
Using the Low Pin Count (LPC) USB Development Board with the PIC16F145x Devices

The original LPC USB Development board was designed for the PIC18F14K50, but the subsequent PIC16F145x USB microcontrollers have pinout backwards compatibility with the PIC18F14K50. Therefore, it is possible to use these PIC16F devices with the LPC USB Dev board. However, the board was not optimized for this newer device, and therefore there are several things that are useful to know when trying to use a PIC16F USB device with this board.

Programming the PIC16F145x Device on the LPC USB Dev Kit Board: The PIC16F145x microcontrollers feature two ICSP programming ports. One port (full programming and debug supported) is multiplexed with the MCLR/RC0/RC1 I/O pins. The other ICSP programming only port (no debug) is multiplexed with the MCLR/D+/D- I/O pins, so as to provide pinout backwards compatibility with the PIC18F14K50. By default, on the original version of the LPC Dev Kit board, only the MCLR/D+/D- programming interface is made available on the ICSP1 PICkit 3 style programming header. In order to program a PIC16F145x device using the ICSP1 header, it is required that the "Enable Low Voltage Programming" checkbox for the programmer device be selected in the MPLAB IDE build configuration settings. This is NOT the default setting (by default, high voltage programming is used instead, which the PIC16F145x silicon does not support through the MCLR/D+/D- ICSP port). This programmer option is under the "Program Options" option categories:



In addition to checking the low voltage programming check box, it is normally necessary to unplug the USB cable from the demo board (and USB host) during the programming operation, to minimize the capacitance and potential for I/O contention on the D+/D- pins, during the programming operation. Therefore, it is often convenient to program the microcontroller while the LPC Dev Kit board is being powered from the ICSP programmer (such as the PICkit 3):



Alternatively, if full program and debug operations are desired (using standard high voltage programming mode), it is recommended to connect the ICSP programmer to the MCLR/RC0/RC1 programming/debug port. This is the preferred ICSP port on the PIC16F145x devices, but they are not routed to a ICSP header on the original LPC Dev Kit Board (they are routed to header ICSP2 on the updated revision of the board). Therefore, if you have the original board, it is suggested to solder a new 6-pin standard male header (standard 100 mil pin spacing) to the prototyping area of the PCB, and then connect air wires to connect up to the MCLR, VDD, VSS, RC0, and RC1 pins. When using the MCLR/RC0/RC1 programming/debug port, it is not necessary to unplug the USB cable from the host during program/debug operations, and it is not necessary to power the board from the programmer or to use the low voltage programming mode.

Using a PIC16F145x Device on the LPC USB Dev Kit Board with the HFINTOSC+PLL+Active Clock Tuning: In order to use the HFINTOSC + PLL + Active Clock Tuning to operate in full speed USB mode, it is necessary for the VDD microcontroller supply rail to be stable and free of noise. However, the original LPC USB Dev Kit board does not have much VDD rail capacitance (0.2uF total), but it has a substantial noise generating source (the MAX3232 level translator chip, which uses build in capacitive charge pumps to generate positive voltages above VDD and negative voltages below VSS for RS232 level communication). The charge pumping action generates a substantial ripple/noise on the VDD rail, which can disrupt the HFINTOSC stability enough to cause USB communication issues (even though HSPLL mode is unaffected, as

the crystal is a resonant device that is harder to disrupt by noise). To fix this, it is necessary to add additional capacitance across the VDD/VSS nets on the LPC USB Development kit board. A value of 1uF to 8uF, preferably ceramic, is ideal, and provides good smoothing of the VDD rail noise. It is therefore recommended to solder a new 1-8uF ceramic capacitor (ex: 0603 or 0805) on top of the existing capacitor C1 on the demo board, to provide the VDD noise smoothing effect. Once this change has been made, the HFINTOSC + PLL + Active Clock Tuning may be used to successfully/reliably operate the microcontroller in USB full speed mode. If you have a newer revision PCB, this extra capacitance will already be populated on the PCB, and therefore, no soldering or other changes are required.

More Information

[Product webpage](#)

[PIC18F14K50 webpage](#)

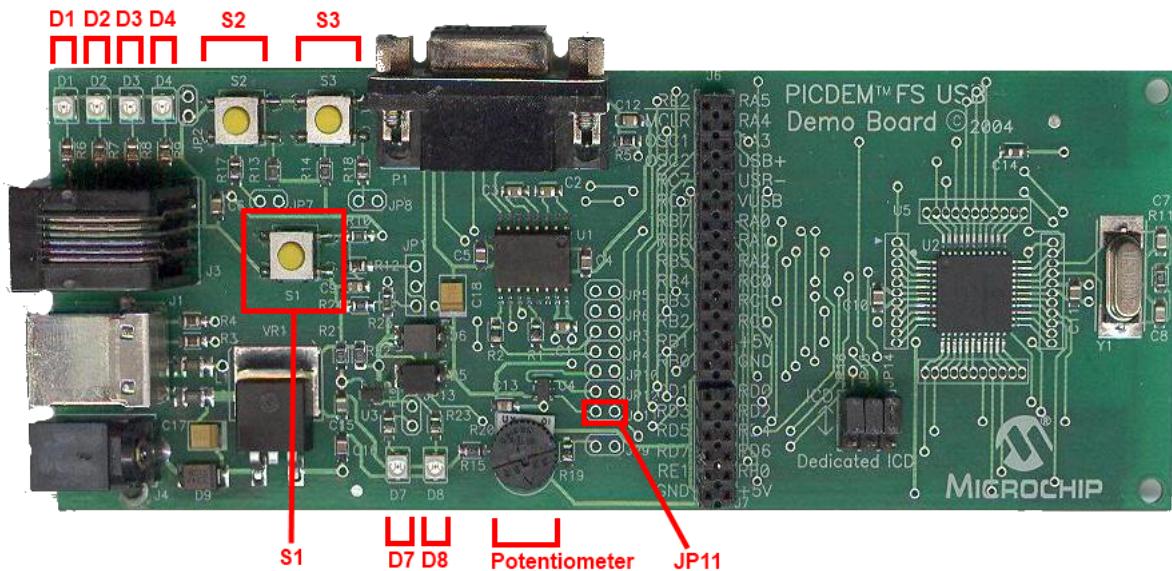
[PIC16F1459 webpage](#)

1.5.2 PICDEM FS USB Board

The PICDEM FS USB Board is the development platform for the PIC18F4550 family. It includes a temperature sensor, potentiometer, 2 buttons, 4 LEDs, and a PICtail connector.

Description

Overview



S1 - MCLR reset button

S2 - Application button

S3 - Application button

D1 - Application LED

D2 - Application LED

D3 - Application LED

D4 - Application LED

D7 - Bus powered indicator - When this LED is illuminated, the board is being powered by the USB bus.

D8 - Self powered indicator - When this LED is illuminated, the board is being powered by an external power supply.

JP11 - connects RB2 of the microcontroller to the temperature sensor on the board (U4). On some revisions of the board there is a trace shorting this jumper that needs to be cut in order to open this jumper.

More Information

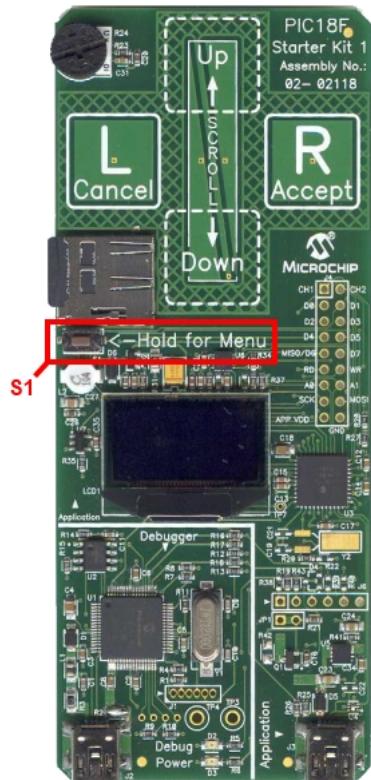
[Product website](#)

1.5.3 PIC18 Starter Kit

The PIC18F Starter Kit is a feature rich board with lots of on board features for customers to work with. The board includes an accelerometer, 2 capacitive touch buttons, 1 capacitive touch slider, 1 switch, an OLED, potentiometer, SD-card slot, and an on board debugger.

Description

Overview



S1 - Application switch. Connected to RB0.

More Information

[Product Website](#)

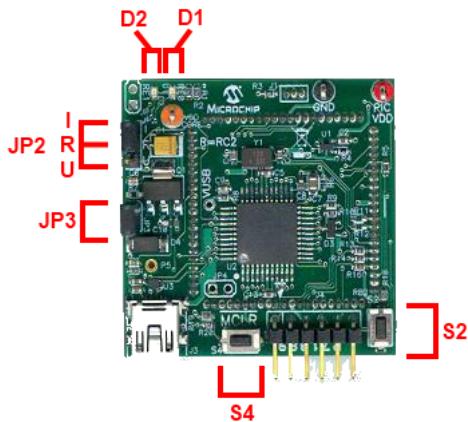
[Introduction Video](#)

1.5.4 PIC18F46J50 Plug-In-Module (PIM)

The PIC18F46J50 PIM services as a development platform for the PIC18F46J50 family. It includes 2 LEDs and a push button. It can also optionally be connected to the PIC18 Explorer board for access to additional features, such as potentiometer, PICtail, temperature sensor, etc.

Description

Overview



- JP2 - This is a three-pin header with the labels, "I", "R" and "U". The "R" is an abbreviation referring to microcontroller pin, RC2. "I" is an abbreviation referring to the "ICE" female header pin for the RC2 signal. "U" is an abbreviation for the USB VBUS line. When the jumper is in the "R" to "I" position, the RC2 pin connects only to the ICE female header pin, just like most of the other general purpose I/O pins. When the jumper is in the "R" to "U" position, RC2 (which is 5.5V tolerant) can be used to sense when the USB cable has been attached to the host, and when the host is actively providing power to the +5V VBUS line. According to the USB 2.0 specifications, no device should ever pull the D+ or D- lines high (such as with the D+ or D- pull-up resistor) until the host actively powers the +5V VBUS line. This is intended to prevent self-powered peripherals from ever sourcing even small amounts of power to the host when the host is not powered. Small amounts of current could potentially prevent the host (and possibly other USB peripherals connected to that host) from fully becoming depowered, which may cause problems during power-up and initialization. Self-powered peripherals should periodically monitor the +5V VBUS line and detect when it is driven high. Only when it is powered should user firmware enable the USB module and turn on the D+ (for full speed) or D- (for low speed) pull-up resistor, signaling device attach to the host. The recommended method of monitoring the +5V VBUS line is to connect it to one of the microcontroller's 5.5V tolerant I/O pins through a large value resistor (such as 100 kOhms). The resistor serves to improve the ESD ruggedness of the circuit as well as to prevent microcontroller damage if user firmware should ever unintentionally configure the I/O pin as an output. Peripherals which are purely bus powered obtain all of their power directly from the +5V VBUS line itself. For these types of devices, it is unnecessary to monitor when the VBUS is powered, as the peripheral will not be able to source current on the D+, D- or VBUS lines when the host is not powered.
- JP3 - This jumper is located in series with the +5V VBUS power supply line from the USB connector. When the jumper is removed, a current meter may be placed between the header pins to measure the board current which is being drawn from the USB port. Additionally, by removing the jumper cap altogether, JP3 provides a means of preventing the board from consuming USB power.
- S2 - Switch for application use. Tied to RB2.
- S4 - MCLR reset switch
- D1 - LED for application use. Tied to RE0.
- D2 - LED for application use. Tied to RE1.

More Information

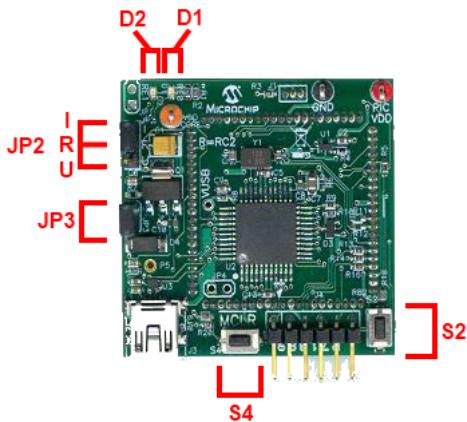
[Product webpage](#)

1.5.5 PIC18F47J53 Plug-In-Module (PIM)

The PIC18F47J50 PIM services as a development platform for the PIC18F47J50 family. It includes 2 LEDs and a push button. It can also optionally be connected to the PIC18 Explorer board for access to additional features, such as potentiometer, PICtail, temperature sensor, etc.

Description

Overview



- JP2 - This is a three-pin header with the labels, "I", "R" and "U". The "R" is an abbreviation referring to microcontroller pin, RC2. "I" is an abbreviation referring to the "ICE" female header pin for the RC2 signal. "U" is an abbreviation for the USB VBUS line. When the jumper is in the "R" to "I" position, the RC2 pin connects only to the ICE female header pin, just like most of the other general purpose I/O pins. When the jumper is in the "R" to "U" position, RC2 (which is 5.5V tolerant) can be used to sense when the USB cable has been attached to the host, and when the host is actively providing power to the +5V VBUS line. According to the USB 2.0 specifications, no device should ever pull the D+ or D- lines high (such as with the D+ or D- pull-up resistor) until the host actively powers the +5V VBUS line. This is intended to prevent self-powered peripherals from ever sourcing even small amounts of power to the host when the host is not powered. Small amounts of current could potentially prevent the host (and possibly other USB peripherals connected to that host) from fully becoming depowered, which may cause problems during power-up and initialization. Self-powered peripherals should periodically monitor the +5V VBUS line and detect when it is driven high. Only when it is powered should user firmware enable the USB module and turn on the D+ (for full speed) or D- (for low speed) pull-up resistor, signaling device attach to the host. The recommended method of monitoring the +5V VBUS line is to connect it to one of the microcontroller's 5.5V tolerant I/O pins through a large value resistor (such as 100 kOhms). The resistor serves to improve the ESD ruggedness of the circuit as well as to prevent microcontroller damage if user firmware should ever unintentionally configure the I/O pin as an output. Peripherals which are purely bus powered obtain all of their power directly from the +5V VBUS line itself. For these types of devices, it is unnecessary to monitor when the VBUS is powered, as the peripheral will not be able to source current on the D+, D- or VBUS lines when the host is not powered.
- JP3 - This jumper is located in series with the +5V VBUS power supply line from the USB connector. When the jumper is removed, a current meter may be placed between the header pins to measure the board current which is being drawn from the USB port. Additionally, by removing the jumper cap altogether, JP3 provides a means of preventing the board from consuming USB power.
- S2 - Switch for application use. Tied to RB2.
- S4 - MCLR reset switch
- D1 - LED for application use. Tied to RE0.
- D2 - LED for application use. Tied to RE1.

More Information

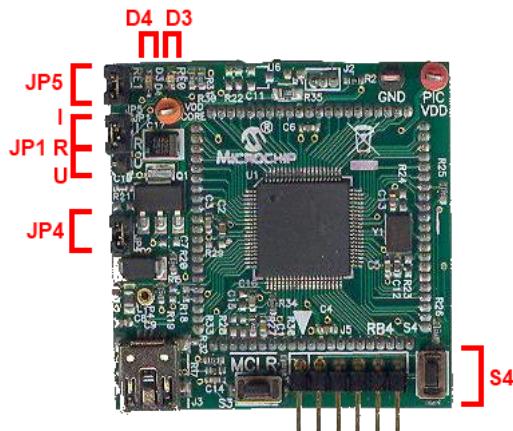
[Product website](#)

1.5.6 PIC18F87J50 Plug-In-Module (PIM) Demo Board

The PIC18F87J50 PIM services as a development platform for the PIC18F87J50 family. It includes 2 LEDs and a push button. It can also optionally be connected to the PIC18 Explorer board for access to additional features, such as potentiometer, PICtail, temperature sensor, etc.

Description

Overview



- JP1 - This is a three-pin header with the labels, "I", "R" and "U". The "R" is an abbreviation referring to microcontroller pin, RB5. "I" is an abbreviation referring to the "ICE" female header pin for the RB5 signal. "U" is an abbreviation for the USB VBUS line. When the jumper is in the "R" to "I" position, the RB5 pin connects only to the ICE female header pin, just like most of the other general purpose I/O pins. When the jumper is in the "R" to "U" position, RB5 (which is 5.5V tolerant) can be used to sense when the USB cable has been attached to the host, and when the host is actively providing power to the +5V VBUS line. According to the USB 2.0 specifications, no device should ever pull the D+ or D- lines high (such as with the D+ or D- pull-up resistor) until the host actively powers the +5V VBUS line. This is intended to prevent self-powered peripherals from ever sourcing even small amounts of power to the host when the host is not powered. Small amounts of current could potentially prevent the host (and possibly other USB peripherals connected to that host) from fully becoming depowered, which may cause problems during power-up and initialization. Self-powered peripherals should periodically monitor the +5V VBUS line and detect when it is driven high. Only when it is powered should user firmware enable the USB module and turn on the D+ (for full speed) or D- (for low speed) pull-up resistor, signaling device attach to the host. The recommended method of monitoring the +5V VBUS line is to connect it to one of the microcontroller's 5.5V tolerant I/O pins through a large value resistor (such as 100 kOhms). The resistor serves to improve the ESD ruggedness of the circuit as well as to prevent microcontroller damage if user firmware should ever unintentionally configure the I/O pin as an output. Peripherals which are purely bus powered obtain all of their power directly from the +5V VBUS line itself. For these types of devices, it is unnecessary to monitor when the VBUS is powered, as the peripheral will not be able to source current on the D+, D- or VBUS lines when the host is not powered.
- JP4 - This jumper is located in series with the +5V VBUS power supply line from the USB connector. When the jumper is removed, a current meter may be placed between the header pins to measure the board current which is being drawn from the USB port. Additionally, by removing the jumper cap altogether, JP4 provides a means of preventing the board from consuming USB power.
- JP5 - This jumper provides a means of removing the LED pin loading on the RE0 and RE1 pins.
- S4 - Switch for application use. Tied to RB4.
- D3 - LED for application use. Tied to RE0.
- D4 - LED for application use. Tied to RE1.

More Information

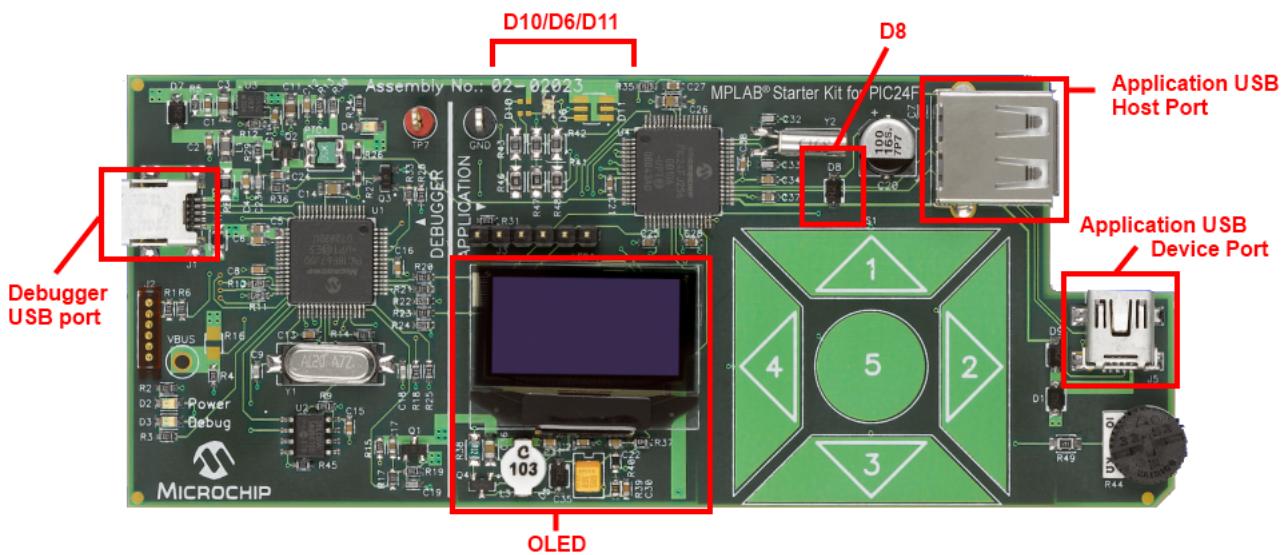
- [Product webpage](#)

1.5.7 PIC24F Starter Kit

The PIC24F Starter Kit is a feature rich board with lots of on board features for customers to work with. The board includes a RGB LED, a USB host port, a USB device port, an OLED, 5 capacitive touch buttons, a potentiometer, and an on board debugger.

Description

Overview



D8 - For dual role examples on the PIC24F starter kit, D8 needs to be removed. D8 allows the firmware to verify that the 5v has been delivered to the application USB host port. This, however, is also tied to the application USB device port. With the diode in place the controller can not determine if the 5v it sees is from the USB host port being powered or from the USB device port on an attachment to a USB host.

More Information

[Product Website](#)

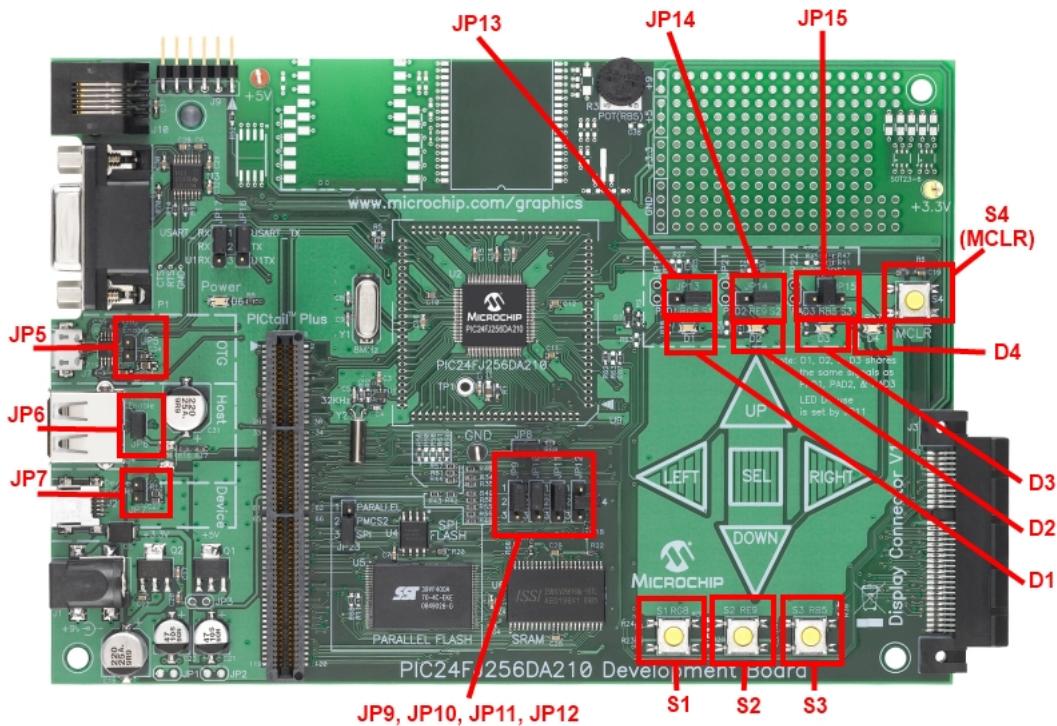
[Introduction Video](#)

1.5.8 PIC24FJ256DA210 Development Board

The PIC24FJ256DA210 development board services the PIC24FJ256DA210 family devices. It has 4 LEDs, 3 push buttons, 5 capacitive touch buttons, a USB host port, a USB device port, an USB OTG port, an RS232 port, and the ability to drive an graphics display.

Description

Overview



S1 - Application switch. Tied to RG8 when JP13 is shorted from S1 to RG8 settings.

S2 - Application switch. Tied to RE9 when JP14 is shorted from S1 to RE9 settings.

S3 - Application switch. Tied to RB5 when JP15 is shorted from S1 to RB5 settings.

S4 - MCLR reset button. Resets the microcontroller on the board.

D1 - Application LED. Connected to RG8 when JP13 is shorted from PAD1 to RG8.

D2 - Application LED. Connected to RE9 when JP14 is shorted from PAD2 to RE9.

D3 - Application LED. Connected to RB5 when JP15 is shorted from PAD3 to RB5.

D4 - Application LED. Connected to RA7 when JP11 is shorted from 1 to 2.

JP5 - Connect USB OTG port to VBUS.

JP6 - Connect USB Host port to VBUS.

JP7 - Connect USB Device port to VBUS.

JP11 - Functionality selection for RA7.

JP13 - Functionality selection for RG8.

JP14 - Functionality selection for RE9.

JP15 - Functionality selection for RB5.

More Information

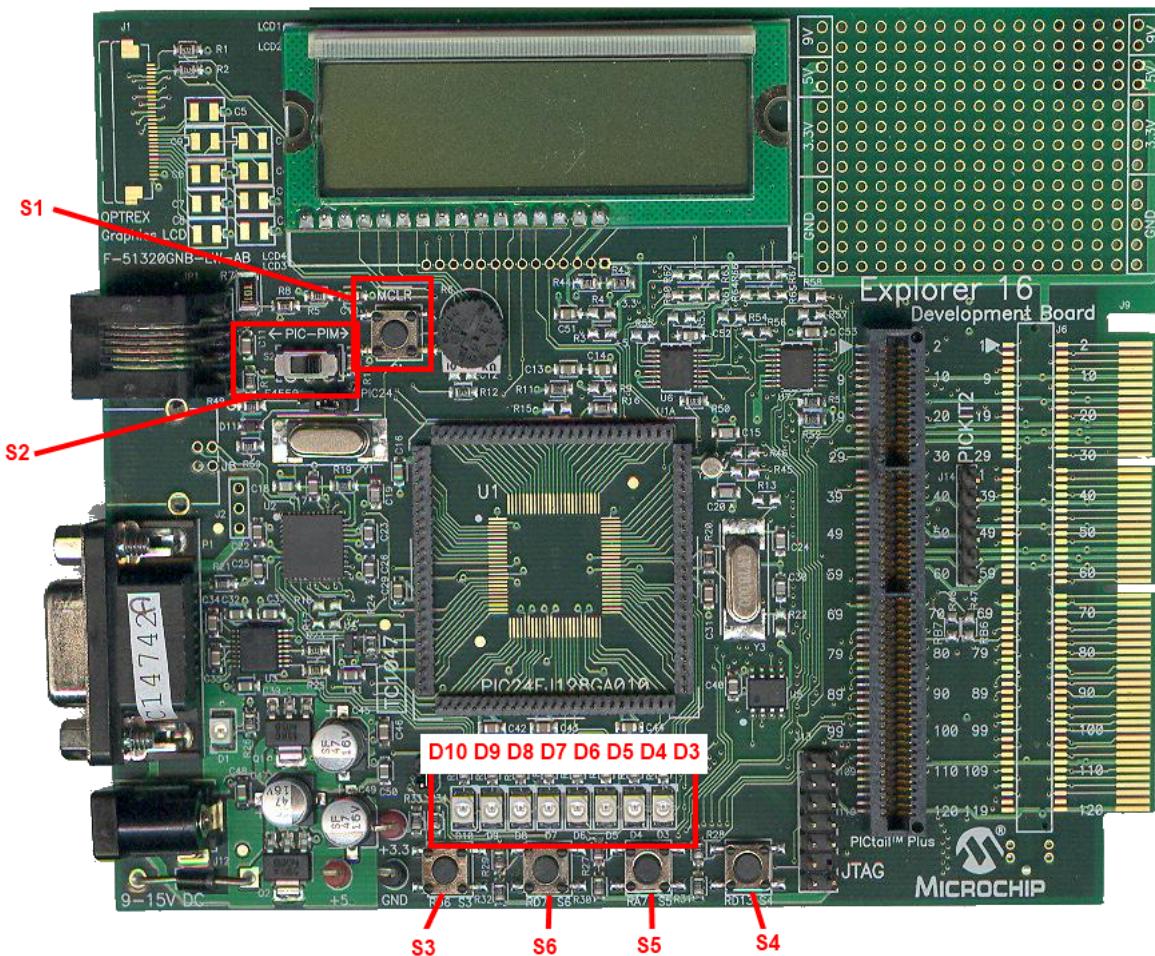
[Product Webpage](#)

1.5.9 Explorer 16

The Explorer 16 is the base development platform for the 16-bit processors. It has a processor header that allows various Processor Interface Modules (PIMs) to be attached allowing the user to utilize various processors on the board. The Explorer 16 includes an LCD screen, potentiometer, EEPROM, temperature sensor, 8 LEDs, 4 push buttons, RS232 port, and the PICtail+ expansion connectors/card-edge that allows for various add-on boards to be connected.

Description

Overview:



S1 - Reset button (MCLR)

S2 - Processor switch. This switch determines which processor is running, the processor on the board or the processor on the Plug-In-Module (PIM).

S3, S4, S5, S6 - Application switches. For information about what pin is connected to this switch, please refer to the information for the PIM in use.

D3 through D10 - Application LEDs. For information about what pin is connected to this LED, please refer to the information for the PIM in use.

More Information:

[Product webpage](#)

1.5.9.1 PIC24FJ256GB110 Plug-In-Module (PIM)

Processor module for the PIC24FJ256GB110 family for the Explorer 16.

Description

Overview

The PIC24FJ256GB110 Plug-In-Module (PIM) is not a standalone board. It requires the use of the Explorer 16 ([DM240001](#)). For USB applications the USB PICTail plus daughter board ([AC164131](#)) is also required.

More Information

[Information sheet](#)

1.5.9.2 PIC24FJ256GB210 Plug-In-Module (PIM)

Processor module for the PIC24FJ256GB210 family for the Explorer 16.

Description

Overview

The PIC24FJ256GB210 Plug-In-Module (PIM) is not a standalone board. It requires the use of the Explorer 16 ([DM240001](#)). For USB applications the USB PICTail plus daughter board ([AC164131](#)) is also required.

For USB operation, jumpers JP1, JP2, and JP3 should be shorted from pins 1 to 2.

More Information

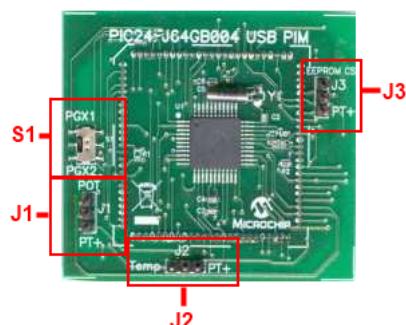
[Information sheet](#)

1.5.9.3 PIC24FJ64GB004 Plug-In-Module (PIM)

Processor module for the PIC24FJ64GB004 family for the Explorer 16.

Description

Overview



S1 - Select which programming pins are going to be used on the microcontroller. The "PGX1" setting must be used for USB operation.

J1 - A/D setting for RC1 (center tap). Setting the jumper to "POT" connects the pin to the potentiometer on the Explorer 16. Setting the jumper to "PT+" connects the pin to the PICTail+ connector on the Explorer 16.

J2 - A/D setting for RC0 (center tap). Setting the jumper to "Temp" connects the pin to the temperature sensor on the Explorer 16. Setting the jumper to "PT+" connects the pin to the PICTail+ connector on the Explorer 16.

J3 - I/O selection for RA8 (center tap). Setting the jumper to "EEPROM CS" connects the pin to the chip select line of the EEPROM on the Explorer 16. Setting the jumper to "PT+" connects the pin to the PICTail+ connector on the Explorer 16.

More Information

[Plug-In-Module \(PIM\) Information Sheet](#)

1.5.9.4 PIC24EP512GU810 Plug-In-Module (PIM)

Processor module for the PIC24EP512GU810 family for the Explorer 16.

Description

[More Information](#)

[Information Sheet](#)

1.5.9.5 dsPIC33EP512MU810 Plug-In-Module (PIM)

Processor module for the dsPIC33EP512MU810 family for the Explorer 16.

Description**More Information**

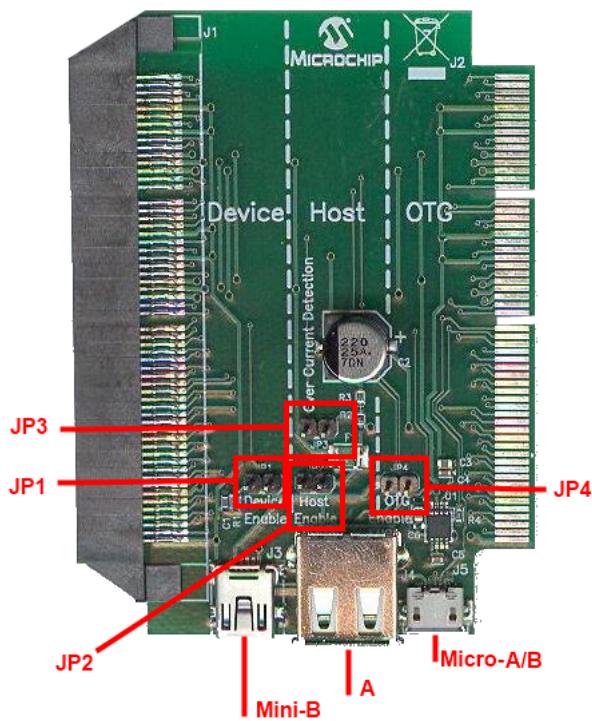
[Information Sheet](#)

1.5.9.6 USB PICTail Plus Daughter Board

The USB PICTail+ board is a add-on side card for the Explorer 16 that adds a USB device, USB host, and USB OTG port to the Explorer 16 capability.

Description

[Overview](#)



JP1 - Connects the VBUS pin of the mini-B connector to the VBUS pin of the microcontroller.

JP2 - Connects the VBUS pin of the A connector (and associated circuitry) to the VBUS pin of the microcontroller.

JP3 - Connects the VBUS voltage detection resistor divider circuit to the microcontroller (pin varies depending on the processor module).

JP4 - Connects the VBUS pin of the micro-A/B connector (and associated circuitry) to the VBUS pin of the microcontroller.

This board has 3 USB connectors on it.

- The mini-B connector is for USB device operation. For use in this mode JP1 should be shorted and JP2, JP3, and JP4 should be open.
- The A connector is for USB host support. JP2 should be short for this mode. JP3 can be shorted to enable VBUS voltage sensing. Some demos may require this feature. JP1 and JP4 should be open.
- The micro-A/B connector is for USB OTG operation. JP4 should be short and JP1, JP2, and JP3 should be open.

More Information

[Product website](#)

[Ordering information](#)

1.6 Demos

A description of how to what each demo is and how to run it.

Description

1.6.1 Device - Audio Microphone Basic Demo

This demo shows how to implement a simple USB microphone.

Description

Supported Demo Boards

The matrix of which demos are supported on a specific board can be found in the Release Notes demo board support section. Verify that the board you wish to use will work with this demo. This table also describes some of the limitations that the board might have while running this demo.

Demo Board I/O Mapping

Each demo board has a different number of push buttons, LEDs, and other features with various different names for these components. To determine which board features are used for which demo features, please refer to the io_mapping.h file in the demo folder under the system_config folder. Each demo board will have a corresponding folder with an io_mapping.h file in it. For example, for the PIC18F46J50 PIM this would be the following file:

<install_directory>/apps/usb/device/audio_microphone/firmware/src/system_config/pic18f46j50_pim/io_mapping.h

For more information about each demo board, please refer to the Demo Board Information section.

Demo Operation

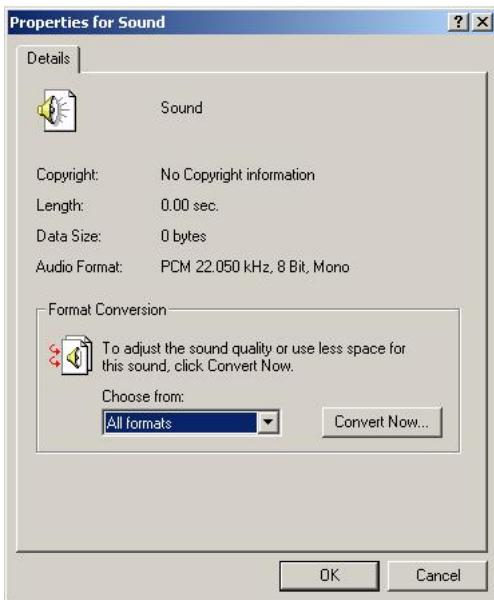
This demo uses the selected hardware platform as a USB Microphone Device. The demo emulates a PCM, 16 bits/Sample, 8000 Samples/ second, mono Microphone. Connect the device to the computer. Open a sound recording software package. Each sound recording software interface is different so the following instructions may not apply the to software package you are using. Please refer to the user's manual for the software package you are using for more details of how to configure that tool for Sound recording.

Using Sound Recorder [Windows Computers]

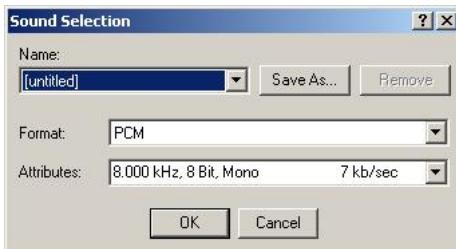
Open Sound Recorder from Start->Programs->Accessories->Entertainment->Sound Recorder. Click on File-> Properties.



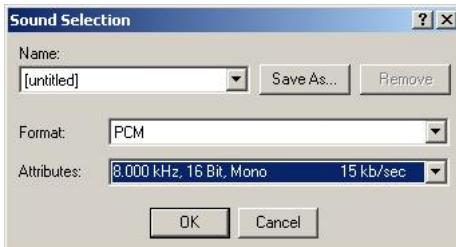
Now the 'Properties for Sound' Window gets opened as shown below. Click on 'Convert Now' button.



This opens up the 'Sound Selection' window as shown below.



Change the 'Attributes' to "8.00kHz, 16 Bit, Mono 15kb/sec" in the 'Sound Selection' Window.



Click on OK button on the 'Sound Selection' Window. Click OK button on the 'Properties for Sound' Window.

Click on the Record Button on the Sound Recorder.



At this point you can press the pushbutton on the demo board and it will record a voice that is stored in the USB device. Once you finish with the recording click on the 'Play' button to play the recorded voice which can be heard through your computer Speaker.

1.6.2 Device - Audio MIDI Demo

This demo shows how to implement a simple bi-directional USB MIDI device.

Description

Supported Demo Boards

The matrix of which demos are supported on a specific board can be found in the Release Notes demo board support section. Verify that the board you wish to use will work with this demo. This table also describes some of the limitations that the board might have while running this demo.

Demo Board I/O Mapping

Each demo board has a different number of push buttons, LEDs, and other features with various different names for these components. To determine which board features are used for which demo features, please refer to the io_mapping.h file in the demo folder under the system_config folder. Each demo board will have a corresponding folder with an io_mapping.h file in it. For example, for the PIC18F46J50 PIM this would be the following file:

```
<install_directory>/apps/usb/device/audio_midi/firmware/src/system_config/pic18f46j50_pim/io_mapping.h
```

For more information about each demo board, please refer to the Demo Board Information section.

Demo Operation

This demo uses the selected hardware platform as a USB MIDI device. Connect the device to the computer. Open a MIDI recording software package. Each MIDI recording software interface is different so the following instructions may not apply to the software package you are using. Please refer to the user's manual for the software package you are using for more details of how to configure that tool for a USB MIDI input.

In this demo each time you press the button on the board, it will cycle through a series of notes.

1.6.2.1 Garage Band '08 [Macintosh Computers]

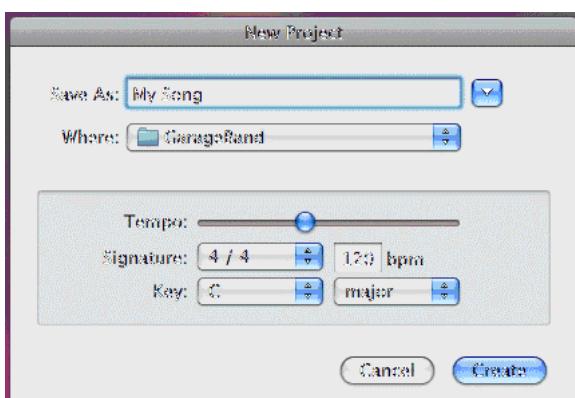
This section shows how to run the MIDI demo using GarageBand '08.

Description

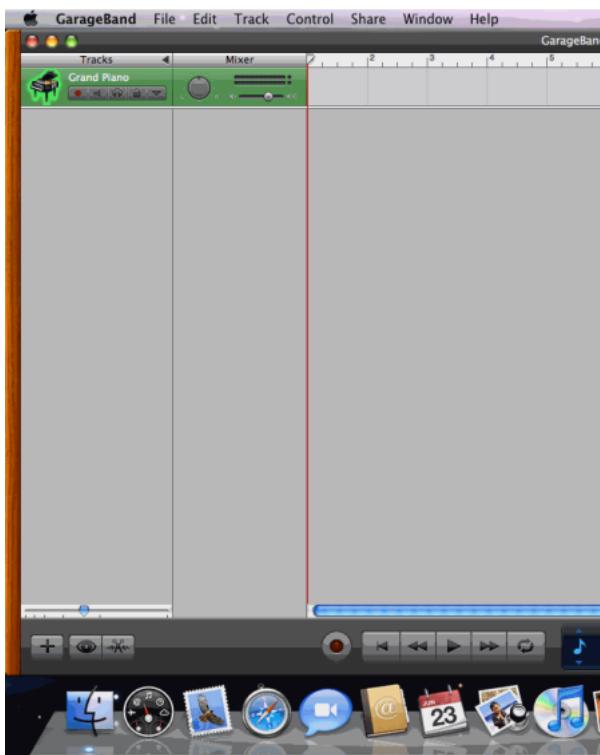
Open Garage Band. If you haven't opened Garage Band before you will see an opening window. Select "Create New Music Project"



The next window will prompt you for information about the song. Change any of the information is desired. Click "Create" when done.



The Garage Band main window will open. In this window there should be a single default track if the USB device is already attached. At this point you can press the pushbutton on the demo board and it will cycle through a series of notes and play these notes through the computer speakers.



1.6.2.2 Using Linux MultiMedia Studio (LMMS) [Linux and Windows Computers]

This section shows how to run the MIDI demo using Linux MultiMedia Studio (LMMS).

Description

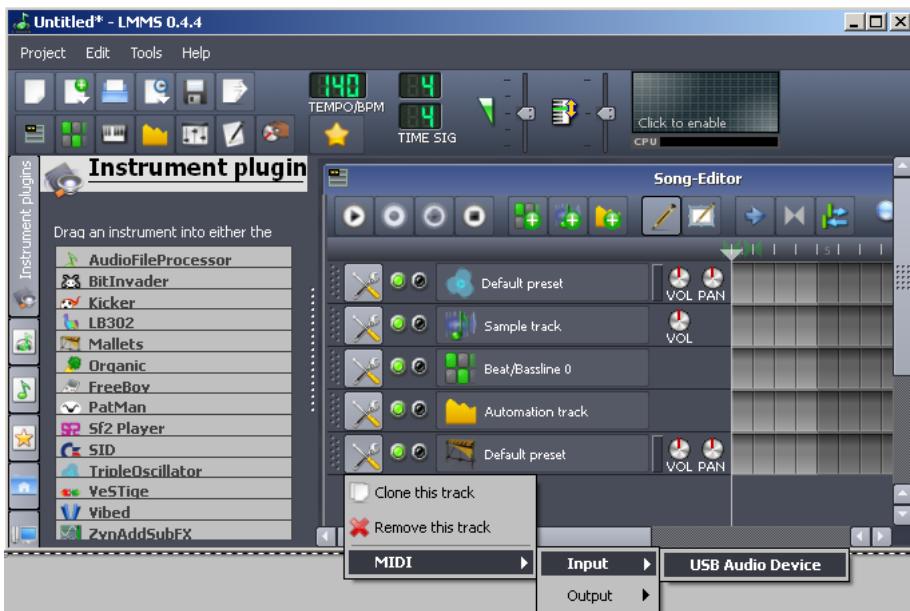
In this example we will be using Linux MultiMedia Studio (LMMS) available at <http://sourceforge.net/projects/lmms/>. Install LMMS. Attach the demo board to the computer. Make sure to attach the USB Audio MIDI example board to the computer before opening LMMS as LMMS polls for USB MIDI devices upon opening but may not find the devices attached after the program is opened.



Click on the instrument plug-in button and click and drag the desired instrument plug in to the song editor window.



Once the new instrument is available in the song editor window, “click on the actions” for this track button. Select the “MIDI > Input > USB Audio Device” option.



If you open this option again you should see a green check mark indicating that the device is selected as the input.



At this point you can press the pushbutton on the demo board and it will cycle through a series of notes and play these notes through the computer speakers.

1.6.3 Device - Boot Loader - HID

An example boot loader using the HID device class.

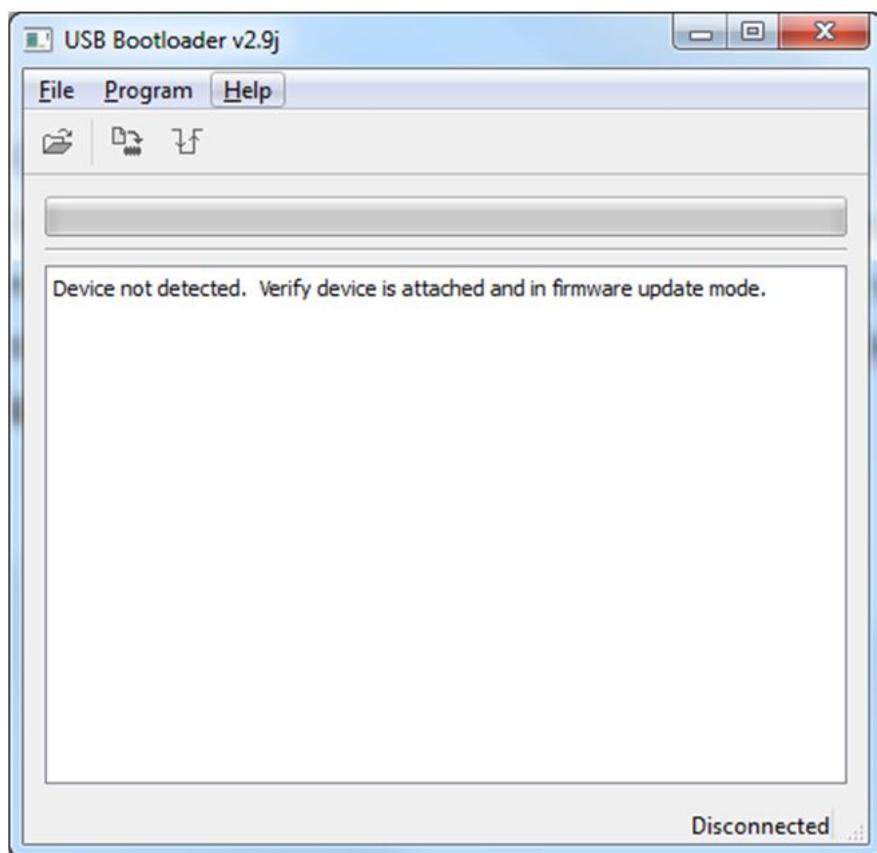
Description

Supported Demo Boards

The matrix of which demos are supported on a specific board can be found in the Release Notes demo board support section. Verify that the board you wish to use will work with this demo. This table also describes some of the limitations that the board might have while running this demo.

Demo Operation

Included with the MCHPFSUSB HID Bootloader firmware is a simple PC-side host application. This application implements a basic set of commands, allowing a user to invoke the bootloader on an appropriately-programmed PIC microcontroller and program new application code. The bootloader host application interface looks as shown in figure below:



Using the application is straightforward. After launching the host application, connect the hardware containing a PIC microcontroller with the bootloader already programmed to the host PC with a USB cable. The host responds by enumerating the hardware as a HID class device; the host application responds with the message 'Device Attached' and enabling several options.

NOTE: The host application, like the rest of the HID bootloader, is an example application; it should be thought of as a framework for development. Users may want to consider modifying the interface or its functions to suit their own purpose.

The host application supports the following functions:

- Import Firmware Image: selects a HEX image file on the PC to be loaded onto the microcontroller, using the standard Windows File Open dialog. The file is stored in the host's internal buffer.

- Erase/Program/Verify Device: This is a 3-step operation.
 - 1) It first erases the Flash program memory on the target microcontroller. If “Allow Configuration Word Programming” is selected, the configuration bits or flash configuration words (depending on the device) will also be erased.
 - 2) It then programs the target microcontroller with the recently selected HEX file.
 - 3) Finally it compares the program image on the target to a HEX file image stored in the host's buffer to complete verify operation.
- Reset Device: issues a RESET instruction to the target microcontroller. This is typically done following successful erase/program/verify operation, so that the application can begin executing the newly reprogrammed firmware image.

The “Program” file menu option has a “Settings...” which contains checkboxes allowing the user to select what type of program memory regions should be programmed on the erase/program/verify operation. This allows for deselecting EEPROM from being programmed for example. This also allows for enabling/disabling reprogramming of the configuration bits (and therefore last page of flash memory on PIC18FxxJxx devices).

NOTE: The SD Card PICtail™ Daughter Board (Microchip Direct: AC164122) uses the RB4 I/O pin for the card detect (CD) signal when used with the PIC18F87J50 FS USB Demo Board (PIM), and is actively driven by the PICtail. The active drive overpowers the pull up resistor on the RB4 pushbutton (on the PIC18F87J50 FS USB Demo Board). As a result, if the PIC18F87J50 is programmed with the HID bootloader, and an SD Card is installed in the socket when the microcontroller comes out of reset, the firmware will immediately enter the bootloader (irrespective of the RB4 pushbutton state). To exit the bootloader firmware, remove the SD Card from the SD Card socket, and tap the MCLR button. When the SD Card is not plugged in, the PICtail will drive the card detect signal (which is connected to RB4) logic high, which will enable the bootloader to exit to the main application after coming out of reset. Once the main application firmware is operating, the SD Card can be plugged in. The SD Card is “hot-swappable” and should be recognized by the host upon insertion. To avoid this inconvenience when using the bootloader with the PICtail, it is suggested to modify the bootloader firmware to use some other I/O pin for bootloader entry, such as RB0 (which has a pushbutton on it on the HPC Explorer board).

1.6.3.1 Customizing for an Application

This section covers various topics that should be considered when implementing a boot loader. It also discusses possible changes in behavior from the default behavior that a user might want to implement in their boot loading solution.

Description

1.6.3.1.1 Importance of Change the VID/PID

Discusses why it is critical to change the VID/PID in the example code to one unique to the product being developed.

Description

The USB Vendor ID (VID) and Product ID (PID) are special numbers (16-bits each) that all USB devices implement. These numbers are especially important for bootloader operation, since they are the primary identifying numbers that a PC application (such as the HID bootloader host/PC GUI program) needs in order to “find” and connect up to the correct USB device on the bus (a USB root port can have up to ~127 USB devices attached to it, and some means must be used to “find” specific products).

When implementing a USB device with a bootloader, it is especially important to change the VID/PID value from the original/default value as distributed in the example firmware/PC application, to new values that are unique for your application product line. This is important, to ensure that no unintended host/PC application software can ever unintentionally connect up to your USB device, and reprogram it with an incorrect/incompatible USB firmware image, designed for some other manufacturer’s product.

The VID/PID values can be modified in the bootloader firmware project by editing the USB device descriptor in the `usb_descriptors.c` file. Once the value has been modified, the host/PC GUI bootloader program that performs the firmware

updating must also be modified, so as to search for and connect up to the proper USB device with the new VID/PID. In the “\usb\device\bootloaders\utilities\qt5_src\Bootloader” host/PC GUI source code, the VID/PID values can be changed by editing the #define VID and #define PID constants (which currently resides in the Comm.h file).

1.6.3.1.2 Safe Boot Loading Considerations

This section discusses some items to consider to avoid common pitfalls when implementing a boot loading solution.

Description

When an application implements self reprogramming capability, it is strongly recommended to also simultaneously implement provisions to ensure the microcontroller does not execute at voltages that are too low for the configured frequency (ex: don't violate the voltage versus frequency graph in the datasheet). Overclocking the microcontroller (ex: by running a full frequency, but at a voltage below the required minimum from the device datasheet) can result in possible instruction op-code mis-fetch or mis-execution. This can result in unexpected code flows, allowing normally unreachable code to get reached. This can potentially result in unintended activation of bootloader/flash memory self programming code, possibly causing the erasure or corruption of important program memory. This potential problem is best avoided by implementing provisions in both the bootloader firmware and the application firmware project, to either outright prevent all code execution during the low/inadequate voltage condition (ex: by enabling and using BOR, and/or putting the microcontroller to sleep mode), or by clock switching to a low enough frequency at runtime, so as to always meet the datasheet voltage versus frequency requirements.

Additionally, special consideration is needed if enabling the watchdog timer (WDT) feature of the microcontroller. The WDT can be used in applications with a bootloader, but the timeout period must always be configured to be longer than the worst case flash page erase and block programming duration. Failure to do so may result in unexpected timeout/reset occurring during the erase/program sequence, leading to unintended NVM contents.

1.6.3.1.3 Configuration Bits

Configuration bits and their impact on boot loading applications.

Description

Make certain that all configuration bit settings between the bootloader firmware project, and the application firmware project, match 100% exactly. If they do not match, modify one or both projects until they do. The microcontroller hardware only implements one set of configuration bits, and therefore, the configuration bit settings are always shared between the bootloader firmware and application firmware projects.

Attempting to declare two sets of configuration bits (that are not 100% exactly the same) can prevent the application and bootloader firmware image .hex files from being successfully merged when using the loadable project feature in MPLAB X (see the Merging Bootloader and Application Project Output section).

By default, the HID bootloader does not reprogram the microcontroller config bits during an erase/program/verify sequence. Reprogramming the configuration bits is generally not recommended, since doing so is generally considered much more “dangerous” to the application, than reprogramming the normal application firmware code. When reprogramming the configuration bits, it is very easy to leave the application in a permanently broken (“bricked”) condition, if any of the new configuration bit settings are not 100% compatible with the hardware of the application. Certain config bit settings, such as the oscillator, BOR, extended instruction set, WDT, etc., are especially hazardous, since changing them can easily leave both the bootloader firmware and application firmware images in a non-operable (or non-USB operable) state, thereby preventing further re-programming operations.

However, if absolutely necessary, the HID bootloader firmware and PC GUI applications do support reprogramming of the configuration bits. Doing so requires a special PC GUI/bootloader firmware “unlock” sequence to be executed. This occurs when the host/PC GUI program sends the “UNLOCK_CONFIG” bootloader command to the firmware. For applications that will support config bit reprogramming, it is recommended to hide the option from the PC GUI program, so that it is not accessible to end consumers, except in special circumstances when truly necessary.

For some USB microcontrollers (namely PIC18FxxJxx USB microcontrollers), the configuration bits are stored in normal program flash memory, at the end of the application firmware program space. On these microcontrollers, the configuration

bits are therefore stored on the same flash memory erase page as other (non-critical) program memory space that could theoretically be used by the application program. On these devices, it is generally recommended to modify the linker settings for the application program image, so that the application project is built so that it does not use any of the program memory that is shared with the flash memory erase page containing the configuration bits (ex: the last implemented flash memory page). This ensures that the application firmware image can be fully re-programmed, without requiring the configuration bit reprogramming mode to be enabled (by the UNLOCK_CONFIG command).

NOTE: The HID bootloader PC application and firmware will not reprogram the last page of flash memory on these PIC18FxxJxx devices (which is shared with the config bits), unless the user configures the bootloader PC GUI program to perform config bit reprogramming operations, with the UNLOCK_CONFIG command. This restriction only applies to the specific microcontrollers that implement the configuration bits in the program flash memory space.

1.6.3.1.4 Application Version Information

This section describes how to use the application version feature of the boot loader example.

Description

The application firmware version word resides in the application space, but it is used by the bootloader to read out the application firmware version number. This makes it potentially possible to make a PC GUI that can check the currently programmed application firmware image version and determine if the user is trying to program an older .hex file than what was already programmed into the device. If a custom PC GUI application is used to perform the firmware updates, the GUI application software can then use this information to warn the user about programming an older image than the existing image, block the user from performing the operation, etc.

The Application Firmware Version Word can be read by the bootloader host/PC GUI software using the Extended Query Command Response, described below.

In order to fully implement and use the application firmware version word feature, the application firmware image must place a constant in the flash memory, at the magic version word address. For PIC18 targets, this would typically be done with code in the application firmware project, such as:

Version Word with XC8

```
const unsigned int VersionWord @ 0x1016 = 0x0100;      //Initialize to the appropriate
version value
```

Version Word with C18

```
#pragma code VersionWordSection=0x1016
rom unsigned int VersionWord = 0x0100;      //Initialize to the appropriate version value
#pragma code
```

1.6.3.1.5 Host Application Responsibilities

Discusses some of the responsibilities of the host application. These tasks must be performed in order to insure proper boot loading operation.

Description

In order to minimize the bootloader firmware program memory size and complexity, the bootloader is architected to perform the most complex tasks in the host host/PC GUI software. Things such as .hex file parsing, and issuing commands, are therefore the responsibility of the host/PC software. Additionally, the firmware is currently written in such a way as to impose the following restrictions on the host/PC GUI software:

1. When the PROGRAM_DEVICE command is used, the total program data payload sent prior to the next PROGRAM_COMPLETE command is sent, must be an exact integer number of program instructions. In other words for PIC16 and PIC18 devices, the total program memory programmed must be an even number (since program instructions are 2 bytes wide on these architectures). Sending an odd number of total data payload bytes is not supported, and therefore should be padded with 0xFF (the blank/default value of the flash memory after an erase operation) if necessary to achieve a total PROGRAM_DEVICE payload quantity (spanning multiple packets) that is even.
2. When multiple PROGRAM_DEVICE packets are sent to the device, the addresses sent must always be contiguous (ex: second packet address must be equal to first packet's address + the payload size), growing from lowest address to

highest address. Non-contiguous address jumping is only allowed, if the host/PC program sends the PROGRAM_COMPLETE command in between the non-continuous address regions.

3. After sending one or more PROGRAM_DEVICE packets, the host/PC software must send the PROGRAM_COMPLETE command once the end of the region is reached (of if it wants to abort operation of the entire region). The microcontroller firmware is allowed to buffer up received bytes intended for programming, without necessarily committing all of them to the non-volatile memory, until the PROGRAM_COMPLETE command is issued by the host/PC software.

1.6.3.2 Implementation Details

This section discusses the lower level details of the boot loader and how it was implemented.

Description

1.6.3.2.1 Command Set

Details the commands implemented in the HID boot loader example.

Description

The host application GUI program communicates with the USB HID bootloader firmware using a set of 9 commands. The host application is the “master” of the bootloading operation, and is responsible for issuing commands to the bootloader firmware that is responsible for fulfilling the requests.

All commands that the host application sends to the microcontroller firmware are fixed 64-byte USB packets that are sent over the HID interrupt OUT endpoint to the device. Some commands that the host software sends to the microcontroller firmware require that the firmware responds with a fixed 64-byte response packet on the HID interrupt IN endpoint, while other commands require no response.

The first byte of the packet is always the command for the current packet. The remaining 63 bytes are command-specific information, where required. The commands are listed and summarized below.

Command Byte (Hex)	Command	Device Response Packet Expected
02	QUERY_DEVICE	Yes
03	UNLOCK_CONFIG	No
04	ERASE_DEVICE	No
05	PROGRAM_DEVICE	No
06	PROGRAM_COMPLETE	No
07	GET_DATA	Yes
08	RESET_DEVICE	No
09	SIGN_FLASH	No
0C	QUERY_EXTENDED_INFO	Yes

1.6.3.2.1.1 QUERY_DEVICE

The QUERY_DEVICE command (0x02) is a request from the host to determine the valid memory ranges that are allowed to be programmed, among other things about the microcontroller.

Description

The QUERY_DEVICE command (0x02) is a request from the host to determine the valid memory ranges that are allowed to be programmed, among other things about the microcontroller. This information can be obtained by the PC GUI application by sending the QUERY_DEVICE command, and then reading back the response packet which will describe the device’s programmable regions (ex: application firmware, EEPROM, and User ID programmable region addresses/size). The

command when sent to the device firmware has no data payload.

Table: QUERY_DEVICE command format (when sent from the host to the device).

Packet Byte	Content
0	QUERY_DEVICE (0x02)
1-63	(padding – init to 0x00)

After the QUERY_DEVICE command packet has been sent to the device firmware, the device responds with an USB IN (that is, IN to the host) packet, with the format:

Table: QUERY_DEVICE response format (sent from USB device firmware to host PC GUI application)

Packet Byte	Content
0	QUERY_DEVICE (0x02)
1	bytesPerPacket
2	deviceFamily
3	Type1
4-7	Address1
8-11	Size1
12	Type2
13-16	Address2
17-20	Size2
...	...
48	Type6
49-52	Address6
53-56	Size6
57	versionFlag
58-63	(padding)

The bytesPerPacket indicates how many data bytes are sent in the write or read command data payload section.

The deviceFamily parameter indicates which device family (ex: PIC18, PIC24, PIC32, etc.) that is the current target. Type1, Type2 etc. indicate the type of memory (ex: flash, EEPROM, User ID, config bits) that is targeted by this memory range description.

Address1, Address2 etc. is the 32-bit starting address of the data range (in little endian format). Size1, Size2 etc. is the size of memory range in bytes (in 32-bit unsigned integer, little endian format).

The versionFlag is an indicator that is used to tell the PC GUI program that the version is v1.01 or later, and that it supports the QUERY_EXTENDED_INFO command (v1.00 based bootloader firmware did not implement this particular command).

For each memory range a new range description is added. A total of 6 memory ranges will fit into the 64-byte packet. After the entry of the last valid data field, the Type parameter for the next entry should be 0xFF (End of List).

The padding bytes should be treated as reserved and should be initialized to 0x00.

Table: Parameters for QUERY_DEVICE response deviceFamily field

Value	Family
0x01	PIC18
0x02	PIC24F
0x03	PIC32

0x04	PIC16
------	-------

Table: Parameters for QUERY_DEVICE response type field

Value	Family
0x01	Program Memory
0x02	Data EEPROM
0x03	Config Words
0x04	User ID
0xFF	End of list

1.6.3.2.1.2 UNLOCK_CONFIG

The UNLOCK_CONFIG (0x03) command is used to unlock protected sections of the program memory.

Description

The UNLOCK_CONFIG (0x03) command is used to unlock protected sections of the program memory (ex: configuration bit reprogramming, and on PIC18FxxJxx devices, reprogramming of the last page of program flash memory). The Lock/Unlock field allows the bootloader to either lock or unlock the configuration and other sensitive regions. A value of 0x00 unlocks the configuration range, and a value of 0x01 locks it.

This command will cause the QUERY_DEVICE results of the device to change. To re-discover the valid memory ranges, issue a second QUERY_DEVICE command.

This command does not directly have an associated response (although the host application is responsible for sending another QUERY_DEVICE request, which will have a response).

Table: UNLOCK_CONFIG command format

Packet Byte	Content
0	UNLOCK_CONFIG (0x03)
1	Lock (0x01) / Unlock (0x00)
2-63	(padding – init to 0x00)

1.6.3.2.1.3 ERASE_DEVICE

The ERASE_DEVICE (0x04) command erases all of the reprogrammable memory regions indicated by the response to the QUERY_DEVICE command.

Description

The ERASE_DEVICE (0x04) command erases all of the reprogrammable memory regions indicated by the response to the QUERY_DEVICE command. If it is necessary to erase the protected memory regions, issue the UNLOCK_CONFIG command before the ERASE_DEVICE command.

The command does not have any data payload or associated response. Typically, the host application would issue a QUERY_DEVICE following the ERASE_DEVICE command, as a means to “poll” for when the erasing process inside the microcontroller has completed (since the firmware doesn’t respond to the QUERY_DEVICE command until the internal erase operation completes).

Table: ERASE_DEVICE command format

Packet Byte	Content
0	ERASE_DEVICE (0x04)
1-63	(padding)

1.6.3.2.1.4 PROGRAM_DEVICE

The PROGRAM_DEVICE (0x05) command sends the data that is going to be written to the device.

Description

The PROGRAM_DEVICE (0x05) command sends the data that is going to be written to the device. Since this command has a maximum data payload of 58 bytes, it is necessary to issue multiple commands to program a single area of memory.

The data payload section of the packet is packed from the end, with the padding following the Size field. Ex: If the host sends 4 bytes of data to be programmed, the resulting packet would have 54 bytes of padding followed by four bytes data.

This command does not have any associated response.

Table: PROGRAM_DEVICE command format

Packet Byte	Content
0	PROGRAM_DEVICE (0x05)
1-4	32-bit address to program (little endian)
5	Size (1-58 bytes)
6-(n)	Padding (0-57 bytes)
(n+1)	Data Payload

1.6.3.2.1.5 PROGRAM_COMPLETE

The PROGRAM_COMPLETE command (0x06) is used to indicate to the device that the host program is finished sending contiguous address PROGRAM_DEVICE commands.

Description

The PROGRAM_COMPLETE command (0x06) is used to indicate to the device that the host program is finished sending contiguous address PROGRAM_DEVICE commands. This is required in case the device has any remaining bytes buffered that it needs to commit to NVM, before a new memory address range can begin to be address/programmed. Always issue this command after the last PROGRAM_DEVICE command is sent for any given memory region, or if the PROGRAM_DEVICE address will contain a non-contiguous jump (ex: because the .hex file contains a blank region in between implemented code sections, as one example).

The command does not have any data payload or associated response.

Table: PROGRAM_COMPLETE command format

Packet Byte	Content
0	PROGRAM_COMPLETE(0x06)
1-63	(padding)

1.6.3.2.1.6 GET_DATA

The GET_DATA (0x07) command reads the requested data from the device.

Description

The GET_DATA (0x07) command reads the requested data from the device. This command is normally used to allow the host/PC GUI software to perform programming verification operations, by reading back a program region that has just recently been reprogrammed (and check it against the original input .hex file contents for that location). The Size field is the number of bytes to be read from the device.

Table: GET_DATA command format (sent from host to device)

Packet Byte	Content
0	GET_DATA(0x07)
1-4	Address
5	Size (1-58 bytes)
6-63	(padding)

The response to the GET_DATA command is almost identical to the PROGRAM_DEVICE command format. The address and size fields of the response packet must match the command packet requesting the data.

Table: GET_DATA response format (sent from device to the host)

Packet byte	Content
0	GET_DATA(0x05)
1-4	Address
5	Size (1-58 bytes)
6-(n)	Padding (0-57 bytes)
(n+1)	Data Payload

1.6.3.2.1.7 RESET_DEVICE

The RESET_DEVICE (0x08) command reads the requested data from the device.

Description

The RESET_DEVICE command (0x08) causes the device to issue a software reset. The command does not have any data payload or associated response. This command is typically used to effectively switch from firmware update mode back into application run mode.

Table: RESET_DEVICE command format (sent from host to device)

Packet Byte	Content
0	RESET_DEVICE(0x08)
1-63	(padding)

1.6.3.2.1.8 SIGN_FLASH

The SIGN_FLASH command (0x09) causes a special flash signature word to be programmed at a fixed address in the flash memory.

Description

The SIGN_FLASH command (0x09) causes a special flash signature word to be programmed at a fixed address in the flash memory. The command does not have any data payload.

Table: SIGN_FLASH command format (sent from host to device)

Packet Byte	Content
0	SIGN_FLASH(0x09)
1-63	(padding)

This command should only be issued once, after a fully completed (and successful) erase/program/verify operation on all memory regions intended to be reprogrammed. The command should be followed by a QUERY_DEVICE command so as to 'poll' for the completion of SIGN_FLASH command request.

1.6.3.2.1.9 QUERY_EXTENDED_INFO

The QUERY_EXTENDED_INFO command (0x0C) is used by the host PC application to get additional info about the device, beyond the basic NVM layout provided by the QUERY_DEVICE command.

Description

The QUERY_EXTENDED_INFO command (0x0C) is used by the host PC application to get additional info about the device, beyond the basic NVM layout provided by the QUERY_DEVICE command.

Table: QUERY_EXTENDED_INFO command format

Packet Byte	Content
0	QUERY_EXTENDED_INFO
1-63	(padding)

The response packet to the QUERY_EXTENDED_INFO command is microcontroller architecture specific (ex: different format for PIC18 versus PIC24 targets). See the the processor specific sections for the responses for each specific system.

1.6.3.2.2 Boot Loader Entry

This section discusses the methods of boot loader entry that are implemented in the example projects.

Description

When an application implements self reprogramming capability with a bootloader, some method needs to be implemented so as to be able to enter into the bootloader “firmware update mode” (as opposed to the standard application run mode, where self reprogramming is normally not performed).

There are lots of ways to end up in boot loader mode. The examples provided implement two possible methods for entry into boot loader mode:

1. An I/O pin check at power-up/after any reset (ex: a user pressing a pushbutton attached to a general purpose input pin).
2. Software entry from the application run mode, into the bootloader firmware update mode, via an absolute jump to address 0x001C. This can be accomplished by receiving some application specific stimulus (ex: a custom command from a PC GUI program intended to be used in conjunction with the USB application), and then in the firmware, executing an appropriate goto instruction.

1.6.3.2.2.1 Input Button/Hardware Entry

Discusses hardware entry into boot loader mode.

Description

The HID bootloader firmware occupies the hardware reset vector of the microcontroller (ex: the reset vector is 0x0000, which is part of the HID bootloader program memory space). During boot up of the microcontroller, firmware execution will therefore begin from within the bootloader firmware project.

Early in the boot up sequence, the HID bootloader firmware can optionally perform a general purpose I/O pin check, which would typically be connected to an external push button and pull up (or pull down) resistor. If the I/O pin is in the “active” state (ex: user is actively pressing a button), as defined by the application, the HID bootloader firmware can decide to stay in the “firmware update mode” and can therefore hold off normal execution of the application run mode image. If however the I/O pin is in the inactive state (ex: button not pressed), then the boot up routine would normally decide to execute in application run mode, and would perform a jump to the remapped application image reset vector (see Memory Map Overview section).

The I/O pin check method is the most rugged method of entering the firmware update mode, since it does not require the application firmware image to be intact (at all) to get into the bootloader mode. However, software entry is also possible and may be more convenient in applications that do not have user exposed pushbutton(s) available.

The I/O pin check entry method into the firmware update mode is optional, and can be disabled if entry into the bootloader will only be performed by using the software entry method.

1.6.3.2.2.2 Software/Application Entry

Discusses software entry (entry from application) into boot loader mode.

Description

Optionally, the HID bootloader firmware may be configured to allow entry into the bootloader firmware update mode via software entry at application runtime. This would typically be performed by:

1. Allowing the microcontroller to boot up into normal application run mode (as opposed to firmware update mode).
2. While in application run mode, some application specific event occurs, whereby the application decides it wants to switch into firmware update mode.
3. The firmware executes an absolute “goto” instruction to jump to the software entry point into the HID bootloader (0x001C for PIC16 and PIC18 based projects).
4. The bootloader begins executing and keeps the processor in application firmware update mode, until the user has finished programming a new application run mode firmware image.

The software entry method stimulus event (step #2 above) is application specific, and needs to be implemented in the application firmware image project (ex: the code associated with it does not reside in the bootloader firmware program space). Typically, this would be implemented by having some custom host/PC application GUI program that sends a custom command to the application firmware image (in normal “application run mode” over the normal application USB endpoints), letting it know it should switch into the HID bootloader firmware update mode.

The software entry method is generally not as “rugged” as the I/O pin check entry method however. The software entry method necessarily requires that the application firmware image be intact and functional, prior to receiving the stimulus needed to switch into the firmware update mode. Therefore, programming a non-functional application firmware image into the microcontroller, could prevent further entry back into the firmware update mode (needed for re-programming another application firmware image). This creates a so called “chicken and egg” scenario, where recovery is not possible (without reprogramming the microcontroller with a conventional ICSP™ based programming tool, such as the PICkit 3 or MPLAB ICD 3 programmer/debugger devices).

However, some recoverability in the event of failed reprogramming sequences, such as the case of sudden power loss or the user unplugging the USB cable during the erase/program/verify sequence, is possible. This is achieved through the “Flash Signature Process”.

The software based entry method into the bootloader firmware update mode is optional, and is not mutually exclusive with the I/O pin check method. Applications may choose to use either or both entry methods as desired. The software entry method is always available (even if not used). The I/O pin entry method can be enabled or disabled based on the `usb_config.h` settings in the HID bootloader firmware project.

If the software entry method will be used, the entry into the firmware update mode can be accomplished by executing the appropriate software entry routine.

Software Entry for PIC16 Devices

```
#asm
    movlp 0x00      //most significant bits of dest, needed for PIC16 devices
    goto 0x001C
#endasm
```

Software Entry for PIC18 Devices

```
#asm
    goto 0x001C
#endasm
```

The above assumes the XC8 compiler is used. If the MPLAB C18 compiler is used instead, the `#asm` and `#endasm` should be changed to `_asm` and `_endasm` respectively.

NOTE: When the "application" image is executing, the application may jump into firmware update mode by executing the appropriate goto instruction. Before doing so however, the firmware should configure the current clock settings to be compatible with USB module operation, if they are not already. Once the goto 0x001C has been executed, the USB device will detach from the USB bus (if it was previously attached), and will re-enumerate as a HID class device with a new VID/PID (adjustable via `usb_descriptors.c` settings in bootloader firmware), which can communicate with the associated USB host software that loads and programs the new application .hex file.

1.6.3.2.3 Processor Specific Implementation Details

A boot loader is tied much closer to hardware specific features (core, tools, etc.) than most applications. As such, many features of the boot loader are part specific. This section covers sections that are part specific, and how to make part specific customizations.

Description

A boot loader is tied much closer to hardware specific features (core, tools, etc.) than most applications. As such, many features of the boot loader are part specific. This section covers sections that are part specific, and how to make part specific customizations.

1.6.3.2.3.1 PIC16 and PIC18

This section covers the PIC18 and PIC16 product line USB boot loaders.

Description

This section covers the PIC18 and PIC16 product line USB boot loaders.

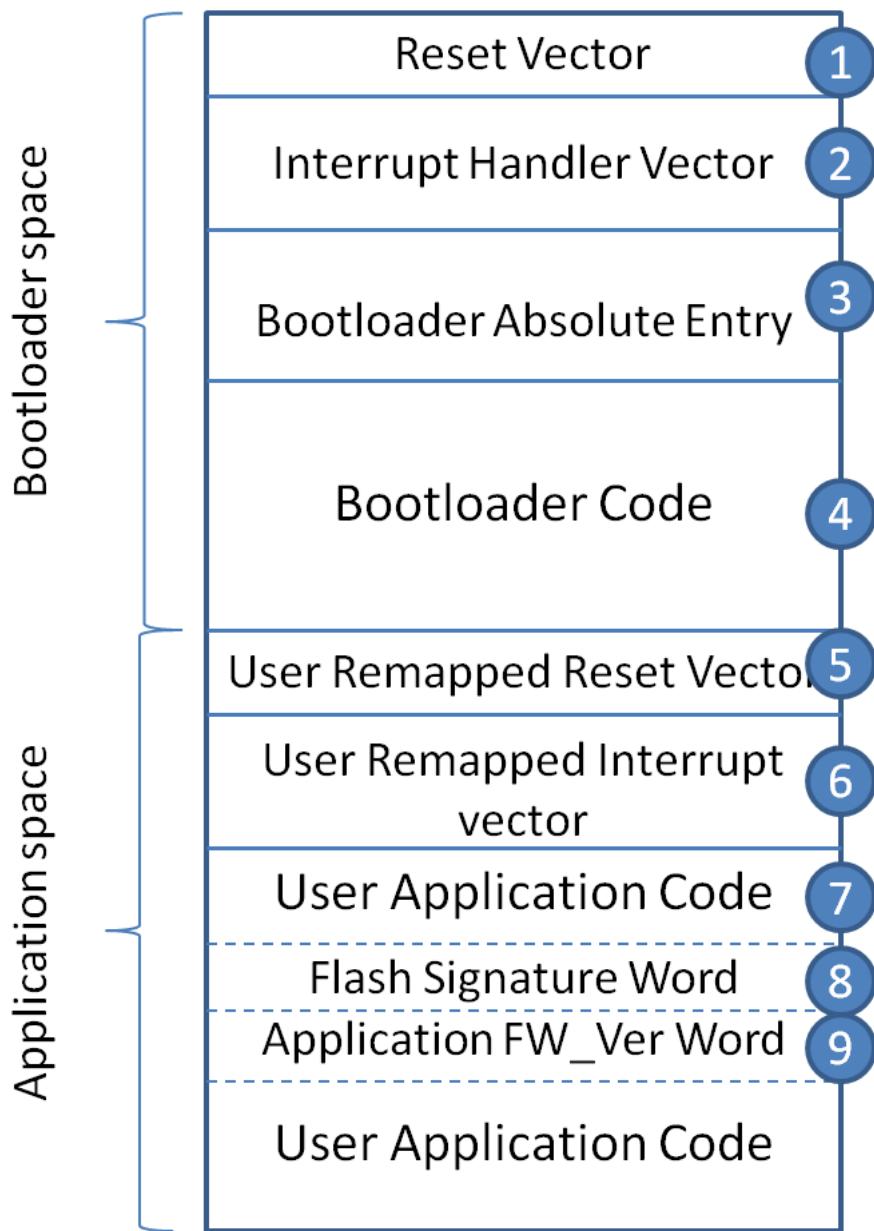
1.6.3.2.3.1.1 Memory Map

Discussion of the PIC16/PIC18 memory map and how it is utilized in the USB boot loader.

Description

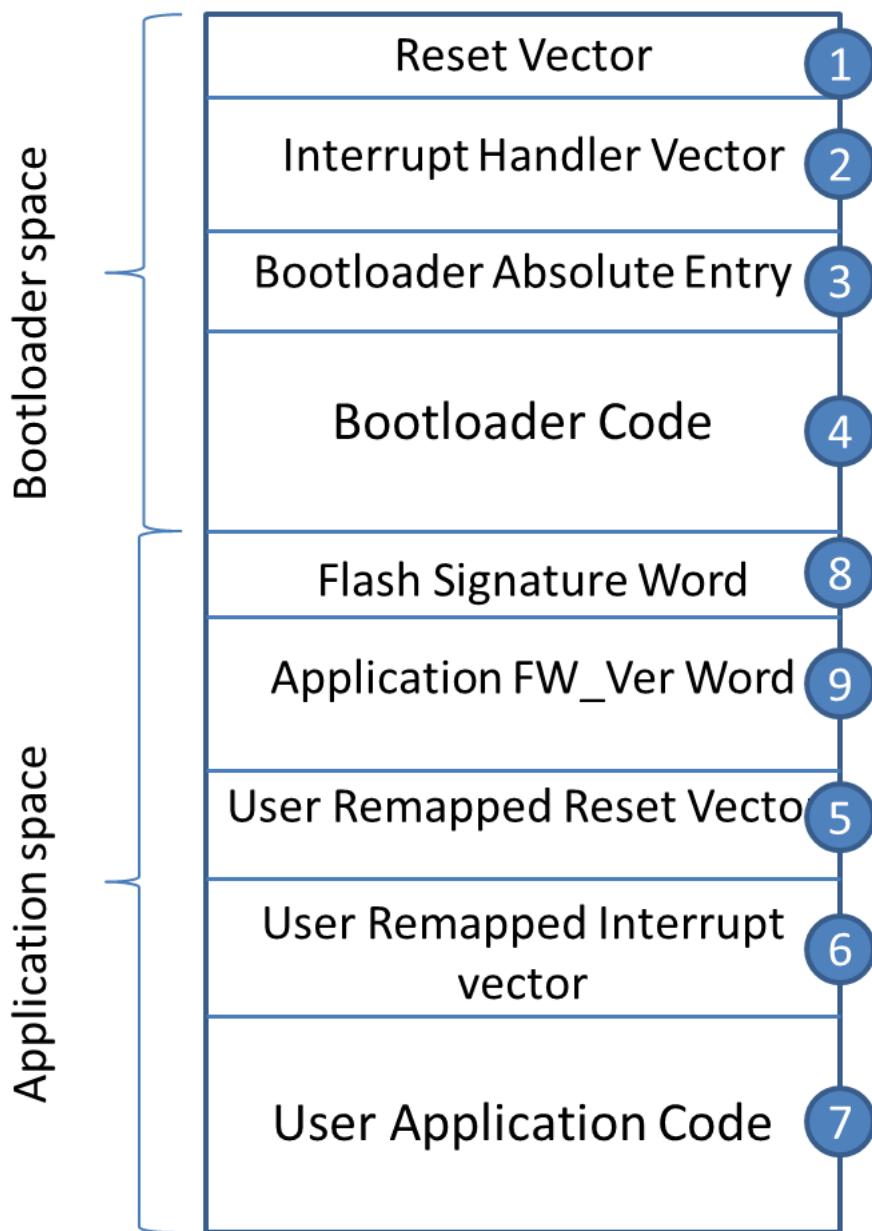
As configured by default, PIC16 and PIC18 USB HID bootloader uses the flash memory mapping as shown the figure below:

PIC18 HID Bootloader Memory Map



PIC16 HID Bootloader Memory Map

PIC16F HID Bootloader memory map



The figure below shows the table with currently implemented memory addresses for PIC16 and PIC18 USB bootloaders.

Actual flash memory address implementation values

	Flash Memory Address Values						
	Reset	Interrupt Vector	Bootloader Absolute Entry	Bootloader End	Application Reset	Application Remapped Vector	
PIC16F145x Devices	0x0000	0x0004	0x001C	0x8FF	0x904	0x908	
PIC18F USB Devices	0x0000	0x0008 (Hi Pri. Int) 0x0018 (Lo Pri. Int)	0x001C	0xFFFF	0x1000	0x1008 (Hi Pri. ISR) 0x1018 (Lo Pri. ISR)	

Let us examine each section in the memory mapping closely:

1. Reset Vector – The reset vector is defined by hardware. This is located at address 0x0000. Any reset to the CPU will go to the reset vector. The main job of the reset vector is to jump to the code that needs to be run. In the case of a bootloader, this means jumping to the bootloader code (4)).
2. Interrupt Handler Vector – This is another section that is defined by the hardware. PIC18F devices have two interrupt vectors, high priority interrupt vector and low priority interrupt vector, and they are located at 0x0008 and 0x0018 respectively. PIC16F devices have one interrupt vector located at 0x0004.
3. Bootloader Absolute Entry – This is the absolute entry point into the bootloader firmware, so application projects can jump into bootloader mode at runtime. The entry address that is currently set in the PIC18 and PIC16 bootloader is 0x001C.
4. Bootloader code – This is where the bootloader code resides. This section handles all of the loading of the new application code.
5. User Remapped Reset Vector – This section is defined by the bootloader. This is the remapped reset vector address that indicates start of the main application code.
6. User Remapped Interrupt Vector – Since the hardware interrupt vector is located in the bootloader space, the bootloader must remap all of the interrupts to the application space. This is done using user remapped interrupt vector. In PIC16 and PIC18 bootloader implementations, user remapped interrupt vector(s) are defined in the bootloader code.
7. Application code – This section is where the main application code is located.
8. Flash Signature Word – This is the address in program memory in Application Space where Flash Signature Word is located. This is a special program memory word that gets programmed (only after the entire erase/program/verify process is completed successfully) with a known value, and indicates to the bootloader code that the application firmware image is fully intact. It is necessary for the flash signature word to be located on the very first erase page during the erase sequence, and must also be the very last portion of program memory that gets re-programmed. In the current implementation, the Flash Signature Word is located within the application program memory space, at a specific fixed address. Additional details on Flash Signature process are described in the Flash Signature section.
9. Application Firmware Version Word – This word also resides in the application space, but it is used by the bootloader firmware to read out the application firmware version number. In the current implementation, this word is located at fixed address in the application space. Additional details on Application Firmware Version Word are described below.

1.6.3.2.3.1.2 QUERY_EXTENDED_INFO Response

This section discusses the results returned by the QUERY_EXTENDED_INFO command for the PIC16/PIC18 devices.

Description

QUERY_EXTENDED_INFO is a command that may be sent from the PC GUI application controlling the bootloading process, to the bootloader firmware. This command is only supported in bootloader firmware version 1.01 or later.

When the firmware receives this command from the host, the firmware is obligated to send back a response packet on the HID interrupt IN endpoint. For PIC16 and PIC18 devices, the QUERY_EXTENDED_INFO has the following structure (note: format is architecture specific, and will not necessarily be the same for devices identifying themselves as PIC24 or other devices):

```
//Structure for the QUERY_EXTENDED_INFO command (and response)
struct{
    unsigned char Command;
    unsigned int BootloaderVersion;
    unsigned int ApplicationVersion;
    unsigned long SignatureAddress;
    unsigned int SignatureValue;
    unsigned long ErasePageSize;
    unsigned char Config1LMask;
    unsigned char Config1HMask;
    unsigned char Config2LMask;
    unsigned char Config2HMask;
    unsigned char Config3LMask;
    unsigned char Config3HMask;
    unsigned char Config4LMask;
    unsigned char Config4HMask;
    unsigned char Config5LMask;
    unsigned char Config5HMask;
    unsigned char Config6LMask;
    unsigned char Config6HMask;
```

```
unsigned char Config7LMask;
unsigned char Config7HMask;
};
```

The “ConfigxH/LMask” values in the response structure should be loaded with the appropriate AND mask values that the PC application should use when performing the verify operation, and comparing the read out contents of memory versus the .hex file contents. Generally speaking, unimplemented configuration bit positions should be excluded from the verify operations, since the .hex file may contain a ‘1’ or ‘0’ in these locations, even though the microcontroller hardware may not implement some of these locations. If the PC GUI application were to attempt to verify these locations, they would therefore always read back as the default unimplemented value (ex: 0 for PIC18 devices), which may not necessarily match the .hex file (ex: if it had a ‘1’ in the same location). Therefore, the firmware should properly respond to the host’s QUERY_EXTENDED_INFO command, with AND mask values that the PC GUI program should use to AND the .hex file values before comparing them to the read values, ensuring that the verify operation successfully ignores unimplemented/unimportant locations.

1.6.3.2.3.1.3 Changing the Memory Footprint

Describes how to change the location and size of the memory that the boot loader uses.

Description

By default, the bootloader firmware project reserves and occupies the 0x000-0xFFFF region (PIC18) or 0x000-0x8FF (PIC16) program memory region. It is possible to move this boundary (ex: if more space is needed for the bootloader firmware, such as when trying to build/use the bootloader with a free/non-optimizing version of the C compiler).

In order to move the boundary, the new boundary must be chosen that aligns perfectly with a native flash memory erase block size (ex: the boundary cannot reside in the middle of an flash memory erase page). Additionally, moving the boundary requires changing the bootloader definitions (REMAPPED_APPLICATION_RESET_VECTOR, REMAPPED_APPLICATION_HIGH_ISR_VECTOR, REMAPPED_APPLICATION_LOW_ISR_VECTOR, APP_SIGNATURE_ADDRESS, APP_VERSION_ADDRESS) in both the bootloader firmware project, as well as in the application firmware project intended to be programmed by the bootloader firmware.

When moving the APP_SIGNATURE_ADDRESS, make sure that the newly selected address always resides fully on the very first flash memory erase page that gets erased by the bootloader, in order to avoid defeating the robustness features offered by the flash signing process.

Additionally, both the bootloader firmware project and application linker settings must be modified. When using the C18 compiler, this entails modifying both the application and bootloader firmware .lkr files, so as to allocate more space for the bootloader. When using the XC8 compiler, the linker settings must be modified in both the bootloader firmware build configuration, as well as the application firmware image project build configuration.

1.6.3.2.3.1.4 C18 Compiler

Discusses how to use the C18 compiler with the boot loader examples.

Description

The C18 compiler only supports PIC18 based targets, and therefore, this section is only relevant for PIC18 devices. For PIC16 or PIC18 targets using the XC8 compiler, see the XC8 Compiler section.

The PIC18F HID bootloader firmware is compatible with both the C18 and XC8 compilers. When the firmware is build with the C18 compiler, the bootloader project needs to be built with the all of the compiler optimizations full enabled, and using the Default Storage Class “Static”. Using a free/unlicensed version of the compiler will not directly work, as some of the compiler optimizations are disabled in the free version.

If the firmware project is built without the full compiler optimizations enabled, the total code size will be too large to fit within the program memory range 0x000-0xFFFF reserved by default for the bootloader firmware. It is possible, albeit not preferred, to move the bootloader/application program memory space boundary, so as to allocate/reserve more flash memory for the bootloader, if the bootloader will be built without full compiler optimizations enabled. See “Moving the Application/Bootloader Flash Boundary”.

The bootloader project is provided with the modified linker scripts to be used with the bootloader project (ex:

BootModified.18f14k50_g.lkr), and modified example linker scripts that can be used with the application projects (ex: Linker files for applications\ rm18f14k50_g.lkr). However, certain application projects may require a slightly modified application linker script (ex: projects using large RAM buffers > 256 bytes in size for instance), but the provided application linker files should still be used as a template/starting point when configuring the linker script.

1.6.3.2.3.1.5 XC8 Compiler

Discusses using the XC8 compiler with the USB boot loader.

Description

1.6.3.2.3.1.5.1 Optimization Mode Requirements

Discusses compiler optimization requirements for using the USB boot loader.

Description

The currently implemented bootloader firmware may be built with the XC8 compiler v1.21 or later (in PRO mode), for both PIC16 and PIC18 devices.

On PIC16 targets this bootloader firmware is intended to occupy the program memory region 0x000-0x8FF (14-bit word addresses). The application firmware is supposed to occupy the 0x900-[end of flash] region of program memory (Note: The addresses [0x900-0x903] are technically part of the application firmware project space, but they must be reserved for the flash signature word and application version word values).

On PIC18 targets, the bootloader firmware occupies the 0x000-0xFFFF program memory region, while the application firmware image is supposed to occupy the 0x1000-[end of flash] region of program memory.

In order to fit within the 0x000-0x8FF (or 0x000-0xFFFF for PIC18) region, this bootloader project must be built with the PRO mode optimizations enabled. If you attempt to build the bootloader firmware in either the Free or Standard modes, the code size required for the bootloader firmware will likely be too large to fit within the reserved space for the bootloader, resulting in one or more linker errors (with text typically referring to “cannot find space” for “such and such” section name).

It is therefore recommended to use the bootloader with the fully licensed PRO version of the XC8 compiler. If however a free or standard version of the compiler must be used, it is potentially possible to move the bootloader/application program space boundary. If this will be attempted, please see the Changing the Memory Footprint section.

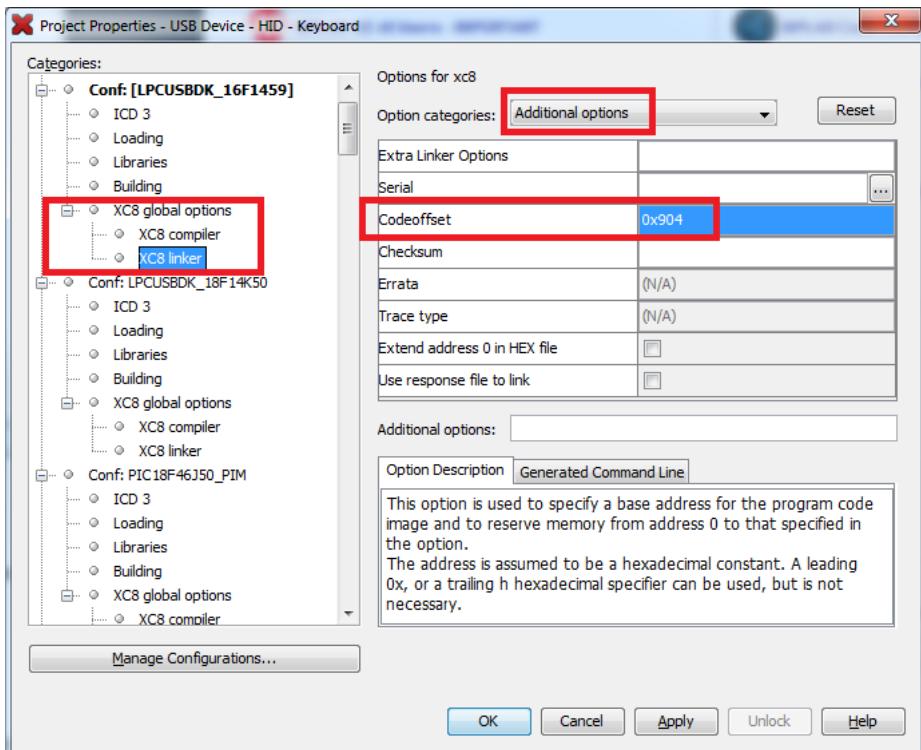
1.6.3.2.3.1.5.2 Linking Options for PIC16 Devices

Link option modifications that are required for using XC8 on PIC16 devices for the USB boot loader.

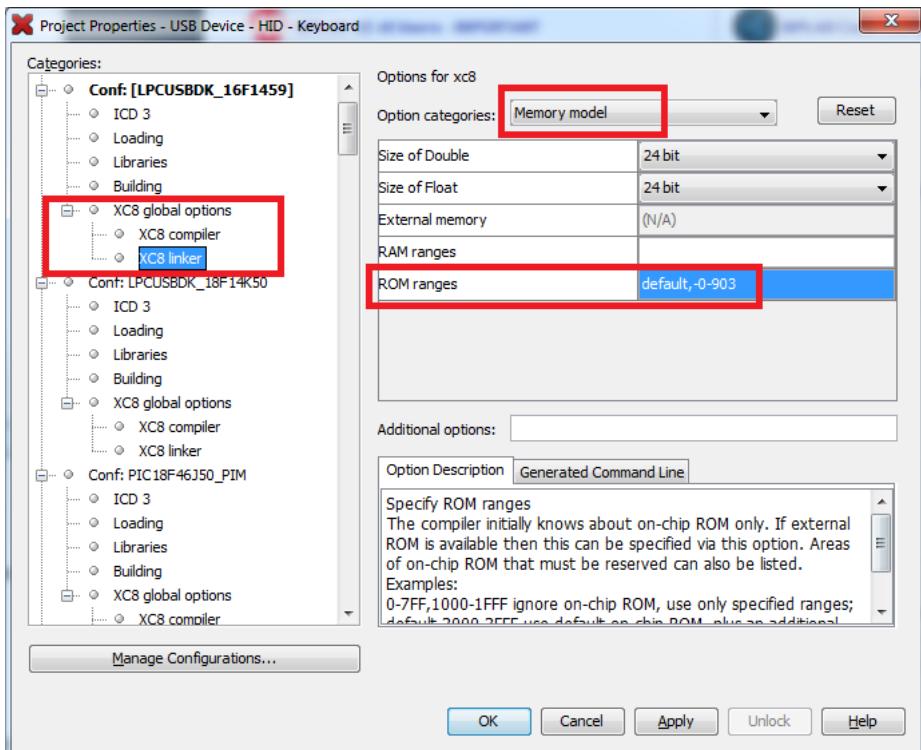
Description

If the application firmware project needs to be programmable by this bootloader firmware, there are two linker setting changes that are required to the application project:

- 1) Under the build configuration > XC8 global options > XC8 linker > Option categories: Additional options, the “Codeoffset” must be set to 0x904



2) Under the build configuration > XC8 global options > XC8 linker > Option categories: Memory Model, the "ROM Ranges" must be set to: default, -0-903



NOTE: Once the above changes are made to the application project, the output .hex file will no longer work when programmed stand alone, but it will be programmable by the HID bootloader firmware. This can make further application development less convenient, until the bootloader and application project output is "merged" using the procedures described in the Merging Bootloader and Application Project Output section.

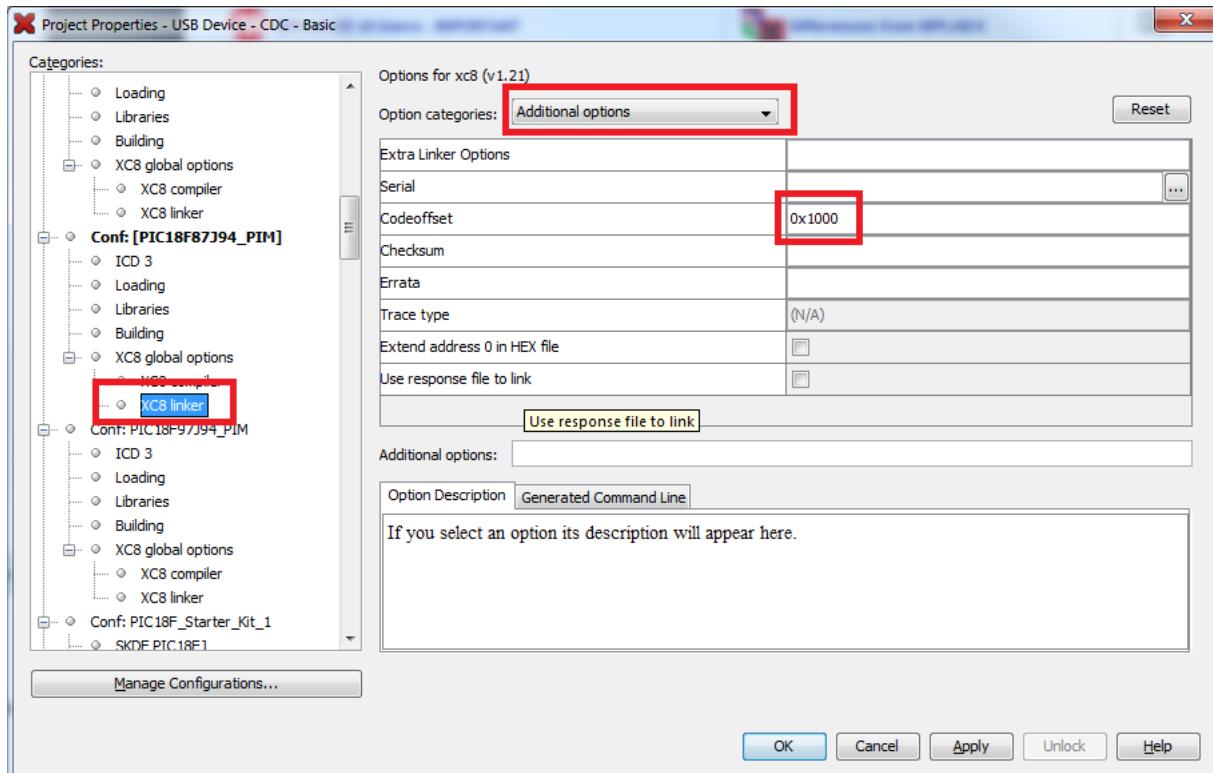
1.6.3.2.3.1.5.3 Linking Options for PIC18 Devices

Link option modifications that are required for using XC8 on PIC18 devices for the USB boot loader.

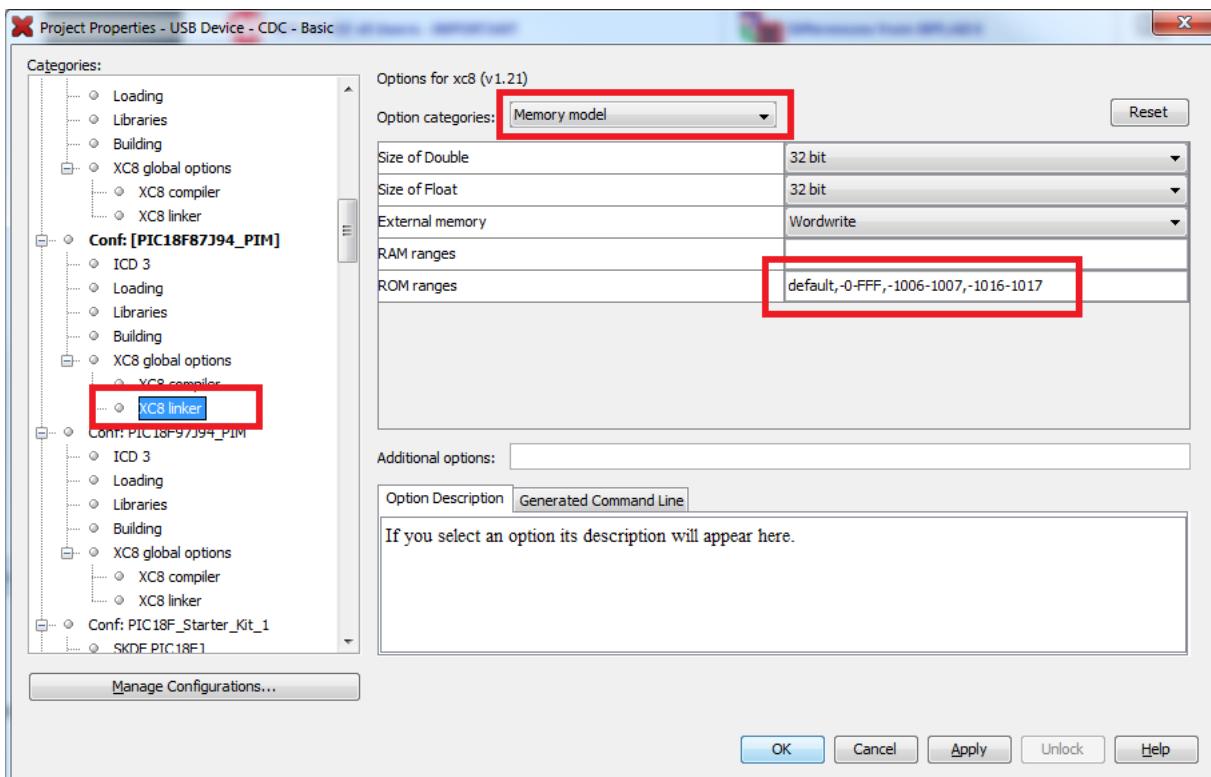
Description

If the application firmware project needs to be programmable by this bootloader firmware, there are two linker setting changes that are required to the application project:

- 1) Under the build configuration > XC8 global options > XC8 linker > Option categories: Additional options, the “Codeoffset” must be set to 0x1000



- 2) Under the build configuration > XC8 global options > XC8 linker > Option categories: Memory Model, the “ROM Ranges” must be set to: default, -0-FFF, -1006-1007, -1016-1017



NOTE: Once the above changes are made to the application project, the output .hex file will no longer work when programmed stand alone, but it will be programmable by the HID bootloader firmware. This can make further application development less convenient, until the bootloader and application project output is “merged” using the procedures described in the Merging Bootloader and Application Project Output section.

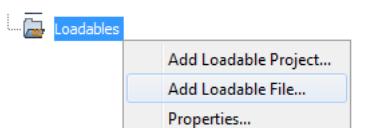
1.6.3.2.3.1.6 Merging Bootloader and Application Project Output

Discusses how to generate a single .hex file that contains both the boot loader and the application project output.

Description

To “merge” the two projects together (so that the application firmware .hex file is not required to be programmed by the bootloader, but may be programmed either stand alone via ICSP or by the bootloader), the following can be done:

1. Open and build the HID bootloader firmware project in the MPLAB X IDE.
2. Open the application firmware project in the MPLAB X IDE.
3. Add the “MPLAB.X.production.hex” file (generated by the HID bootloader firmware project) to the application firmware project as a “Loadable” file in MPLAB X. This can be done from the project view window and right clicking on the “Loadables” folder, and selecting the “Add Loadable File...” option.



4. Build the application firmware project. After the project builds, MPLAB X will automatically run the “HEXMATE” utility in the background and generate a merged .hex file (which will contain both the bootloader firmware and application firmware). This merged .hex file can be programmed either stand alone with an ICSP™ programmer or using the bootloader.

If step number 4 above fails, this implies that there are overlapping but non-equivalent sections of the program memory values in the two hex files (from the HID bootloader firmware and the application project). Normally, this error will be caused by having slightly different configuration bit settings between the application firmware project and the HID bootloader firmware project. To fix this error, you must modify the two projects so that the configuration bits are 100% identical between the two or are only implemented in one of the firmware projects.

Alternatively, it is possible to use the HEXMATE utility directly from the command line, instead of invoking it through the MPLAB X IDE through the "Loadables" feature to merge the hex files. When used at the command line, the HEXMATE utility supports a variety of advanced options, such as code offset shifting, giving "priority" to one hex file or another (ex: in the case of conflicts due to overlapping but non-identical .hex file content), merging more than two hex files at once, etc. To learn more about how to use the HEXMATE utility, please refer to the "HEXMATE" section in the MPLAB XC8 compiler user's manual.

1.6.3.2.3.2 PIC24F

This section covers the PIC24F product line USB boot loaders.

Description

This section covers the PIC24F product line USB boot loaders.

1.6.3.2.3.2.1 Adding a boot loader to your project

This section covers how to add a boot loader to your application.

Description

The boot loader implementations available in the USB Library take a two application approach. What this means is that the boot loader and the application are development, compiled, and loaded separately.

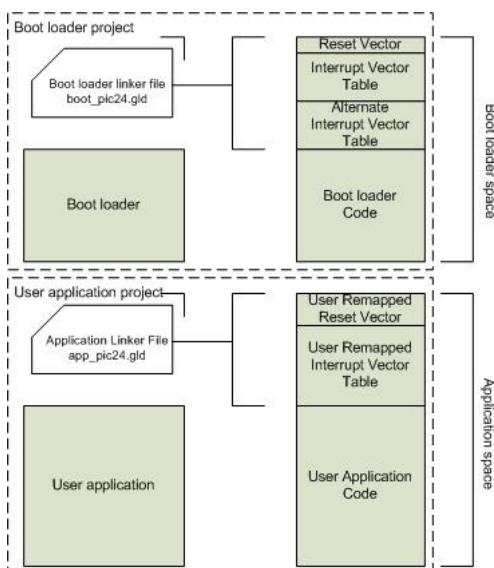
With this approach there are two separate linker scripts that are required, one for the boot loader, and one for the application.

For the PIC24F applications intended to be used with a boot loader, all that is required is to attach the specific linker file designed for the applications of that boot loader to the project.

- No modifications are required to the linker file. Just attach the provided application linker file to the application project without modification.
- No modifications are required to the application code. Just write your code as you always would and attach the provided application linker file to the application project without modification.

The required application linker files are found in the folders that contain the targeted boot loaders. These linker files can be referenced directly from the application projects, or can be copied locally to the project folder.

These provided linker files generate the required code to handle the reset and interrupt remapping sections that are required.



1.6.3.2.3.2.2 Memory Map

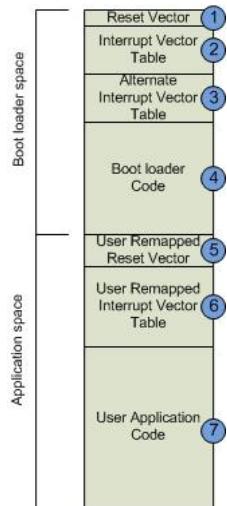
This section discusses the various memory regions in the PIC24F device and how they are arranged between the boot

loaders and the target applications.

Description

The PIC24F boot loaders have several different special memory regions. Some of these regions are defined by the hardware. Others are part of the boot loader implementation and usage. This section discusses what each of these memory regions are. For more information about how these sections are implemented or how to change them, please refer to the Understanding and Customizing the Boot Loader Implementation section.

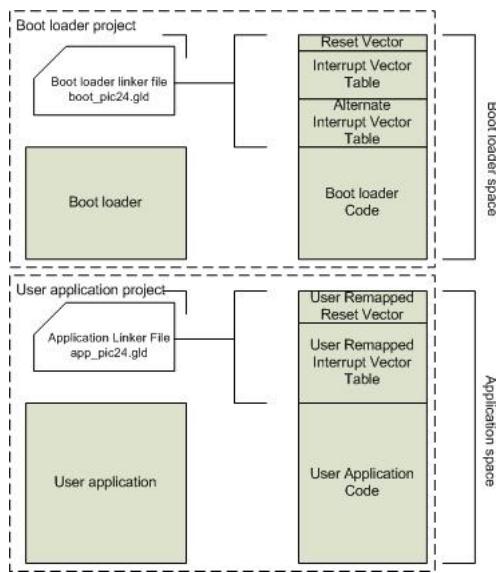
The different memory regions are shown below:



- 1) Reset Vector - the reset vector is defined by the hardware. This is located at address 0x0000. Any reset of the CPU will go to the reset vector. The main responsibility of the reset vector is to jump to the code that needs to be run. In the case of the boot loader, this means jumping to the boot loader code (section 4).
- 2) The interrupt vector table (IVT) is another section that is defined by the PIC24F hardware. The IVT is a fixed set of addresses that specify where the CPU should jump to in the case of an interrupt event. Each interrupt has its own vector in the table. When that interrupt occurs, the CPU fetches the address in the table corresponding to that interrupt and jumps to that address.
- 3) The alternate interrupt vector table (AIvt) behaves just like the IVT. The user must set a bit to select if they are using the IVT or AIvt for their interrupt handling. The IVT is the default. For the current boot loader applications, the AIvt is either used by the boot loader or is not remapped to user space so the AIvt is not available for application use.
- 4) The boot loader code - This section is where the boot loader code resides. This section handles all of the loading of the new application code.
- 5) User Remapped Reset Vector - This is a section that is defined by the boot loader. The boot loader must always know how to exit to the application on startup. The User Remapped Reset Vector is used as a fixed address that the boot loader can jump to in order to start an application. The application must place code at this address that starts their application. In the PIC24F implementations this is handled by the application linker file.
- 6) User Remapped Interrupt Vectors - Since the IVT is located in the boot loader space, the boot loader must remap all of the interrupts to the application space. This is done using the User remapped interrupt vectors. The IVT in the boot loader will jump to a specific address in the User remapped interrupt vector. The User remapped interrupt vector table jumps to the interrupt handler code defined in the user code. In the PIC24F implementations this table is generated by the application linker file and doesn't require any user modifications.
- 7) The user application code - this is the main application code for the project that needs to be loaded by the boot loader. In the PIC24F implementation, only the application linker file for the specific boot loader needs to be added to the project. No other files are required. No changes or additions are required to the user application code either in order to get the code working.

The boot loader and application linker files provided with the USB Library enforce all of the memory regions specified above. If an application tries to specify an address outside of the valid range, the user should get a linker error.

Separate linker files are required for the boot loader and the application. These linker files generate the material required for several of the different memory regions in the device. Below is a diagram showing which sections of the final device image are created by the linker files. All of the regions of the device are specified within one of the two linker files. This image merely shows where the content for each of those regions is generated.



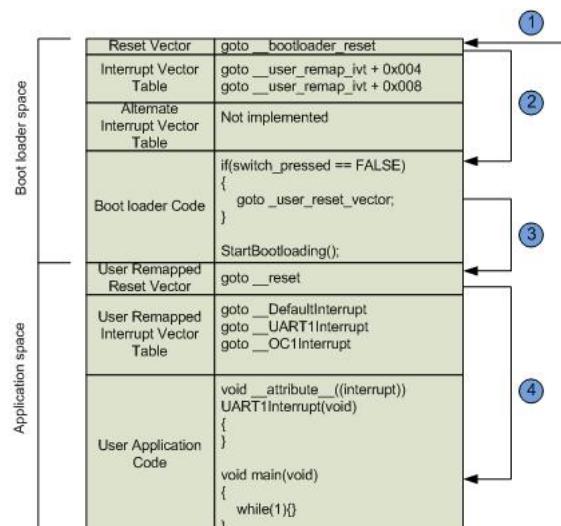
1.6.3.2.3.2.3 Startup Sequence and Reset Remapping

This section discusses how the device comes out of reset and how the control passes between the boot loader and the application.

Description

Before continuing with this section, please review the preceding sections to understand some of the implementation details that aren't discussed in detail in this section. Some of the implementation details of how this works is described the Understanding and Customizing the Boot Loader Implementation section. This section covers the basic flow and how it passes between the boot loader and the application.

In the boot loader implementations provided in USB Library library, the boot loader controls the reset vector. This is true for the PIC24F boot loaders as well. The reset vector resides within the boot loader memory space. This means that the boot loader must jump to the target application. This processes in show below in the following diagram and described in the following paragraphs.



- 1) On PIC24F devices, when a reset occurs the hardware automatically jumps to the reset vector. This is located at address

0x0000. This address resides within the boot loader memory. The compiler/linker for the boot loader code places a 'goto' instruction at the reset vector to the boot loader startup code.

- 2) The 'goto' instruction at the reset address will jump to the main() function for the boot loader.
- 3) In the boot loader startup sequence there is a check to determine if the boot loader should run or if the boot loader should jump to the application instead. In the provided examples the code checks a switch to determine if it should remain in the boot loader. If the switch is not pressed then the boot loader jumps to the user_remapped_reset_vector. At this point the control of the processor has just changed from the boot loader to the application.
- 4) The code at the user_remapped_reset_vector is controlled by the application project, not the boot loader. This vector effectively emulates the behavior that the normal reset vector would if a boot loader wasn't used. In this case it should jump to the startup code for the application. This is done by modified linker script for the application.

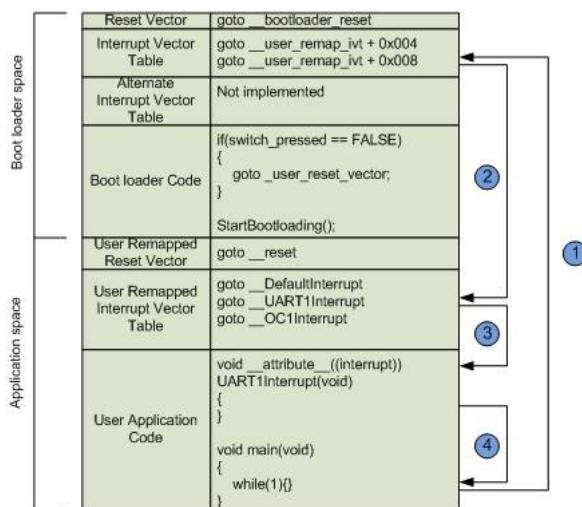
1.6.3.2.3.2.4 Interrupt Remapping

This section discusses how interrupts are handled between the boot loader and application.

Description

Before continuing with this section, please review the preceding sections to understand some of the implementation details that aren't discussed in detail in this section. Some of the implementation details of how this works is described the Understanding and Customizing the Boot Loader Implementation section. This section covers the basic flow and how it passes between the boot loader and the application.

In the boot loader implementations provided in USB Library library, the boot loader controls the interrupt vectors for PIC24F devices. The hardware interrupt vector table resides within the boot loader memory space. This means that the boot loader must jump to the appropriate user target application interrupt handler when an interrupt occurs. This processes in show below in the following diagram and described in the following paragraphs.



- 1) During the course of normal code execution, an interrupt occurs. The CPU vectors to the interrupt vector table (IVT) as described in the appropriate PIC24F datasheet.
- 2) The IVT is located in boot loader space, but the application needs to handle the interrupt. The boot loader jumps to the correct entry in the User Remapped Interrupt Vector Table. At this point the CPU is jumping from the boot loader memory space to the application memory space and effectively transferring control to the application.
- 3) At the entry in the User Remapped Interrupt Vector table there is placed a 'goto' instruction that will jump to the appropriate interrupt handler if one is defined in your application and to the default interrupt if there isn't a handler defined. In this way the behavior of the application with or without the boot loader is identical. The User Remapped Interrupt Vector table is created by the application linker file for the specific boot loader in use. This table is automatically generated and doesn't need to be modified. More about how this table is generated can be found in the Understanding and Customizing the Boot Loader Implementation.
- 4) Finally once the interrupt handler code is complete, the code will return from the interrupt handler. This will return the CPU

to the instruction that the interrupt occurred before.

1.6.3.2.3.2.5 Understanding and Customizing the Boot Loader Implementation

This section discusses the customizations that have been made from the default linker scripts in order to make the boot loader work and how to customize these implementations if you wish to change the behavior or location of the boot loader.

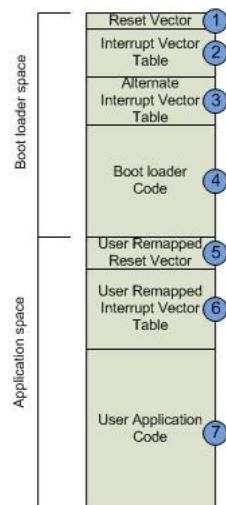
Description

1.6.3.2.3.2.5.1 Memory Region Definitions

This section describes how each of the memory regions gets defined.

Description

First let's take a look how each of the memory regions are defined. The address ranges for each of the regions seen in the diagram below must be defined in either the application linker file or the boot loader linker files.



Below is an excerpt from one of the HID boot loader linker files. This is from the linker script for the boot loader itself so this will be covering sections (1), (2), (3), and (4).

```
/*
** Memory Regions
*/
MEMORY
{
    data  (a!xr) : ORIGIN = 0x800,           LENGTH = 0x4000
    reset      : ORIGIN = 0x0,                LENGTH = 0x4
    ivt       : ORIGIN = 0x4,                LENGTH = 0xFC
    aivt      : ORIGIN = 0x104,              LENGTH = 0xFC
    program (xr) : ORIGIN = 0x400,            LENGTH = 0x1000
    config4   : ORIGIN = 0x2ABF8,             LENGTH = 0x2
    config3   : ORIGIN = 0x2ABFA,             LENGTH = 0x2
    config2   : ORIGIN = 0x2ABFC,             LENGTH = 0x2
    config1   : ORIGIN = 0x2ABFE,             LENGTH = 0x2
}
```

The region named "reset" is defined to start at address 0x0 and has a length of 0x4. This means that the first two instructions of the device are used for the reset vector. This is just enough for one 'goto' instruction. This corresponds to hardware implementation and should not be changed. This defines section (1).

Section (2) is the IVT table. This is defined with the "ivt" memory entry. It starts at address 0x4 and is 0xFC bytes long. This corresponds to hardware implementation and should not be changed.

Section (3) is the AIVT table. This is defined with the "aivt" memory entry. It starts at address 0x104 and is 0xFC bytes long. This corresponds to hardware implementation and should not be changed.

Section (4) is the section for the boot loader code. This section is covered by the "program" entry in the memory table. This section starts at address 0x400 and is 0x1000 bytes long in this example (ends at 0x1400). As you can see with this section it has been decreased from the total size of the device to limit the boot loader code to this specific area. This is how the linker knows where the boot loader code is allowed to reside.

Looking in the corresponding application linker file will result in a similar table.

```
/*
** Memory Regions
*/
MEMORY
{
    data  (a!xr)      : ORIGIN = 0x800,          LENGTH = 0x4000
    reset           : ORIGIN = 0x0,             LENGTH = 0x4
    ivt             : ORIGIN = 0x4,             LENGTH = 0xFC
    aivt            : ORIGIN = 0x104,            LENGTH = 0xFC
    app_ivt         : ORIGIN = 0x1400,            LENGTH = 0x10C
    program (xr)    : ORIGIN = 0x1510,            LENGTH = 0x296E8
    config4          : ORIGIN = 0x2ABF8,            LENGTH = 0x2
    config3          : ORIGIN = 0x2ABFA,            LENGTH = 0x2
    config2          : ORIGIN = 0x2ABFC,            LENGTH = 0x2
    config1          : ORIGIN = 0x2ABFE,            LENGTH = 0x2
}
```

Note that the "reset", "ivt", and "aivt" sections are all still present in the application linker script. These sections remain here so that applications compiled with the boot loader can be programmed with or without the boot loader. This aids in the development of the application without having to use the boot loader while maintaining identical interrupt latency and memory positioning.

Sections (5) and (6) are created in the special "app_ivt" section. The following discussion topic describes how the content of this section is created. This entry in the memory table is how the space for that area is allocated. Note that the "app_ivt" section starts at address 0x1400 (the same address that the boot loader ended at). Since different parts have different number of interrupts, the size of the "app_ivt" section may change.

The "program" memory section has changed for the application space. It starts at address 0x1510 in this example. This will vary from part to part based on the size of the "app_ivt" section. The "program" memory section corresponds to the user application code (section (7)). Note that it takes up the rest of the memory of the device that is available to load.

1.6.3.2.3.2.5.2 Special Region Creation

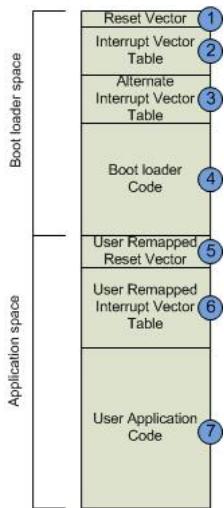
This section covers how each of the special memory regions are created/populated within the linker files.

Description

The Memory Region Definitions section described how each of the memory regions are defined. This allocates the room for each of the memory regions.

This discussion covers how the values of some of the special memory regions are created/populated. Please refer to the earlier sections for an understanding of how the reset and interrupt remapping works before proceeding through this section.

Let's take a look at each of the memory regions in order. Please note that there are two linker scripts, one for the boot loader and one for the application. In order for some of these section definitions to make sense, we will be showing excerpts from either or both of these files for any given section. Please pay close attention to which linker file we are referring to when we show an example.



- Section (1) is the reset vector. This belongs to the boot loader space so this is located in the boot loader linker file. What we need at the reset vector is a jump to the start of the boot loader code. In the boot loader linker script:

```
/*
 ** Reset Instruction
 */
.reset :
{
    SHORT(ABSOLUTE(__reset));
    SHORT(0x04);
    SHORT((ABSOLUTE(__reset) >> 16) & 0x7F);
    SHORT(0);
} >reset
```

The code in this section generates a "goto __reset" instruction located in the "reset" memory section. This will cause the CPU to jump to the boot loader startup code after any device reset. This is common code that is present in any default linker script for PIC24F.

- The second section is the IVT. In the IVT we need to jump to the user's remapped IVT table.

```
_APP_IVT_BASE = 0x1400;
.ivt __IVT_BASE :
{
    LONG(ABSOLUTE(__APP_IVT_BASE) + 0x004); /* __ReservedTrap0*/
    LONG(ABSOLUTE(__APP_IVT_BASE) + 0x008); /* __OscillatorFail*/
    LONG(ABSOLUTE(__APP_IVT_BASE) + 0x00C); /* __AddressError*/
    LONG(ABSOLUTE(__APP_IVT_BASE) + 0x010); /* __StackError*/
    LONG(ABSOLUTE(__APP_IVT_BASE) + 0x014); /* __MathError*/
...
    LONG(ABSOLUTE(__DEFAULT_VECTOR)); /* __Interrupt116 not implemented */
    LONG(ABSOLUTE(__DEFAULT_VECTOR)); /* __Interrupt117 not implemented */
} >ivt
```

This linker code will place the _APP_IVT_BASE constant + an offset address at each of the IVT vector entries. This will cause the CPU to jump to the specified vector in the user's remapped IVT table.

Note that each entry is 4 bytes away from the previous entry. Is is because the resulting remapped IVT will need to use "goto" instructions at each entry in order to reach the desired handler. The "goto" instruction takes two instruction words at 2 bytes of memory address each.

- Section (3), the AIVT, is either not used or is used by the boot loader and shouldn't be used by the application. If the boot loader requires interrupts, then it uses the AIVT and switches to AIVT interrupts before starting and switches back to the IVT before jumping to the customer code. No linker modifications are required here. For boot loaders that don't require interrupts, some have the AIVT section removed since they are not remapped to the user space and not used by the boot loader.

- Section (4), the boot loader code - the only modification required in the linker script for the boot loader code is the changes to the memory region definitions discussed previously in the Memory Region Definitions section.

- Section (5) is the user remapped reset. This is the address where the boot loader jumps upon completion. This address

needs to be at a fixed location in code that both the boot loader and the application know about. At this address there needs to be a jump to the user application code. In the application linker script:

```
.application_ivt __APP_IVT_BASE :
{
    SHORT(ABSOLUTE(__reset)); SHORT(0x04); SHORT((ABSOLUTE(__reset) >> 16) & 0x7F);
SHORT(0);
    SHORT(DEFINED(__ReservedTrap0) ? ABSOLUTE(__ReservedTrap0) :
ABSOLUTE(__DefaultInterrupt)); SHORT(0x04); SHORT(DEFINED(__ReservedTrap0) ?
(ABSOLUTE(__ReservedTrap0) >> 16) & 0x7F : (ABSOLUTE(__DefaultInterrupt) >> 16) & 0x7F);
SHORT(0);
    SHORT(DEFINED(__OscillatorFail) ? ABSOLUTE(__OscillatorFail) :
ABSOLUTE(__DefaultInterrupt)); SHORT(0x04); SHORT(DEFINED(__OscillatorFail) ?
(ABSOLUTE(__OscillatorFail) >> 16) & 0x7F : (ABSOLUTE(__DefaultInterrupt) >> 16) & 0x7F);
SHORT(0);
    SHORT(DEFINED(__AddressError) ? ABSOLUTE(__AddressError) :
ABSOLUTE(__DefaultInterrupt)); SHORT(0x04); SHORT(DEFINED(__AddressError) ?
(ABSOLUTE(__AddressError) >> 16) & 0x7F : (ABSOLUTE(__DefaultInterrupt) >> 16) & 0x7F);
SHORT(0);
```

This section of code has been added to the default linker script. This creates a section in code located at `__APP_IVT_BASE` address. In this case the `__APP_IVT_BASE` address is also defined in the application linker file:

```
__APP_IVT_BASE = 0x1400;
```

This address must match exactly between the boot loader code, boot loader linker file, and the application linker file. If any of these do not match then the linkage between the interrupt remapping or reset remapping will not work and the application will fail to run properly.

The first entry in this table is the user remapped reset. This code generates a "goto `__reset`" at address `__APP_IVT_BASE`. This allows the boot loader to jump to this fixed address to then jump to the start of the user code (located at the `__reset` label).

6) Section (6) is the remapped IVT table. This section allows the interrupt to be remapped from the boot loader space to the application space. In order to do this the boot loader must either know the exact address of every interrupt handler, or must have another jump table that it jumps to in order to redirect it to the correct interrupt handler. The second approach is the one used in the implemented boot loaders. This is implemented in the following table:

```
.application_ivt __APP_IVT_BASE :
{
    SHORT(ABSOLUTE(__reset)); SHORT(0x04); SHORT((ABSOLUTE(__reset) >> 16) & 0x7F);
SHORT(0);
    SHORT(DEFINED(__ReservedTrap0) ? ABSOLUTE(__ReservedTrap0) :
ABSOLUTE(__DefaultInterrupt)); SHORT(0x04); SHORT(DEFINED(__ReservedTrap0) ?
(ABSOLUTE(__ReservedTrap0) >> 16) & 0x7F : (ABSOLUTE(__DefaultInterrupt) >> 16) & 0x7F);
SHORT(0);
    SHORT(DEFINED(__OscillatorFail) ? ABSOLUTE(__OscillatorFail) :
ABSOLUTE(__DefaultInterrupt)); SHORT(0x04); SHORT(DEFINED(__OscillatorFail) ?
(ABSOLUTE(__OscillatorFail) >> 16) & 0x7F : (ABSOLUTE(__DefaultInterrupt) >> 16) & 0x7F);
SHORT(0);
    SHORT(DEFINED(__AddressError) ? ABSOLUTE(__AddressError) :
ABSOLUTE(__DefaultInterrupt)); SHORT(0x04); SHORT(DEFINED(__AddressError) ?
(ABSOLUTE(__AddressError) >> 16) & 0x7F : (ABSOLUTE(__DefaultInterrupt) >> 16) & 0x7F);
SHORT(0);
```

This first entry in the table is the remapped reset vector that we just discussed. The second entry in the table is the first possible interrupt. In this case it is the `ReservedTrap0` interrupt. This line of linker code will look for the `__ReservedTrap0` interrupt function. If it exists it will insert a "goto `__ReservedTrap0`" at the second address in this table. If it doesn't find the `__ReservedTrap0` function, it will put a "goto `__DefaultInterrupt`" at this entry in the table. In this way just by defining the appropriate interrupt handler function in the application code, the linker will automatically create the jump table entry required.

Looking at an example application_ivt table as generated by the linker script where the `ReservedTrap0` interrupt is not defined and the `OscillatorFail` and `AddressError` handlers are defined, starting at address `__APP_IVT_BASE` you will have the following entries in program memory:

```
goto __reset
```

```

goto __DefaultInterrupt
goto __OscillatorFail
goto __AddressError
...

```

7) Section (7), the user application code - the only modification to the linker script required for the application code is the changes to the memory region definitions discussed previously in the Memory Region Definitions section.

1.6.3.2.3.2.5.3 Changing the memory footprint of the boot loader

This section covers how to modify how much memory is used by the boot loader. This can be useful when adding features to the boot loader that increase the size beyond the default example or if a version of the compiler is used that doesn't provide a sufficient level of optimizations to fit the default boot loader.

Description

This section covers how to modify the size of the HID boot loader. This can be useful when adding features to the boot loader that increase the size beyond the default example or if a version of the compiler is used that doesn't provide a sufficient level of optimizations to fit the default boot loader. The boot loaders provided by default assume full optimizations and may not work with compilers that don't have access to full optimizations.

Please read all of the other topics in the PIC24F boot loader section before proceeding in this topic. This topic will show where the modifications need to be made and how they need to match up, but will not describe what the sections that are being modified are or how they are implemented. This information is in previous sections.

There are three places that require corresponding changes: the boot loader linker script, the application linker script, and the boot loader code. You may wish to make copies of the original files so that you preserve the original non-modified files.

In the following examples we will be increasing the size of the boot loader from 0x1400 to 0x2400 in length.

First start by determining the size that you want the boot loader to be. This must be a multiple of an erase page. On many PIC24F devices there is a 512 instruction word erase page (1024 addresses per page). Please insure that the address you select for the end of the boot loader corresponds to a page boundary. There are several ways to determine the size of the boot loader application. Below is an example of one method.

- 1) Remove the boot loader linker script provided if it is causing link errors due either to optimization settings or added code.
- 2) Build the project
- 3) Open the memory window and find the last non-blank address in the program memory space.
- 4) Find the next flash erase page address after this address. Add any additional buffer room that you might want for future boot loader development, growth, or changes. Use this address as your new boot loader end address.

Once the end address of the boot loader is known, start by modifying the boot loader linker script program memory region to match that change. The boot loader linker script can either be found in the folder containing the boot loader project file or in a folder that is specified for boot loader linker scripts. In the linker script find the memory regions.

```

MEMORY
{
data (a!xr) : ORIGIN = 0x800, LENGTH = 0x4000
reset : ORIGIN = 0x0, LENGTH = 0x4
ivt : ORIGIN = 0x4, LENGTH = 0xFC
aivt : ORIGIN = 0x104, LENGTH = 0xFC
program (xr) : ORIGIN = 0x400, LENGTH = 0x2000
config4 : ORIGIN = 0x2ABF8, LENGTH = 0x2
config3 : ORIGIN = 0x2ABFA, LENGTH = 0x2
config2 : ORIGIN = 0x2ABFC, LENGTH = 0x2
config1 : ORIGIN = 0x2ABFE, LENGTH = 0x2
}

```

Change the LENGTH field of the program memory section to match the new length. Note that this is length and not the end address. To get the end address, please add LENGTH + ORIGIN.

Next, locate the __APP_IVT_BASE definition in the linker file. Change this to equal the end address of your boot loader.

```
__APP_IVT_BASE = 0x2400;
```

Once the length of the boot loader is changed, you will need to make similar changes in the application boot loader linker script. The application boot loader linker scripts are typically found in a folder with the boot loader project. In the application linker file, locate the memory regions section. In this section there are three items that need to change.

1. The first is the ORIGIN of the app_ivt section. This needs to be modified to match the new end address of the boot loader.
2. Second, move the ORIGIN of the program memory section to the ORIGIN of app_ivt + the LENGTH of the app_ivt section so that the program memory starts immediately after the app_ivt section.
3. Last, change the LENGTH field of the program section so that it goes to the end of the program memory of the device. Remember that the LENGTH field is the length starting from the origin and not the end address. An easy way to make sure that this address is correct is by just subtracting off from the LENGTH the same amount that was added to the ORIGIN.

```
MEMORY
{
    data (a!xr) : ORIGIN = 0x800, LENGTH = 0x4000
    reset : ORIGIN = 0x0, LENGTH = 0x4
    ivt : ORIGIN = 0x4, LENGTH = 0xFC
    aivt : ORIGIN = 0x104, LENGTH = 0xFC
    app_ivt : ORIGIN = 0x2400, LENGTH = 0x110
    program (xr) : ORIGIN = 0x2510, LENGTH = 0x286E8
    config4 : ORIGIN = 0x2ABF8, LENGTH = 0x2
    config3 : ORIGIN = 0x2ABFA, LENGTH = 0x2
    config2 : ORIGIN = 0x2ABFC, LENGTH = 0x2
    config1 : ORIGIN = 0x2ABFE, LENGTH = 0x2
}
```

The final changes that needs to be made are in the boot loader code itself. Open up the boot loader project.

1. Find the ProgramMemStart definition in the main.c file. Change the start address to match the new address.

```
#define ProgramMemStart 0x00002400
```

2. Next find the #ifdef section that applies to the device that you are working with. This section will contain definitions used by the boot loader to determine what memory is should erase and re-write.

```
#if defined(__PIC24FJ256GB110__) || defined(__PIC24FJ256GB108__) ||
defined(__PIC24FJ256GB106__)
#define BeginPageToErase 5 //Bootloader and vectors occupy first six 1024 word (1536 bytes
due to 25% unimplemented bytes) pages
#define MaxPageToEraseNoConfigs 169 //Last full page of flash on the PIC24FJ256GB110, which
does not contain the flash configuration words.
#define MaxPageToEraseWithConfigs 170 //Page 170 contains the flash configurations words on
the PIC24FJ256GB110. Page 170 is also smaller than the rest of the (1536 byte) pages.
#define ProgramMemStopNoConfigs 0x0002A800 //Must be instruction word aligned address. This
address does not get updated, but the one just below it does:
//IE: If AddressToStopPopulating = 0x200, 0x1FF is the last programmed address (0x200 not
programmed)
#define ProgramMemStopWithConfigs 0x0002ABF8 //Must be instruction word aligned address.
This address does not get updated, but the one just below it does: IE: If
AddressToStopPopulating = 0x200, 0x1FF is the last programmed address (0x200 not programmed)
#define ConfigWordsStartAddress 0x0002ABF8 //0x2ABFA is start of CW3 on PIC24FJ256GB110
Family devices
#define ConfigWordsStopAddress 0x0002AC00
```

3. Modify the BeginPageToErase to indicate which page is the first page it should erase. This will be the ProgramMemStart/Page Size. In this case we are starting at 0x2400 and each page is 0x400 so this should now be 9.

```
#define BeginPageToErase 9
```

4. Locate the start of the main() function. In the first few lines of code there is a check to determine if the code should stay in the boot loader or jump to the application code. Change the address in the "goto" statement to match the new end of the boot loader and start of the application.

```
__asm__("goto 0x2400");
```

This should be all of the changes required in order to change the size of the HID boot loader.

Please note that since the boot loader and the application code are developed as two separate applications, they do not need to use the same optimization settings.

1.6.3.2.4 Flash Signature

Discusses what a flash signature is, why it is important, and how it is used.

Description

The flash signature feature is a robustness/recoverability feature, which is particularly useful for applications that are not using an I/O pin for entry into the bootloader mode, and instead rely on entry into the bootloader only by software from the application firmware image.

Consider the following situation:

1. User boots up microcontroller and begins running application image.
2. User runs special PC application (or something similar) that sends command to the application image, to switch into the bootloader mode.
3. Firmware executes a goto 0x001C jump straight into the bootloader mode (via software entry).
4. User starts an erase/program/verify sequence using the PC GUI program for bootloading new application firmware images.
5. The firmware erases some or all of the application flash contents.
6. Before the flash has been reprogrammed with the new values, the user unplugs the USB cable and/or AC power is lost to the entire system.

At this point, the application would normally be permanently “bricked” (unless the user plugs in a conventional ICSP programmer like the MPLAB ICD3), since the application image would be corrupt or missing, and that may have been the only method for receiving the command to jump into bootloader mode.

The above scenario can however be made recoverable, through the use of a “flash signature” process.

The flash signature is a special program memory word that gets programmed (only after the entire erase/program/verify process is completed successfully) with a magic/known value. This value, when present and correctly programmed at the magic address with the proper value, indicates to the bootloader code that the application firmware image is fully intact.

A typical (successful) bootloading sequence, that uses a flash signature, would be as follows:

1. User boots up microcontroller, which first checks the flash signature word is intact, with the correct/expected value.
 1. Assuming the value is correct, this implies that the application image is intact, and the code jumps into the application firmware run mode.
2. User runs special PC application or something that sends command to the application image, to switch into the bootloader mode.
3. Firmware executes a goto 0x001C jump straight into the bootloader mode.
4. User starts an erase/program/verify sequence using the PC GUI program for bootloading new firmware images.
5. The firmware begins erasing pages of flash memory. Special care is taken in the implementation to ensure that the flash signature word is located on the very first flash erase page that gets erased.
6. After total erasure of the application image is complete, the PC GUI sends commands to reprogram the entire application firmware space with the new image.
7. The PC GUI performs a full verify read back of the flash contents, and verifies that every address contains exactly the correct values from the hex file.
8. Assuming the entire “verify” operation is successful, the PC GUI sends a “sign flash” command to the bootloader firmware.
9. The bootloader firmware programs the special/magic known value into the special/fixed signature address.

At this point the bootloading process is complete. Upon rebooting the microcontroller, the bootup code checks the flash signature address to verify that the contents of that flash memory word contain the correct/expected flash signature value.

1. If the value matches the correct/expected value, this implies that the previous erase/program/verify sequence was fully successful, and therefore, it is safe to jump into and begin executing the application firmware image.
2. If the value does not match (ex: the flash signature word contains an invalid or erased value, like 0xFFFF), then the bootup code knows that the previous erase/program/verify sequence failed at some point, and therefore, the bootup code makes sure to stay in bootload mode, allowing the PC GUI application to connect to the firmware and perform another

attempt to erase/program/verify/sign flash sequence.

NOTE: In order for the flash signature feature to fully protect the application from bricking in the event of USB cable disconnect and/or lost AC power, it is necessary for the flash signature word to be located on the very first erase page during the erase sequence, and must also be the very last portion of the program memory that gets re-programmed, only after the rest of the program/verify sequence has been fully completed successfully.

In the current implementation, the flash signature word is located within the application program memory space, at a specific fixed address. For PIC18 devices, the default address for the flash signature word is 0x1006 (and 1007 for the MSB). For PIC16 devices, the address for the flash signature word is 0x900.

1.6.4 Device - CDC Basic Demo

This example shows how to create a basic CDC demo. CDC devices appear like COM ports on the host computer and be communicated with via regular terminal software.

Description

Supported Demo Boards

The matrix of which demos are supported on a specific board can be found in the Release Notes demo board support section. Verify that the board you wish to use will work with this demo. This table also describes some of the limitations that the board might have while running this demo.

Demo Board I/O Mapping

Each demo board has a different number of push buttons, LEDs, and other features with various different names for these components. To determine which board features are used for which demo features, please refer to the io_mapping.h file in the demo folder under the system_config folder. Each demo board will have a corresponding folder with an io_mapping.h file in it. For example, for the PIC18F46J50 PIM this would be the following file:

```
<install_directory>/apps/usb/device/cdc_basic/firmware/src/system_config/pic18f46j50_pim/io_mapping.h
```

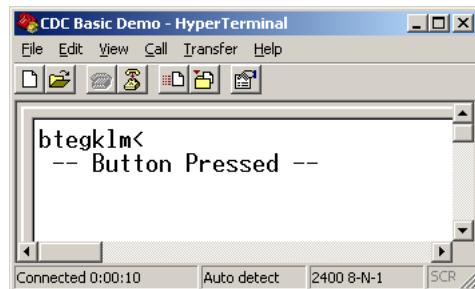
For more information about each demo board, please refer to the Demo Board Information section.

Demo Operation

This demo allows the device to appear like a serial (COM) port to the host. In order to run this demo first compile and program the target device. Please see the following Windows, Linux, and Macintosh sections for how to connect to the device on each of these systems.

Once connected to the device, there are two ways to run this example project. Typing a key in the terminal window will result in the device echoing that key plus one. So if the user presses "a", the device will echo "b". If the pushbutton is pressed the device will echo " – Button Pressed – " to the terminal window.

Note: Some terminal programs, like hyperterminal, require users to click the disconnect button before removing the device from the computer. Failing to do so may result in having to close and open the program again in order to reconnect to the device.



1.6.4.1 Windows

Attach the device to the host. If the host is a PC and this is the first time you have plugged this device into the computer then you may be asked for a .inf file.



Select the “Install from a list or specific location (Advanced)” option. Point to the “<Install Directory>\USB Device - CDC – Basic Demo\inf\win2k_winxp” directory.



Once the device is successfully installed, open up a terminal program, such as hyperterminal. Select the appropriate COM port. On most machines this will be COM5 or higher.

1.6.4.2 Linux

Upon plugging in a USB CDC ACM virtual COM port device into a Linux machine, the OS will automatically enumerate the USB device successfully, and a new object should show up as:

/dev/ttyACMx

(where ttyACMx is usually ttyACM0, but could be some other number such as ttyACM1, if some other ACM device is already attached to the machine).

To determine the exact number value of “x”, a procedure like follows can be used:

1. Open a console.
2. Make sure the USB device has been plugged into the machine.
3. Type: lsusb
4. You should see a line like: Bus 005 Device 004: ID 04d8:000a Microchip Technology, Inc.
5. Type: modprobe cdc-acm vendor=0x04d8 product=0x000a
6. Type: dmesg
7. You should get the status, showing the ttyACMx value, ex: cdc_acm 5-1:1.0: ttyACM0: USB ACM device

Once you know the ttyACMx value, applications and terminal programs (such as GtkTerm) can interface with the USB serial port by configuring them to connect up to the /dev/ttyACMx object.

1.6.4.3 Macintosh

Upon plugging in a USB CDC ACM virtual COM port device into a Mac OS X based machine, the OS should automatically enumerate the USB device successfully, and a new object should show up as:

/dev/tty.usbmodemXXXX

(where XXXX is some value, such as "3d11")

To run the example demo project: "USB\Device - CDC - Basic Demo" on a Mac OS X based machine, a procedure like follows can be used:

Open TERMINAL. This can be done by clicking SPOTLIGHT and searching for TERMINAL. Spotlight is the little magnifying glass in the upper right of the screen.

In Terminal, with the USB CDC ACM device NOT plugged in (yet), type:

ls /dev/tty.*

This will show all serial devices currently connected to the Mac. In the author's case, the following list appears:

/dev/tty.Bluetooth-Modem

/dev/tty.Bluetooth-PDA-Sync

/dev/tty.Rob-1

Now, plug the USB CDC device into a USB port of the Mac. Hit the UP cursor, which will bring the search command back (ls /dev/tty.*) and hit return. You should get the exact same list as before, but this time, with a new serial device. In the author's case, it was:

/dev/tty.usbmodem3d11

Once the complete name is known, the received serial port data can be displayed by typing:

screen /dev/tty.usbmodem3d11

(replace "3d11" in the above line with the value for your machine). If the microcontroller was programmed with the "USB\Device - CDC - Basic Demo", you can then press the user pushbutton, and the standard demo text should be printed to the screen (ex: "BUTTON PRESSED ---").

If the USB device is being operated as a USB to UART translator device (ex: using "USB\Device - CDC - Serial Emulator" firmware, the baud rate can be set by using syntax like follows:

screen -U /dev/tty.usbmodem3d11 38400

Where "usbmodem3d11" should be replaced with the actual value of the device, and "38400" should be replaced with actual desired baud rate (ex: 9600, 19200, 38400, 57600, 115200, etc.). More details and usage information for screen can be found in the man page.

Note: Composite CDC + (any other interface) USB devices (such as the MCP2200, which is a composite CDC+HID device)

will only work on Mac OS X 10.7 (or later). Mac OS X 10.7 is the first OS X version that supports USB Interface Association Descriptors (IADs), which are needed when implementing composite USB devices with multiple interfaces, with at least one CDC-ACM function. Prior versions of Mac OS X did not support IADs, and therefore can only support non-composite, single function CDC-ACM devices.

1.6.5 Device - HID - Custom Demo

Demo showing how to create a device that can transfer custom application data without the need of a driver installation using the HID class.

Description

Supported Demo Boards

The matrix of which demos are supported on a specific board can be found in the Release Notes demo board support section. Verify that the board you wish to use will work with this demo. This table also describes some of the limitations that the board might have while running this demo.

Demo Board I/O Mapping

Each demo board has a different number of push buttons, LEDs, and other features with various different names for these components. To determine which board features are used for which demo features, please refer to the io_mapping.h file in the demo folder under the system_config folder. Each demo board will have a corresponding folder with an io_mapping.h file in it. For example, for the PIC18F46J50 PIM this would be the following file:

```
<install_directory>/apps/usb/device/hid_custom/firmware/src/system_config/pic18f46j50_pim/io_mapping.h
```

For more information about each demo board, please refer to the Demo Board Information section.

Demo Operation

This demo uses the selected hardware platform as a HID class USB device, but uses the HID class for general purpose I/O operations. Typically, the HID class is used to implement human interface products, such as mice and keyboards. The HID protocol is however quite flexible, and can be adapted and used to send/receive general purpose data to/from a USB device. Using the HID class for general purpose I/O operations is quite advantageous, in that it does not require any kind of custom driver installation process. HID class drivers are already provided by and are distributed with common operating systems. Therefore, upon plugging in a HID class device into a typical computer system, no user installation of drivers is required, the installation is fully automatic.

HID devices primarily communicate through one interrupt IN endpoint and one interrupt OUT endpoint. In most applications, this effectively limits the maximum achievable bandwidth for full speed HID devices to 64kBytes/s of IN traffic, and 64kBytes/s of OUT traffic (64kB/s, but effectively “full duplex”).

The GenericHIDSimpleDemo.exe program, and the associated firmware demonstrate how to use the HID protocol for basic general purpose USB data transfer. To make the PC source code as easy to understand as possible, the demo has deliberately been made simple, and only sends/receives small amounts of data.

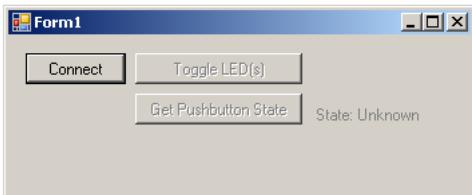
Before you can run the GenericHIDSimpleDemo.exe executable, you will need to have the Microsoft® .NET Framework Version 2.0 Redistributable Package (later versions probably okay, but not tested) installed on your computer. Programs which were built in the Visual Studio® .NET languages require the .NET redistributable package in order to run. The redistributable package can be freely downloaded from Microsoft’s website. Users of Windows Vista® operating systems will not need to install the .NET framework, as it comes pre-installed as part of the operating system.

The source code for GenericHIDSimpleDemo.exe file was created in Microsoft Visual C++® 2005 Express Edition. The source code can be found in the “<Install Directory>\ USB Device - HID - Custom Demos\Generic HID - Simple Demo - PC Software” directory. Microsoft currently distributes Visual C++ 2005 Express Edition for free, and can be downloaded from Microsoft’s website. When downloading Microsoft Visual C++ 2005 Express Edition, also make sure to download and install the Platform SDK, and follow Microsoft’s instructions for integrating it with the development environment.

It is not necessary to install either Microsoft Visual C++ 2005, or the Platform SDK in order to begin using the

GenericHIDSimpleDemo.exe program. These are only required if the source code will be modified or compiled.

To launch the application, simply double click on the executable “GenericHIDSimpleDemo.exe” in the “<Install Directory>\USB Device - HID - Custom Demos” directory. A window like that shown below should appear:



If instead of this window, an error message pops up while trying to launch the application, it is likely the Microsoft .NET Framework Version 2.0 Redistributable Package has not yet been installed. Please install it and try again.

In order to begin sending/receiving packets to the device, you must first find and “connect” to the device. As configured by default, the application is looking for HID class USB devices with VID = 0x04D8 and PID = 0x003F. The device descriptor in the firmware project meant to be used with this demo uses the same VID/PID. If you plug in a USB device programmed with the correct precompiled .hex file, and hit the “Connect” button, the other pushbuttons should become enabled. If hitting the connect button has no effect, it is likely the USB device is either not connected, or has not been programmed with the correct firmware.

Hitting the Toggle LED(s) should send a single packet of general purpose generic data to the HID class USB peripheral device. The data will arrive on the interrupt OUT endpoint. The firmware has been configured to receive this generic data packet, parse the packet looking for the “Toggle LED(s)” command, and should respond appropriately by controlling the LED(s) on the demo board.

The “Get Pushbutton State” button will send one packet of data over the USB to the peripheral device (to the interrupt OUT endpoint) requesting the current pushbutton state. The firmware will process the received Get Pushbutton State command, and will prepare an appropriate response packet depending upon the pushbutton state.

The PC then requests a packet of data from the device (which will be taken from the interrupt IN endpoint). Once the PC application receives the response packet, it will update the pushbutton state label.

Try experimenting with the application by holding down the appropriate pushbutton on the demo board, and then simultaneously clicking on the “Get Pushbutton State” button. Then try to repeat the process, but this time without holding down the pushbutton on the demo board.

To make for a more fluid and gratifying end user experience, a real USB application would probably want to launch a separate thread to periodically poll the pushbutton state, so as to get updates regularly. This is not done in this simple demo, so as to avoid cluttering the PC application project with source code that is not related to USB communication.

Running the demo on an Android v3.1+ device

There are two main ways to get the example application on to the target Android device: the Android Market and by compiling the source code.

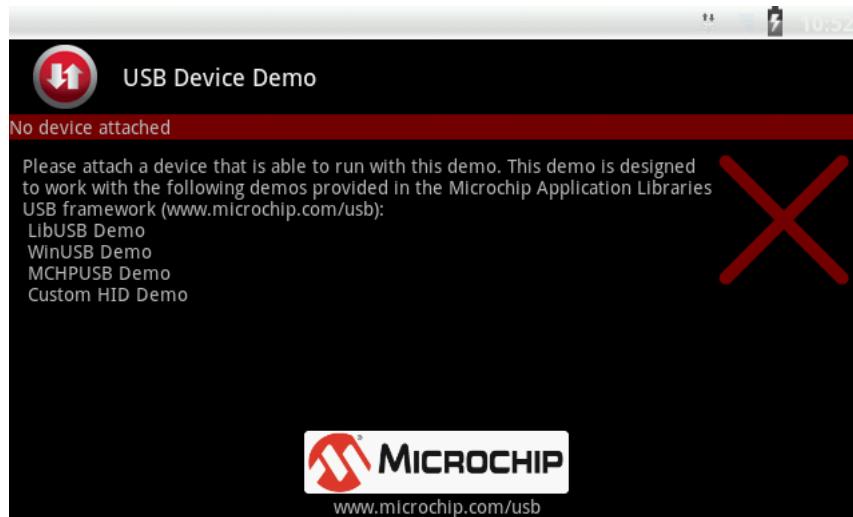
1. The demo application can be downloaded from Microchip's Android Marketplace page:
<https://market.android.com/developer?pub=Microchip+Technology+Inc>



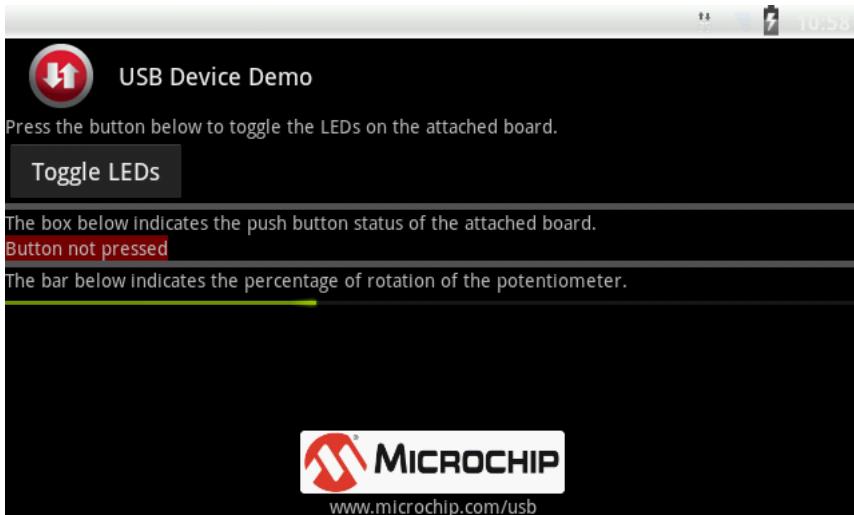
2. The source code for this demo is also provided in the demo project folder. For more information about how to build and load Android applications, please refer to the following pages:

- <http://developer.android.com/index.html>
- <http://developer.android.com/sdk/index.html>
- <http://developer.android.com/sdk/installing.html>

While there are no devices attached, the Android application will indicate that no devices are attached.



When the device is attached, the an alternative screen will allow various control/status features with the hardware on the board.



1.6.6 Device - HID - Digitizer Demos

These are examples of HID digitizers. There are single, and various multi-point touch examples.

Description

Supported Demo Boards

The matrix of which demos are supported on a specific board can be found in the Release Notes demo board support section. Verify that the board you wish to use will work with this demo. This table also describes some of the limitations that the board might have while running this demo.

Demo Board I/O Mapping

Each demo board has a different number of push buttons, LEDs, and other features with various different names for these components. To determine which board features are used for which demo features, please refer to the `io_mapping.h` file in the demo folder under the `system_config` folder. Each demo board will have a corresponding folder with an `io_mapping.h` file in it. For example, for the PIC18F46J50 PIM this would be the following file:

```
<install_directory>/apps/usb/device/hid_digitizer/firmware/src/system_config/pic18f46j50_pim/io_mapping.h
```

For more information about each demo board, please refer to the Demo Board Information section.

Demo Operation

These demos use the selected hardware platform as a USB HID class digitizer device. The Single-Touch demo is a HID class pen digitizer demo, which emulates a pen digitizer touch screen capable of sensing a single contact point. The Multi-Touch demo emulates a touch sensitive touch screen, capable of sensing two simultaneous contact points. The multi-touch demo can potentially be expanded to support additional simultaneous contacts (by modifying the HID report descriptor), however, the standard built in gestures that are recognized by the Microsoft Windows 7 platform only use one or two contacts.

To use the Single-Touch pen digitizer demo, plug the demo board into a free USB port on a Windows Vista or Windows 7 machine. The device should automatically enumerate as a HID class pen digitizer device, and certain additional functions and capabilities built into the operating system will become activated. No manual USB driver installation is necessary, as the built in HID class drivers are used for this device.

To use the Multi-Touch digitizer demo, plug the demo board into a free USB port on a Windows 7 machine. Windows 7 has significantly more "Windows Touch" capabilities than Vista. Although the device will enumerate and provide limited functionality on Windows Vista, multi-touch gestures will not be recognized unless run on Windows 7.

Since the standard demo boards that these demos are meant to be run on do not have an actual touch sensitive contact

area, the firmware demos emulate the data that would be generated by a real touch screen. Both demo projects use a single user pushbutton. By pressing the button, the firmware will send a flurry of USB packets to the host, which contain contact position data that is meant to mimic an actual “gesture” of various types. Each subsequent press of the pushbutton will advance the internal state machine, and cause the firmware to send a gesture to the PC.

To use the demos, it is best to have Microsoft Internet Explorer installed on the machine (although some demo functions can be observed using the pen flick practice area available from the control panel). The latest versions of Internet Explorer (when run on the proper OS: preferably Windows 7, but some function on Windows Vista) supports recognition and use of certain basic gestures, such as “back”, “forward”, as well as certain scroll and zoom operations.

Other Info: Windows 7 adds support for Windows messages such as “WM_GESTURE” and “WM_TOUCH”. These messages can be used to help build customized “touch enabled” PC applications. Documentation for these messages can be found in MSDN.

The following Microsoft developer blog contains useful additional information relating to Windows Touch:

<http://blogs.msdn.com/e7/archive/2009/03/25/touching-windows-7.aspx>

1.6.7 Device - HID - Joystick Demo

This demo shows how to create a USB joystick.

Description

Supported Demo Boards

The matrix of which demos are supported on a specific board can be found in the Release Notes demo board support section. Verify that the board you wish to use will work with this demo. This table also describes some of the limitations that the board might have while running this demo.

Demo Board I/O Mapping

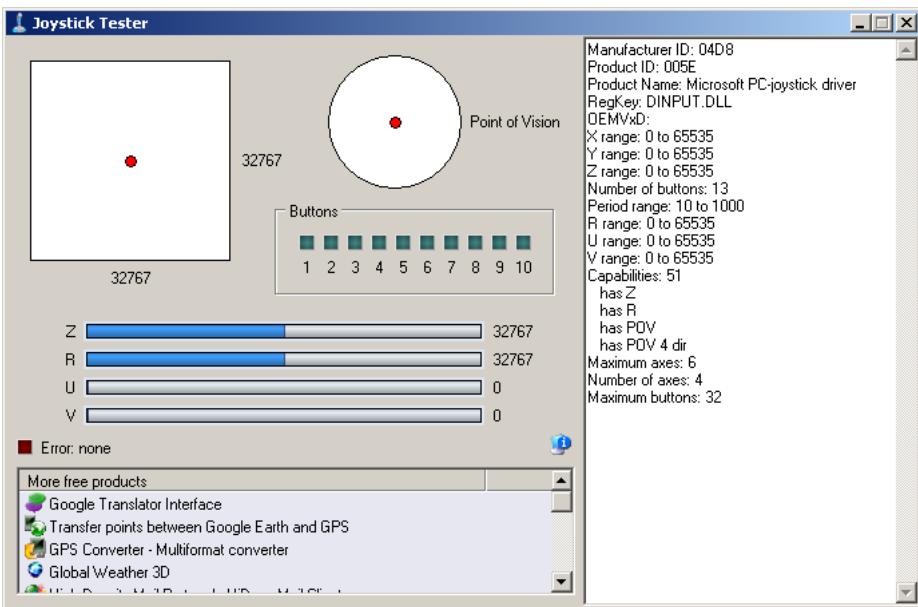
Each demo board has a different number of push buttons, LEDs, and other features with various different names for these components. To determine which board features are used for which demo features, please refer to the io_mapping.h file in the demo folder under the system_config folder. Each demo board will have a corresponding folder with an io_mapping.h file in it. For example, for the PIC18F46J50 PIM this would be the following file:

```
<install_directory>/apps/usb/device/hid_joystick/firmware/src/system_config/pic18f46j50_pim/io_mapping.h
```

For more information about each demo board, please refer to the Demo Board Information section.

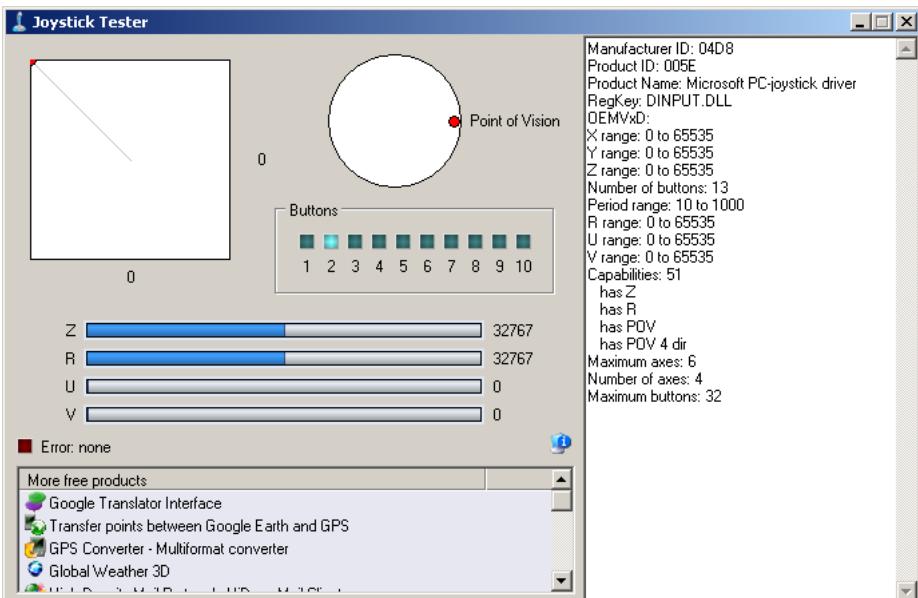
Demo Operation

This demo uses the selected hardware platform as a USB Joystick. To test the joystick feature, open the JoystickTester.exe in the demo project folder. This will launch a window as seen below:



Pressing the button will cause the device to:

- Indicate that the "x" button is pressed, but none others;
- Move the hat switch to the "east" position;
- Move the X and Y coordinates to their extreme values;



1.6.8 Device - CDC - Serial Emulator

This demo shows how to use the CDC class to create a USB to UART bridge device. This demo is very similar to the Device - CDC Basic Demo, with the following differences:

- The CDC serial emulator firmware enables the hardware UART of the microcontroller and forwards any received UART RX bytes to the host terminal program through the virtual COMx port. Similarly, any bytes received from the host terminal program over the COMx port get forwarded out the hardware UART TX microcontroller pin. This provides a bi-directional USB to UART bridge functionality.

2. The CDC serial emulator firmware uses SET_LINE_CODING information to update the hardware UART baud rate setting. The host automatically sends one or more SET_LINE_CODING requests upon the user opening the COMx port or changing the COMx port baud rate (or other data encoding settings) from the terminal program. This allows the host software to control the hardware UART's baud rate value.

The CDC serial emulator demo uses the same USB drivers that are needed for the CDC Basic Demo. Upon plugging in a USB CDC serial emulator device, and installing any necessary drivers (for Windows, for Mac/Linux OSes, the drivers are normally automatically installed and come with the OS), the device will produce a COMx port (on Windows, on Mac a /dev/tty.usbmodemXXXX object, and on a Linux machine a /dev/ttymACMx object) that PC applications such as serial terminal programs can connect to.

See the Device - CDC Basic Demo information for more details.

1.6.9 Device - HID - Keyboard Demo

This example shows how to create a USB keyboard and how to send data to the host.

Description

Supported Demo Boards

The matrix of which demos are supported on a specific board can be found in the Release Notes demo board support section. Verify that the board you wish to use will work with this demo. This table also describes some of the limitations that the board might have while running this demo.

Demo Board I/O Mapping

Each demo board has a different number of push buttons, LEDs, and other features with various different names for these components. To determine which board features are used for which demo features, please refer to the io_mapping.h file in the demo folder under the system_config folder. Each demo board will have a corresponding folder with an io_mapping.h file in it. For example, for the PIC18F46J50 PIM this would be the following file:

```
<install_directory>/apps/usb/device/hid_keyboard/firmware/src/system_config/pic18f46j50_pim/io_mapping.h
```

For more information about each demo board, please refer to the Demo Board Information section.

Demo Operation

This demo uses the selected hardware platform as a USB keyboard. Before pressing the button, select a window in which it is safe to type text freely. Pressing the button will cause the device to print a character on the screen. The characters will print a new letter/number for each press. If a key is held, it will emulate as if the key was held on a keyboard. Pressing the CapsLock button on the host PC will cause an LED to light on the board.

1.6.10 Device - HID - Mouse Demo

This demo is a simple mouse demo that causes the mouse to move in a circle on the screen.

Description

Supported Demo Boards

The matrix of which demos are supported on a specific board can be found in the Release Notes demo board support section. Verify that the board you wish to use will work with this demo. This table also describes some of the limitations that the board might have while running this demo.

Demo Board I/O Mapping

Each demo board has a different number of push buttons, LEDs, and other features with various different names for these

components. To determine which board features are used for which demo features, please refer to the io_mapping.h file in the demo folder under the system_config folder. Each demo board will have a corresponding folder with an io_mapping.h file in it. For example, for the PIC18F46J50 PIM this would be the following file:

```
<install_directory>/apps/usb/device/hid_mouse/firmware/src/system_config/pic18f46j50_pim/io_mapping.h
```

For more information about each demo board, please refer to the Demo Board Information section.

Demo Operation

This demo uses the selected hardware platform as a USB mouse. Before connecting the board to the computer through the USB cable please be aware that the device will start moving the mouse cursor around on the computer. There are two ways to stop the device from making the cursor to continue to move. The first way is to disconnect the device from the computer. The second is to press the correct button on the hardware platform. Pressing the button again will cause the mouse cursor to start moving in a circle again.

1.6.11 Device - HID - Uninterruptible Power Supply

This demo shows how to create a Uninterruptible Power Supply (UPS) device.

Description

Supported Demo Boards

The matrix of which demos are supported on a specific board can be found in the Release Notes demo board support section. Verify that the board you wish to use will work with this demo. This table also describes some of the limitations that the board might have while running this demo.

Demo Board I/O Mapping

Each demo board has a different number of push buttons, LEDs, and other features with various different names for these components. To determine which board features are used for which demo features, please refer to the io_mapping.h file in the demo folder under the system_config folder. Each demo board will have a corresponding folder with an io_mapping.h file in it. For example, for the PIC18F46J50 PIM this would be the following file:

```
<install_directory>/apps/usb/device/hid_ups/firmware/src/system_config/pic18f46j50_pim/io_mapping.h
```

For more information about each demo board, please refer to the Demo Board Information section.

Demo Operation

This demo uses the selected hardware platform as a HID class USB Uninterruptible power supply (UPS). When the device is plugged into a computer, the computer should have an indicator showing that it is connected to a UPS and it should show a charge percentage of the battery of the UPS. This demo uses a fixed time derived from the USB start of frame (SOF) packets to emulate the battery charging by sending updates about the battery status to the computer.

Holding the specified button on the demo board puts the UPS in a emulated discharge state, as if the main power has been removed/failed. As time progresses the board sends updated information about the charge left on the battery. As the battery approaches the minimum threshold, the UPS will send a command to shut down the computer. Release the button at any point of time to simulate a reconnection of the main power supply and to emulate the UPS returning to a charging state.

1.6.12 Device - Mass Storage - Internal Flash Demo

This demo uses the selected hardware platform as an drive on the computer using the internal flash of the device as the drive storage media.

Description

Supported Demo Boards

The matrix of which demos are supported on a specific board can be found in the Release Notes demo board support section. Verify that the board you wish to use will work with this demo. This table also describes some of the limitations that the board might have while running this demo.

Demo Board I/O Mapping

Each demo board has a different number of push buttons, LEDs, and other features with various different names for these components. To determine which board features are used for which demo features, please refer to the io_mapping.h file in the demo folder under the system_config folder. Each demo board will have a corresponding folder with an io_mapping.h file in it. For example, for the PIC18F46J50 PIM this would be the following file:

```
<install_directory>/apps/usb/device/msd_internal_flash/firmware/src/system_config/pic18f46j50_pim/io_mapping.h
```

For more information about each demo board, please refer to the Demo Board Information section.

Demo Operation

This demo uses the selected hardware platform as an drive on the computer using the internal flash of the device as the drive storage media. Connect the hardware platform to a computer through a USB cable.

The device should appear as a new drive on the computer named “Drive Name”. The volume label or file information can be changed in the Files.c file located in the project directory.

1.6.12.1 Troubleshooting

Issue 1: The device appears correctly in the device manager, but no new drive letters appear on a Windows® operating system based machine.

Solution: See Microsoft knowledge base article 297694: <http://support.microsoft.com/kb/297694>

If there is a drive letter conflict (ex: because a network drive has been mapped to a letter low in the alphabet), on some operating systems the newly attached USB drive may not appear. If this occurs, either obtain the hotfix from Microsoft, or remap the conflicting mapped network drive to a letter at the end of the alphabet (ex: Z:).

Issue 2: The device enumerates correctly and I can access the new drive. Even though the drive is not full yet, when I try to write to the drive, I get an error message something like, “Cannot copy (some name): The directory or file cannot be created.”

Solution: In order to copy new files onto the drive volume, both the file contents themselves must be copied to the drive, and the FAT table must also be updated in order to accommodate the new file name and path. Even if the drive has plenty of free space available, the FAT table may have reached its limit. In order to keep the default demos small, the FAT table is configured to be only 512 bytes long. This is not very large, and can easily be exceeded, especially if the files on the drive have long file names. In order to use the remaining space available on the drive, it is recommended to keep the individual file names as short as possible to minimize their size in the FAT table. Alternatively, the firmware can be modified so that the FAT table is larger, and therefore able to accommodate more file name and path characters.

Issue 3: When I try to format the drive, I get an error message and the drive does not get formatted properly.

Solution: By default, common Windows based operating systems will try to place a large FAT table on the newly formatted disk (larger than the default 512 bytes of the demo firmware). If the FAT table is larger than the total drive space, the drive cannot be formatted. In order to successfully format the drive, an alternative method of formatting will be needed that places a smaller FAT table on the drive. For example, the drive can be effectively reformatted by reprogramming the microcontroller with the original HEX file. Alternatively, if the firmware is modified to increase the total drive space, the Windows operating system managed FAT table may be able to fit. Unfortunately, this will shrink the effective drive size, making less of it available for actual file data.

Issue 4: When I format the drive, the drive size shrinks.

Solution: See the solution to issue #3 above.

1.6.13 Device - Mass Storage - SD Card Reader

This demo shows how to implement a simple SD card reader

Description

Supported Demo Boards

The matrix of which demos are supported on a specific board can be found in the Release Notes demo board support section. Verify that the board you wish to use will work with this demo. This table also describes some of the limitations that the board might have while running this demo.

Demo Board I/O Mapping

Each demo board has a different number of push buttons, LEDs, and other features with various different names for these components. To determine which board features are used for which demo features, please refer to the io_mapping.h file in the demo folder under the system_config folder. Each demo board will have a corresponding folder with an io_mapping.h file in it. For example, for the PIC18F46J50 PIM this would be the following file:

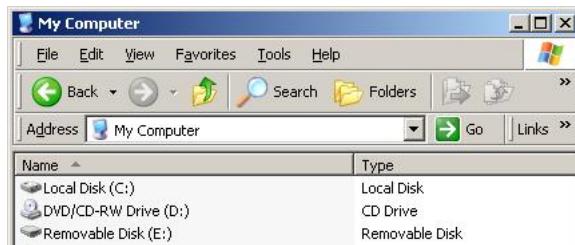
```
<install_directory>/apps/usb/device/msd_sd_card_reader/firmware/src/system_config/pic18f46j50_pim/io_mapping.h
```

For more information about each demo board, please refer to the Demo Board Information section.

Demo Operation

Connect the hardware platform to a computer through a USB cable. If the device was attached to the computer while the data logging occurred, you may need to remove the SD card from the card slot or disconnect and reconnect the device from the computer for the files to appear. Most computers are not expecting the files on an attached drive to change if they are not making the change so some operating systems will not look for additional drive changes.

The device should appear as a new drive on the computer named "Removable Drive".



If no SD Card is inserted in the SD Card PICTail Plus, the following dialog will pop-up.



Once a compatible card is inserted in the card reader, files can be read, deleted, and manipulated like any other drive on the computer.

1.6.14 Device - Vendor Driver Basic Demo

This demo creates a simple vendor class device using the libusb and WinUSB drivers. It includes PC/host software examples as well.

Description

Supported Demo Boards

The matrix of which demos are supported on a specific board can be found in the Release Notes demo board support section. Verify that the board you wish to use will work with this demo. This table also describes some of the limitations that the board might have while running this demo.

Demo Board I/O Mapping

Each demo board has a different number of push buttons, LEDs, and other features with various different names for these components. To determine which board features are used for which demo features, please refer to the io_mapping.h file in the demo folder under the system_config folder. Each demo board will have a corresponding folder with an io_mapping.h file in it. For example, for the PIC18F46J50 PIM this would be the following file:

```
<install_directory>/apps/usb/device/vendor_basic/firmware/src/system_config/pic18f46j50_pim/io_mapping.h
```

For more information about each demo board, please refer to the Demo Board Information section.

Demo Operation

1.6.14.1 Windows

Running the demo on a Windows machine.

Description

This demo uses the selected hardware platform as a Libusb class USB device. Libusb-Win32 is a USB Library for the Windows operating systems. The library allows user space applications to access any USB device on Windows in a generic way without writing any line of kernel driver code. This driver allows users to have access to interrupt, bulk, and control transfers directly.

The SimpleLibUSBDemo.exe program and the associated firmware demonstrate how to use the Libusb device drivers for basic general purpose USB data transfer. To make the PC source code as easy to understand as possible, the demo has deliberately been made simple, and only sends/receives small amounts of data.

Before you can run the SimpleLibUSBDemo.exe executable, you will need to have the Microsoft® .NET Framework Version 3.5 Redistributable Package (later versions probably okay, but not tested) installed on your computer. Programs which were built in the Visual Studio® .NET languages require the .NET redistributable package in order to run. The redistributable package can be freely downloaded from Microsoft's website. Users of Windows Vista® operating systems will not need to install the .NET framework, as it comes pre-installed as part of the operating system.

The source code for SimpleLibUSBDemo.exe file was created in Microsoft Visual C++® 2008 Express Edition. The source code can be found in the “<Install Directory>\ USB Device - Libusb - Generic Driver Demo\ Libusb Simple Demo - Windows Application\Libusb Simple Demo - PC Application - MS VC++ 2008 Express” directory. Microsoft currently distributes Visual C++ 2008 Express Edition for free, and can be downloaded from Microsoft's website.

To launch the application, simply double click on the executable “SimpleLibusbDemo.exe” in the “<Install Directory>\USB Device - Libusb - Generic Driver Demo\Windows Application” directory. A window like that shown below should appear:



If instead of this window, an error message pops up while trying to launch the application, it is likely the Microsoft .NET Framework Version 3.5 Redistributable Package has not yet been installed. Please install it and try again.

In order to begin sending/receiving packets to the device, you must first find and “connect” to the device. As configured by

default, the application is looking for USB devices with VID = 0x04D8 and PID = 0x0204. The device descriptor in the firmware project meant to be used with this demo uses the same VID/PID. To run the demo program the USB device with the correct precompiled .hex file. If you are connecting the device for the first time, Windows pops up a window asking you to install the driver for the device. When asked for the driver point it to the inf file provided along with the demo. Windows takes while to install the driver for the USB device that is just plugged in. Open the Device manager and ensure that the USB device is listed under the 'Libusb Demo Devices'. Once the driver is installed hit the "Connect" button, the other pushbuttons should become enabled. If hitting the connect button has no effect, it is likely the USB device is either not connected, or has not been programmed with the correct firmware.

If a different VID/PID combination from the default is desired, then the descriptors in the firmware must be changed as well as the inf file. The easiest way to change the inf file is to use the utility provided with the LibUSB download for windows on the LibUSB [website](#). This utility can create a new inf file based on a connected device. So make sure to change the VID/PID combination first in the firmware, connect the device, and then run the inf file creator utility. After completing the utility, a new signed driver with inf file is created.

Once the driver is installed hit the "Connect" button, the other pushbuttons should become enabled. If hitting the connect button has no effect, it is likely the USB device is either not connected, or has not been programmed with the correct firmware.

Hitting the Toggle LED(s) should send a single packet of general purpose generic data to the Custom class USB peripheral device. The data will arrive on the Bulk OUT endpoint. The firmware has been configured to receive this generic data packet, parse the packet looking for the "Toggle LED(s)" command, and should respond appropriately by controlling the LED(s) on the demo board.

The "Get Pushbutton State" button will send one packet of data over the USB to the peripheral device (to the Bulk OUT endpoint) requesting the current pushbutton state. The firmware will process the received Get Pushbutton State command, and will prepare an appropriate response packet depending upon the pushbutton state.

The PC then requests a packet of data from the device (which will be taken from the Bulk IN endpoint). Once the PC application receives the response packet, it will update the pushbutton state label.

Try experimenting with the application by holding down the appropriate pushbutton on the demo board, and then simultaneously clicking on the "Get Pushbutton State" button. Then try to repeat the process, but this time without holding down the pushbutton on the demo board.

To make for a more fluid and gratifying end user experience, a real USB application would probably want to launch a separate thread to periodically poll the pushbutton state, so as to get updates regularly. This is not done in this simple demo, so as to avoid cluttering the PC application project with source code that is not related to USB communication.

In order to build the application, copy the file <libusb-win32 unzipped folder>\libusb-win32-device-bin-0.1.12.1\lib\msvc\libusb.lib and paste to 'lib' folder of the VC++. Also copy the file <libusb-win32 unzipped folder>\libusb-win32-device-bin-0.1.12.1\include\usb.h and paste to the "<Install Directory>\USB Device - Libusb - Generic Driver Demo\Windows Application\Microsoft VC++ 2008 Express\SimpleLibusbDemo' folder.

1.6.14.2 Linux

Running the demo on a Linux machine.

Description

The SimpleLibUSBDemo program and the associated firmware demonstrate how to use the Libusb device drivers for basic general purpose USB data transfer. To make the PC source code as easy to understand as possible, the demo has deliberately been made simple, and only sends/receives small amounts of data.

Before you can run the SimpleLibUSBDemo executable, you will need to have the libusb 0.1 driver installed on your computer. The libusb can be downloaded from [sourceforge.net](#).

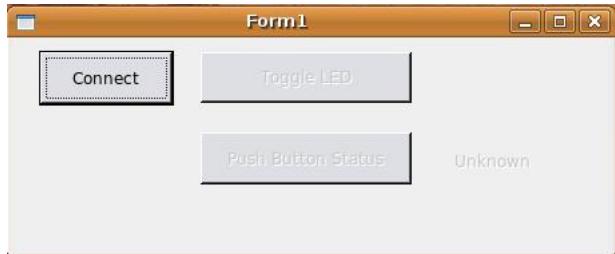
The source code for SimpleLibUSBDemo.exe file was created using QT3 Designer. The source code can be found in the

“<Install Directory>\ USB Device - Libusb - Generic Driver Demo\Libusb Simple Demo - Linux Application\ Libusb Simple Demo - Linux Application -QT3” directory.

To launch the application, open the Terminal and navigate to the “<Install Directory>\USB Device - LibUSB - Generic Driver Demo\Linux Application” directory and execute the following commands

1. chmod a+x SimpleLibusbDemo_Linux (This command gives executable right to the file on this Linux computer)
2. sudo ./SimpleLibusbDemo_Linux.

Enter the Super user password when requested. A window like that shown below should appear:



In order to begin sending/receiving packets to the device, you must first find and “connect” to the device. As configured by default, the application is looking for USB devices with VID = 0x04D8 and PID = 0x0204. The device descriptor in the firmware project meant to be used with this demo uses the same VID/PID. To run the demo program the USB device with the correct precompiled .hex file. If you are connecting the device for the first time, Windows pops up a window asking you to install the driver for the device. When asked for the driver point it to the inf file provided along with the demo. Windows takes while to install the driver for the USB device that is just plugged in. Open the Device manager and ensure that the USB device is listed under the ‘Libusb Demo Devices’. Once the driver is installed hit the “Connect” button, the other pushbuttons should become enabled. If hitting the connect button has no effect, it is likely the USB device is either not connected, or has not been programmed with the correct firmware.

Hitting the Toggle LED(s) should send a single packet of general purpose generic data to the Custom class USB peripheral device. The data will arrive on the Bulk OUT endpoint. The firmware has been configured to receive this generic data packet, parse the packet looking for the “Toggle LED(s)” command, and should respond appropriately by controlling the LED(s) on the demo board.

The “Get Pushbutton State” button will send one packet of data over the USB to the peripheral device (to the Bulk OUT endpoint) requesting the current pushbutton state. The firmware will process the received Get Pushbutton State command, and will prepare an appropriate response packet depending upon the pushbutton state.

The PC then requests a packet of data from the device (which will be taken from the Bulk IN endpoint). Once the PC application receives the response packet, it will update the pushbutton state label.

Try experimenting with the application by holding down the appropriate pushbutton on the demo board, and then simultaneously clicking on the “Get Pushbutton State” button. Then try to repeat the process, but this time without holding down the pushbutton on the demo board.

To make for a more fluid and gratifying end user experience, a real USB application would probably want to launch a separate thread to periodically poll the pushbutton state, so as to get updates regularly. This is not done in this simple demo, so as to avoid cluttering the PC application project with source code that is not related to USB communication.

In order to build the application navigate to the “<Install Directory>\USB Device - LibUSB - Generic Driver Demo\Linux Application\Qt3” directory and execute the command “make”.

1.6.14.3 Android 3.1+

Running the demo on an Android device.

Description

There are two main ways to get the example application on to the target Android device: the Android Market and by compiling the source code.

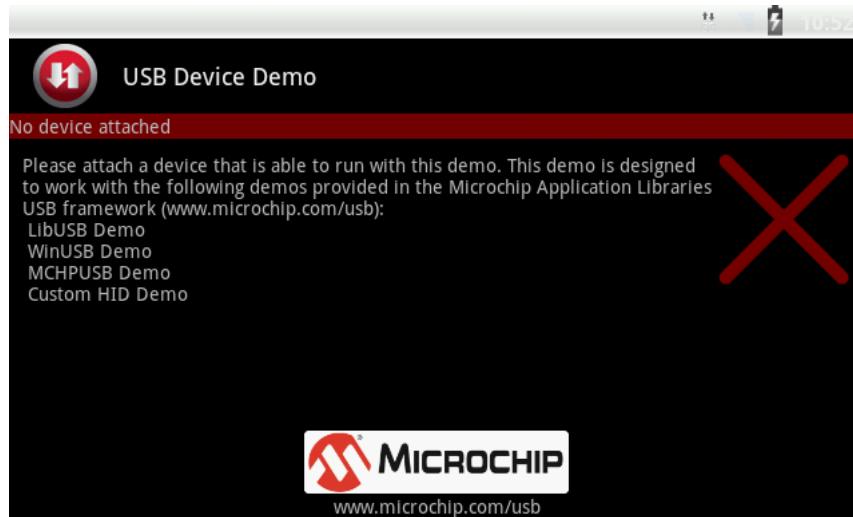
1. The demo application can be downloaded from Microchip's Android Marketplace page:
<https://market.android.com/developer?pub=Microchip+Technology+Inc>



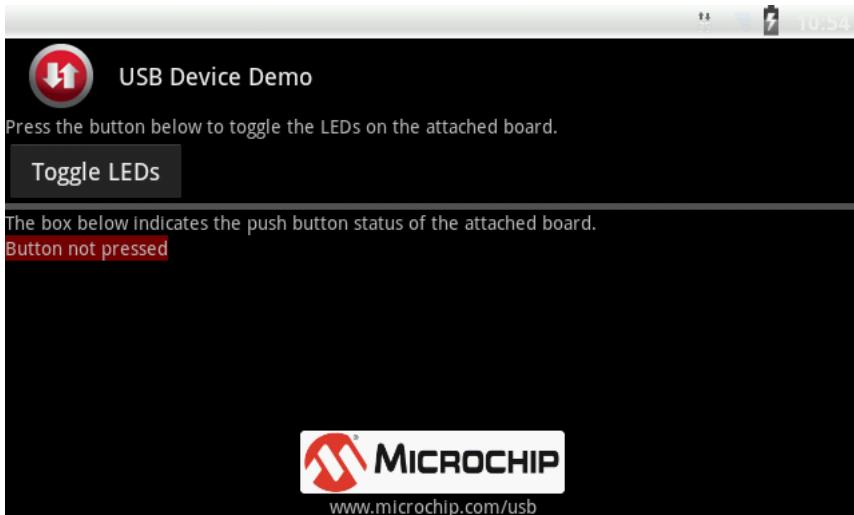
2. The source code for this demo is also provided in the demo project folder. For more information about how to build and load Android applications, please refer to the following pages:

- <http://developer.android.com/index.html>
- <http://developer.android.com/sdk/index.html>
- <http://developer.android.com/sdk/installing.html>

While there are no devices attached, the Android application will indicate that no devices are attached.



When the device is attached, the an alternative screen will allow various control/status features with the hardware on the board.



1.6.15 Device - Vendor High Bandwidth Demo

This demo shows how to get (and measures) the maximum throughput using a vendor class driver.

Description

Supported Demo Boards

The matrix of which demos are supported on a specific board can be found in the Release Notes demo board support section. Verify that the board you wish to use will work with this demo. This table also describes some of the limitations that the board might have while running this demo.

Demo Board I/O Mapping

Each demo board has a different number of push buttons, LEDs, and other features with various different names for these components. To determine which board features are used for which demo features, please refer to the io_mapping.h file in the demo folder under the system_config folder. Each demo board will have a corresponding folder with an io_mapping.h file in it. For example, for the PIC18F46J50 PIM this would be the following file:

```
<install_directory>/apps/usb/device/vendor_throughput_test/firmware/src/system_config/pic18f46j50_pim/io_mapping.h
```

For more information about each demo board, please refer to the Demo Board Information section.

Demo Operation

This demo uses the selected hardware platform as a WinUSB class USB device. WinUSB is a vendor specific driver produced by Microsoft for use with Windows® XP service pack 2 and later operating systems. This driver allows users to have access to interrupt, bulk, and control transfers directly.

The HighBandwidthWinUSB.exe program, and the associated firmware demonstrate how to use the WinUSB device drivers for USB Bulk data transfers. Total Time taken to transmit the data & data transmission rate (Bytes/Sec) is shown in the GUI once the data transmission of 9,60,000 bytes is completed from the PC side.

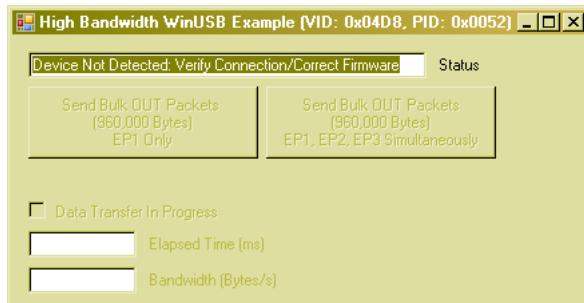
Before you can run the HighBandwidthWinUSB.exe executable, you will need to have the Microsoft® .NET Framework Version 2.0 Redistributable Package (later versions probably okay, but not tested) installed on your computer. Programs which were built in the Visual Studio® .NET languages require the .NET redistributable package in order to run. The redistributable package can be freely downloaded from Microsoft's website. Users of Windows Vista® operating systems will not need to install the .NET framework, as it comes pre-installed as part of the operating system.

The source code for HighBandwidthWinUSB.exe file was created in Microsoft Visual C++® 2005 Express Edition. The source code can be found in the "<Install Directory>\ USB Device - WinUSB - High Bandwidth Demo\WinUSB High Bandwidth Demo - PC Application - MS VC++ 2005 Express" directory. Microsoft currently distributes Visual C++ 2005

Express Edition for free, and can be downloaded from Microsoft's website. When downloading Microsoft Visual C++ 2005 Express Edition, also make sure to download and install the Platform SDK, and follow Microsoft's instructions for integrating it with the development environment.

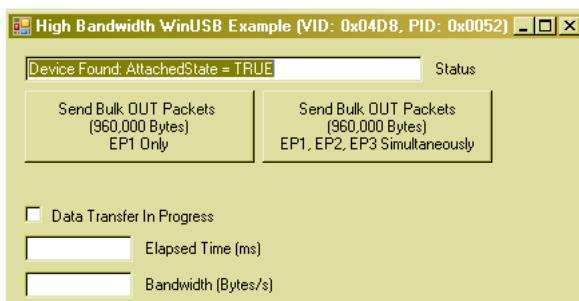
It is not necessary to install either Microsoft Visual C++ 2005, or the Platform SDK in order to begin using the HighBandwidthWinUSB.exe program. These are only required if the source code will be modified or compiled.

To launch the application, simply double click on the executable "HighBandwidthWinUSB.exe" in the "<Install Directory>\USB Device - WinUSB - High Bandwidth Demo" directory. A window like that shown below should appear:



If instead of this window, an error message pops up while trying to launch the application, it is likely the Microsoft .NET Framework Version 2.0 Redistributable Package has not yet been installed. Please install it and try again.

As configured by default, the application is looking for USB devices with VID = 0x04D8 and PID = 0x0052. The device descriptor in the firmware project meant to be used with this demo uses the same VID/PID. Once the device flashed with corresponding firmware is connected to the PC, the below window appears:



Hitting the "Send Bulk OUT Packets" tab will transmit 960,000 bytes of data on the USB bus to the corresponding endpoints (EP1 Only or EP1,EP2, EP3 Simultaneously depending upon the button pressed in the GUI). Elapsed Time (ms) & Bandwidth (Bytes/Sec) are displayed in the GUI once the data transmission is complete.

1.6.16 Host - CDC Serial Demo

This demo shows how to interface to USB CDC devices. This typically includes many cell phone models and USB modems.

Description

Supported Demo Boards

The matrix of which demos are supported on a specific board can be found in the Release Notes demo board support section. Verify that the board you wish to use will work with this demo. This table also describes some of the limitations that the board might have while running this demo.

Demo Board I/O Mapping

Each demo board has a different number of push buttons, LEDs, and other features with various different names for these components. To determine which board features are used for which demo features, please refer to the io_mapping.h file in

the demo folder under the system_config folder. Each demo board will have a corresponding folder with an io_mapping.h file in it. For example, for the PIC24FJ256GB110 PIM for the Explorer 16, this would be the following file:

```
<install_directory>/apps/usb/host/cdc_basic/firmware/src/system_config/exp16/pic24fj256gb110_pim/io_mapping.h
```

For more information about each demo board, please refer to the Demo Board Information section.

Demo Operation

This is a simple demo to show how an embedded CDC host can be implemented. When a CDC-RS232 device is attached to the host, the demo host application polls for input data from the device and displays the data on the LCD mounted on the explorer 16 board.

1.6.17 Host - HID - Keyboard Demo

This demo shows how to interface to USB keyboards. Many USB barcode scanners also appear as a USB keyboard.

Description

Supported Demo Boards

The matrix of which demos are supported on a specific board can be found in the Release Notes demo board support section. Verify that the board you wish to use will work with this demo. This table also describes some of the limitations that the board might have while running this demo.

Demo Board I/O Mapping

Each demo board has a different number of push buttons, LEDs, and other features with various different names for these components. To determine which board features are used for which demo features, please refer to the io_mapping.h file in the demo folder under the system_config folder. Each demo board will have a corresponding folder with an io_mapping.h file in it. For example, for the PIC24FJ256GB110 PIM for the Explorer 16, this would be the following file:

```
<install_directory>/apps/usb/host/hid_keyboard/firmware/src/system_config/exp16/pic24fj256gb110_pim/io_mapping.h
```

For more information about each demo board, please refer to the Demo Board Information section.

Demo Operation

When the device is programmed correctly with the HID host keyboard application the LCD screen on the Explorer 16 should read "Device Detached" if there is no device attached to the USB port. At this point plug in a USB keyboard, bar code scanner that supports HID keyboard emulation, or magnetic card reader that supports HID keyboard emulation. Type a key on the keyboard. This character should be printed on the LCD screen. Pressing the "ESC" key will clear the screen and return the cursor to the first position.

Limitations:

- Neither compound nor composite devices are supported. Some keyboards are either compound or composite.
- The "~" prints as an arrow character instead ("->"). This is an effect of the LCD screen on the Explorer 16. The ascii character for "~" is remapped in the LCD controller.
- The "\\" prints as a "\\$" character instead. This is an effect of the LCD screen on the Explorer 16. The ascii character for "\\" is remapped in the LCD controller.
- Backspace and arrow keys may have issues on Explorer 16 boards with certain LCD modules

1.6.18 Host - HID - Mouse Demo

This demo shows how to read the basic position and button information from a standard USB mouse.

Description

Supported Demo Boards

The matrix of which demos are supported on a specific board can be found in the Release Notes demo board support section. Verify that the board you wish to use will work with this demo. This table also describes some of the limitations that the board might have while running this demo.

Demo Board I/O Mapping

Each demo board has a different number of push buttons, LEDs, and other features with various different names for these components. To determine which board features are used for which demo features, please refer to the io_mapping.h file in the demo folder under the system_config folder. Each demo board will have a corresponding folder with an io_mapping.h file in it. For example, for the PIC24FJ256GB110 PIM for the Explorer 16, this would be the following file:

```
<install_directory>/apps/usb/host/hid_mouse/firmware/src/system_config/exp16/pic24fj256gb110_pim/io_mapping.h
```

For more information about each demo board, please refer to the Demo Board Information section.

Demo Operation

When a device is not attached, the LCD screen indicates to attach a device. When a device is attached, the X/Y coordinates, and left/right mouse button information should be shown on the LCD screen. Some screens don't have enough space to show all of this information and might be truncated. Other boards might not have a screen or the demo has not been ported yet to use their screen. These boards likely print the messages to RAM, which can be viewed using a debugger after the device has been attached.

Limitations:

- Composite and compound device are not currently supported. These devices may not enumerate or operate correctly. Devices with built in USB hubs are a compound device. Many multimedia devices with mouse as one of the interface are composite devices.

1.6.19 Host - Mass Storage - Thumb Drive Data Logger

Description

Supported Demo Boards

The matrix of which demos are supported on a specific board can be found in the Release Notes demo board support section. Verify that the board you wish to use will work with this demo. This table also describes some of the limitations that the board might have while running this demo.

Demo Board I/O Mapping

Each demo board has a different number of push buttons, LEDs, and other features with various different names for these components. To determine which board features are used for which demo features, please refer to the io_mapping.h file in the demo folder under the system_config folder. Each demo board will have a corresponding folder with an io_mapping.h file in it. For example, for the PIC24FJ256GB110 PIM for the Explorer 16, this would be the following file:

```
<install_directory>/apps/usb/host/msd_data_logger/firmware/src/system_config/exp16/pic24fj256gb110_pim/io_mapping.h
```

For more information about each demo board, please refer to the Demo Board Information section.

Demo Operation

This demo will cause the host to start logging the potentiometer data to a thumb drive once the drive is plugged into the board. An LED will blink indicating that the data is being logged to the file. Press the pushbutton on the board to stop the logging to the file. If the drive has an activity LED, wait for it to stop blinking. If it doesn't, wait a few seconds for the write to complete before removing the drive.

NOTE: remove the drive without stopping the write first by pressing the button can result in corrupted or missing data.

1.6.20 Host - Mass Storage (MSD) - Simple Demo

This demo is a simple example of how to write a file to a USB thumb drive.

Description

Supported Demo Boards

The matrix of which demos are supported on a specific board can be found in the Release Notes demo board support section. Verify that the board you wish to use will work with this demo. This table also describes some of the limitations that the board might have while running this demo.

Demo Operation

This demo is a simple example of how to write files to a thumb drive through the Microchip MDD file system library. When a thumb drive is plugged in the code will create a text file on the drive. This process only takes a brief moment. Connect the thumb drive to the board and wait for a couple of seconds. If the drive has an activity LED on it, wait for , remove the drive and plug it back into a computer. There should be an additional text file created named "test.txt".

Limitations:

- Due to the size of this demo, optimizations must be enabled in the compiler in order for this demo to work on the certain hardware platforms. Optimizations are not available on all versions of the compilers.

1.6.21 Configuring the Demo

Low Pin Count USB Development Kit

1. Short J14 between pins 2 and 3. This will power the board from the USB port.
2. Make sure that J12 is left open.

PICDEM FS USB:

- No board specific settings are required

PIC18 Explorer Based Demos

For all of the PIC18 Explorer based demo boards, please follow the following instructions:

1. Set switch S4 to the "ICE" position
2. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:
 - *PIC18F46J50 Plug-In-Module:*
 1. Short JP2 such that the "R" and the "U" options are shorted together.
 2. Short JP3. This allows the demo board to be powered through the USB bus power.
 - *PIC18F47J53 Plug-In-Module:*
 1. Short JP2 such that the "R" and the "U" options are shorted together.
 2. Short JP3. This allows the demo board to be powered through the USB bus power.
 - *PIC18F87J50 Plug-In-Module:*
 1. Short JP1 such that the "R" and the "U" options are shorted together.
 2. Short JP4. This allows the demo board to be powered through the USB bus power.

3. Short JP5. This enabled the LED operation on the board.

Explorer 16 Based Demos

For all of the Explorer 16-based demo boards, please follow the following instructions

1. Connect the USB PICTail+ Daughter Board to the Explorer 16.
2. Short JP1 on the USB PICTail+ board
3. Open JP2, JP3, and JP4 on the USB PICTail+ board
4. Make sure that S2 on the Explorer 16 is switched to the "PIM" setting.
5. Short JP2 on the Explorer 16 to enable the LEDs.
6. Follow any processor specific instructions below. All instructions apply to the PIM unless otherwise stated:
 - *PIC24FJ64GB004 PIM*
 1. Set switch S1 to the "PGX1" setting
 2. Short J1 pin 1 (marked "POT") to the center pin
 3. Short J2 pin 1 (marked "Temp") to the center pin
 4. Short J3 pin 1 (marked "EEPROM CS") to the center pin
 - *PIC24FJ256GB210 PIM*
 1. Short JP1 "U" option to the center pin
 2. Short JP2 "U" option to the center pin
 3. Short JP3 "U" option to the center pin
 4. Short JP4
 - *PIC24EP512GU810 PIM*
 1. Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5
 2. Open jumpers J6, J7, J8, J9, and J10
 - *dsPIC33EP512MU810 PIM*
 1. Short pins 2 and 3 on jumpers J1, J2, J3, J4, and J5
 2. Open jumpers J6, J7, J8, J9, and J10
 - *PIC32MX795F512L PIM*
 1. Open J10
 2. Short pins 1 (marked "USB") and pin 2 (center) of jumpers J1 and J2

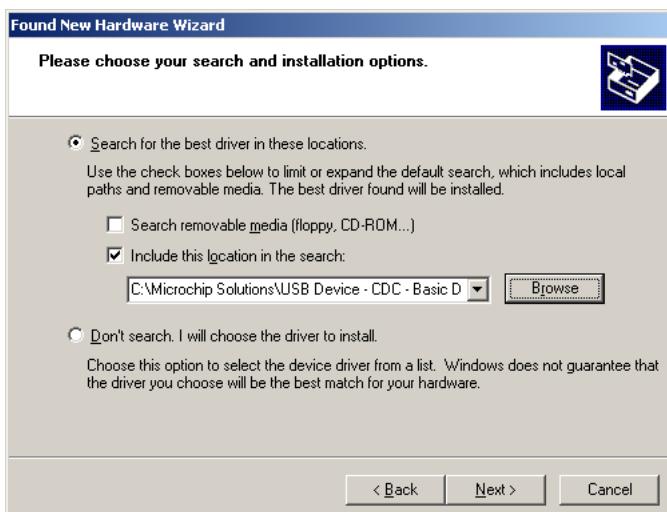
1.6.22 Running the Demo

This demo allows the device to appear like a serial (COM) port to the host. This demo will take data sent over the USB CDC interface and send it on the UART of the microcontroller.

In order to run this demo first compile and program the target device. Attach the device to the host with the USB cable. Also connect the RS232 port of the demo board to a computer. This computer can be the same computer as the USB connection or it can be a different computer. If the host is a PC and this is the first time you have plugged this device into the computer then you may be asked for a .inf file.



Select the “Install from a list or specific location (Advanced)” option. Point to the “<Install Directory>\USB Device - CDC – Serial Emulator\inf” directory



Once the device is successfully installed, open up a terminal program, such as hyperterminal. Select the appropriate COM port for the USB virtual COM port. On most machines this will be COM5 or higher. On the computer where the RS232 cable is attached, open a second terminal program. Select the hardware COM port associated with that computer. Please insure that the baud rate for both terminal windows is the same.

Once everything is configured correctly, typing a key in one terminal window will result in the same data to show up in the second terminal window.

Note: Some terminal programs, like hyperterminal, require users to click the disconnect button before removing the device from the computer. Failing to do so may result in having to close and open the program again in order to reconnect to the device.

1.6.23 Supported Demo Boards

Demo Board (click link for board information)	Notes
Low Pin Count USB Development Kit	
PICDEM FS USB	1

PIC18F46J50 Plug-In-Module (PIM)	1, 2
PIC18F47J53 Plug-In-Module (PIM)	1, 2
PIC18F87J50 Plug-In-Module (PIM)	1, 2
PIC24FJ64GB004 Plug-In-Module (PIM)	1, 3
PIC24FJ256GB110 Plug-In-Module (PIM)	1, 3
PIC24FJ256GB210 Plug-In-Module (PIM)	1, 3
PIC24EP512GU810 Plug-In-Module (PIM)	1, 3
dsPIC33EP512MU810 Plug-In-Module (PIM)	1, 3
PIC32 USB Plug-In-Module (PIM)	1, 3
PIC32 CAN-USB Plug-In-Module (PIM)	1, 3

Notes:

1. These boards require the Speech Playback PICTail/PICTail+ daughter board in order to run this demo.
2. This board can not be used by itself. It requires a PIC18 Explorer board in order to operate with this demo.
3. This board can not be used by itself. It requires an Explorer 16 and a USB PICTail+ Daughter Board in order to operate.

1.7 Appendix (FAQs, Important Information, Reference Material, etc.)

This section contains other useful information about various topics and more detailed information about topics already presented in the help document.

Description

1.7.1 Using breakpoints in USB host applications

This section describes how to use breakpoints when running a USB host application without causing communication issues.

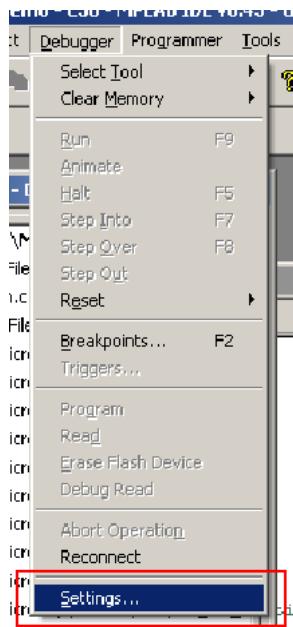
Description

This section describes how to use breakpoints when running a USB host application without causing communication issues.

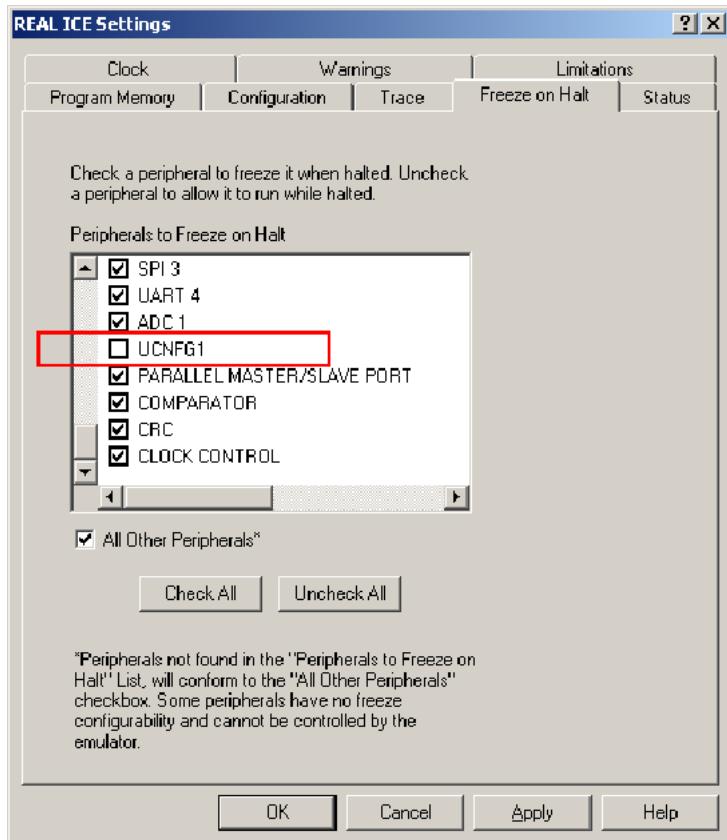
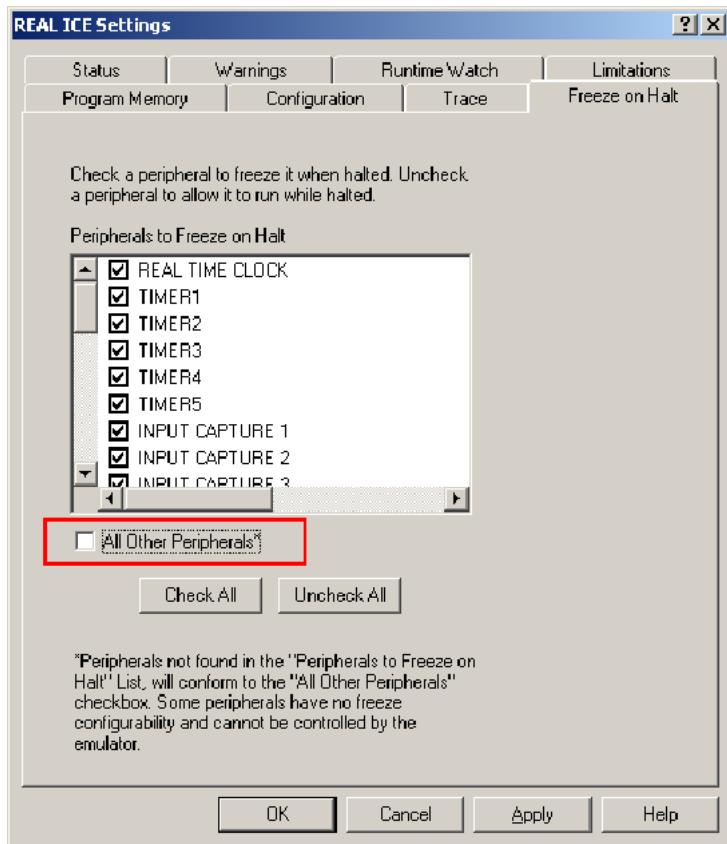
USB has a periodic packet that is sent on the bus once every millisecond, called the start of frame (SOF) packet, that is used to keep the bus from going into an idle/suspended state. When a the microcontroller hits a breakpoint, both the CPU and the modules on the device stop operation. This will cause the attached USB device to enter the suspend mode. Some programmers implement a method that allows specified peripherals to continue to run even after a breakpoint occurs. This section describes how to enable this feature for the USB peripheral on PIC24F and PIC32 devices.

MPLAB v8.x

- 1) Select the desired debugger from the debugger menu
- 2) Go to the “Debugger->Settings” menu option

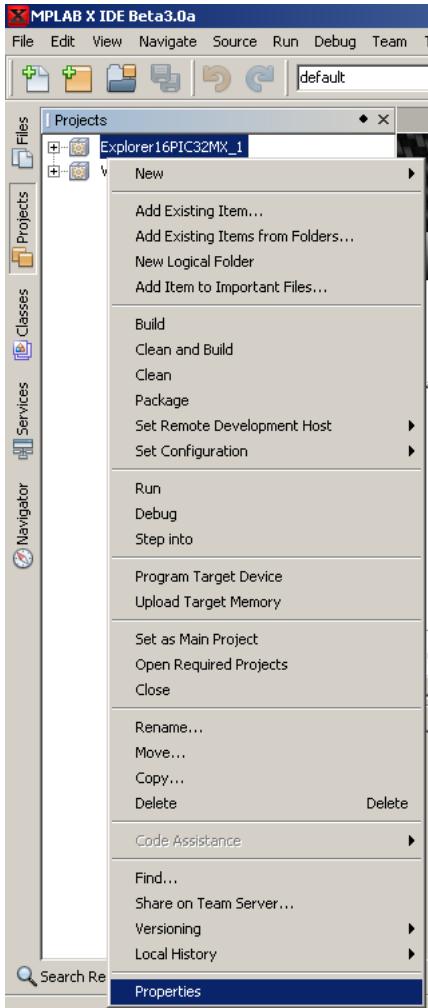


- 3) Go to the Freeze on Halt tab. For PIC24F devices, uncheck the UCNFG1 box. For PIC32 devices, uncheck the “All other peripherals” box located below the scrolling menu.

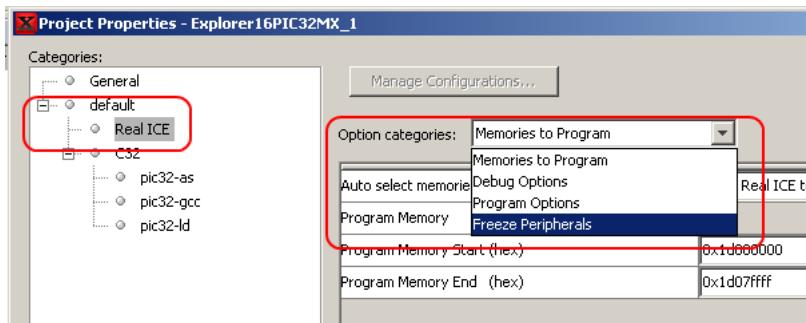
PIC24F**PIC32**

MPLAB X

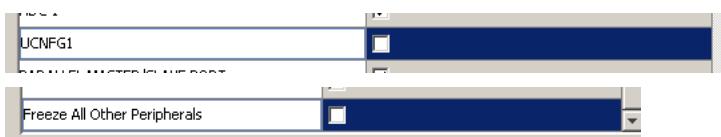
- In the projects window, right click on the project you are working on and select properties from the menu that appears.



- In the properties window, select the debugger that you are currently using from the Categories navigation pane.
- In the resulting form, select "Freeze Peripherals" in the "Option Categories" drop down box.



- In the resulting list uncheck the box corresponding to the USB peripheral. If there is not one on the list, uncheck "All other peripherals". Please note that on PIC24F the USB module may be named UCNFG1.



1.7.2 Notes on .inf Files

Describes important information about .inf file usage and behavior.

Description

Upon initially plugging in a USB device, in some cases Windows will prompt the user for a driver. Rather than having users manually copy .sys files (driver binary files) into important system directories (such as within the “\Windows\system32” directory structure) and manually add registry entries, Windows automates the driver installation process through the use of .INF files. INF files are plain text (can be edited with notepad) installation instruction script files.

Some types of USB devices will not require .INF files or user provided drivers (for example, a HID class mouse). For these types of devices, the operating system makes use of drivers already built into/distributed with the operating system, so no user provided driver or .INF file is necessary.

For other types of devices, Windows will prompt the user for a driver. In these cases, point Windows to the .INF file relevant for the USB device. All of the example projects included in the MCHPFSUSB framework which need an INF file are provided with an example INF file. The INF file will need slight modification (most importantly to change the VID and PID) before commercial distribution.

The INF file for the custom demo can be found in <Install Directory>\USB Tools\MCHPUSB Custom Driver\MCHPUSB DriverRelease.

The INF file for the CDC demos can be found in <Install Directory>\USB Tools\USB CDC Serial Demo\inf\win2k_winxp.

1.7.3 Vendor IDs (VID) and Product IDs (PID)

Describes important information about Vendor IDs (VID) and Product IDs (PID).

Description

Every USB product line must have a unique combination of VID and PID. All firmware examples use Microchip's VID (0x04d8) and a unique PID. Prior to manufacturing and marketing a new USB product, the VID and PID need to be changed. New VID and PID numbers can be obtained by purchasing a VID from the USB Implementers Forum:

<http://www.usb.org/developers/vendor>

Alternatively, Microchip has a free VID sublicensing program. An application form for obtaining a PID (for use with Microchip's VID: 0x04d8) from Microchip can be obtained through the following link: <http://www.microchip.com/usblicensing/Default.aspx>

Once a new VID/PID combination is obtained, both the firmware and the .INF file (when applicable) will need to be updated.

To modify the VID/PID in one of the example USB firmware projects, open the usb_descriptors.c file (found in each of the demo folders). They should appear in the table used for the USB Device Descriptor. Change both values as needed.

To modify the VID/PID in the .INF file, open the relevant INF file and search for the “[DeviceList]” sections. There are two sections, one for 32-bit and one for 64-bit, both sections should be identical. In these sections, some text will appear with the form “USB\VID_xxxx&PID_yyyy”. Update the “xxxx” and “yyyy” sections with the new hexadecimal format VID/PID values.

1.7.4 Using a Diff Tool

Refer to the Section "Using a Diff Tool" in help_mla_getting_started file for more details.

1.7.5 Driver Signing and Windows 8

This section provides information related to USB driver signatures, the types of signatures needed for the different versions of Windows operating system, and how to get a signed driver package.

1.7.5.1 What are "Signed" Drivers?

What are "Signed" Drivers?

Most USB drivers operate in what is known as "kernel mode" on Windows based PCs. Kernel mode drivers have low level access to the PC and its resources. This low level access to the PC is normally necessary to implement the kind of functionality that the driver is intended to provide to top level applications.

However, low level access to a PC has potential security implications. Kernel mode is typically the "ideal" place where malicious software developers would want their software to operate, since it provides the greatest control and access to the PC. Therefore, in the interest of protecting Windows security, Windows OSes place restrictions on what code is allowed to be operated in kernel mode.

Windows "trusts" drivers and executable programs that have been signed, more so than software that is unsigned. Signing a driver package is analogous to placing an embossed wax seal on an envelope. The signature/wax seal does not effect or alter the contents of the package, but it provides proof that the contents have not been modified or tampered with, since the time that the signature/wax seal was first applied.

There are three types of USB driver signatures to be aware of:

Embedded digital signatures: This type of signature resides inside of driver .sys files (kernel mode driver binary files). No additional/external files are associated with this type of signature. These types of signatures only protect against tampering with the .sys file itself, and do not include other files that may be a part of the driver package (ex: .inf and .dll files). All Microsoft OS provided driver .sys files, as well as most third party kernel mode drivers will contain at least this level of signature.

"Full driver package" digital signature – Microsoft Authenticode: This type of signature can be thought of as a "wrapper" over the entire driver package content files. A driver package can be as simple as a single .inf file (a plain text installation instruction file that Windows uses when installing new drivers), or may encompass additional files (such as .dll and/or .sys files). The full driver package signature comes in the form of a properly created security catalog file (.cat), which will be part of the driver package distribution. A driver package signed with an Authenticode signature is relatively easy to create, but it is less trustworthy than that of a WHQL digital signature.

"Full driver package" digital signature – WHQL: This type of signature is the most trusted by Windows, and is very similar to the full driver package Microsoft Authenticode signature above, but is more expensive and harder to obtain. To obtain a Windows Hardware Quality Labs (WHQL) signature, a driver package must undergo extensive testing, and passing log files and submission fees must be supplied to Microsoft. If a driver package has already previously been tested and WHQL certified, but has since been modified, in some cases it is possible to get the driver re-certified through a simpler and cheaper "Driver Update Acceptable" process with Microsoft.

Any modifications to a driver package once the signature has been applied, including adding or deleting a single character of whitespace in the driver .inf file, will invalidate a full driver package signature. A driver package can however have two simultaneous signatures, one covering the full driver package, and one embedded inside the driver binary file(s). Inf file modifications do not invalidate an embedded digital signature inside of a driver binary file.

Once a signature has been invalidated, Windows will no longer trust the driver package as much, and will place restrictions on its installation (or outright prevent its installation on some OSes). The driver package can however be re-signed, to restore the trustworthiness of the driver to Windows.

1.7.5.2 Minimum Driver Signature Requirements

Minimum Driver Signature Requirements

Full driver package WHQL signatures are the best and most trusted by all versions of Windows. Windows allows the installation of properly WHQL signed drivers, without producing a prompt warning the user about the driver's trustworthiness.

However, current Windows versions do not require WHQL signatures to allow installation. Lesser signatures (or no signatures in some cases) are allowed, but will generate user dialogs/warnings during the installation process.

Operating System	Minimum Signature to Allow Installation
Windows 2000	None
Windows XP 32-bit	None
Windows XP 64-bit	None
Windows Vista 32-bit	None
Windows Vista 64-bit	Embedded
Windows 7 32-bit	Embedded
Windows 7 64-bit	Embedded
Windows 8 32-bit	Embedded
Windows 8 64-bit	Embedded + Full package authenticode
Windows RT (ARM)	Third party drivers and driver packages are not currently allowed. All USB devices for this OS must use Microsoft supplied drivers.

1.7.5.3 Using Older Drivers with Windows 8

Using Older Drivers with Windows 8

In general, USB driver packages that are designed for Windows 7 and prior OS versions will also work in Windows 8, but there is one important exception to this.

Starting with Windows 8 64-bit, all drivers must contain a proper "full driver package" digital signature (prior OSes only required an embedded signature in the .sys file, rather than the entire driver package including the .inf file). The driver package signature exists as a .cat file that comes with the driver package, and needs to be correctly referenced from within

the .inf file. If either the .cat file is entirely missing, or it is not being correctly referenced from the .inf file, Windows 8 will generate an error message, when the user attempts to install the driver:

"The third party INF does not contain digital signature information".

If the .cat file is present and is correctly referenced, but something in the driver package was modified since the signature was applied, a slightly different error message will occur:

"The hash for the file is not present in the specified catalog file. The file is likely corrupt or the victim of tampering."

In both cases, Windows 8 64-bit will not allow the driver package to be installed, even though it may technically be capable of functioning correctly. To fix this, the driver package must be properly signed with a full package signature. This signature may be either a WHQL signature (which is the best kind of signature), or a "Microsoft Authenticode" signature.

In the February 2013 or later version of the Microchip Libraries for Applications (MLA, available from www.microchip.com/mla), the CDC, WinUSB, and MCHPUSB driver packages all include a WHQL signature and can be installed successfully on Windows 8 32 and 64 bit (as well as prior OSes). When the firmware is using the same VID/PID as the default value from the demo, then the latest driver package from the MLA should install directly.

When the application uses a customized .inf file (ex: VID/PID and/or strings are different), then it will not be possible to directly use the driver package from the MLA. The reason for this, is that anytime anyone makes any changes whatsoever to the driver package (including adding or deleting one character of whitespace in the .inf file), this will break and invalidate the driver package signature. Therefore, even if the .cat file is present, the signature will be invalid (and still won't install correctly).

Therefore, if an application needs to use a custom modified driver package, the only practical solution is to make the modifications, and then re-sign the driver package. A driver package can be signed with an authenticode signature using the procedure outlined in the section "Using a Code Signing Certificate to Sign Driver Packages". A package signed with the Microsoft authenticode signature will install successfully on Windows 8, but will still produce a user prompt asking if they would like to trust the company that signed the driver package. This user dialog can be suppressed if the driver package instead contains a WHQL signature.

Although not very suitable for end consumers, Windows 8 does have a feature that allows one to temporarily disable driver package signing enforcement. This is particularly useful for development and testing purposes. The feature is hidden under several layers of menus and requires the following steps to enable:

1. From the desktop, move the mouse to the lower right hand corner of the screen, to launch the charm bar.
2. Click the Settings "gear" icon.
3. Click the "Change PC Settings" option.
4. In the PC Settings menu on the left, select the "General" option.
5. In the right hand pane, scroll down to the bottom of the options list. Under the "Advanced startup" section, click the "Restart now" button. This doesn't directly reboot the computer, but launches a page that provides additional restart options.
6. In the "Choose an option" page, select the "Troubleshoot" option.

7. From the Troubleshoot menu, click on “Advanced options”.
8. In the “Advanced options” dialog, click the “Startup Settings” option.
9. From the “Startup Settings” dialog, click the “Restart” button.
10. The computer should now begin a reboot cycle. During the boot up sequence, a special “Startup Settings” dialog screen should appear.
11. On the “Startup Settings” dialog, press the “F7” key, to select the “Disable driver signature enforcement” option.
12. Allow Windows 8 to finish booting up.

Once driver signing enforcement is disabled, unsigned driver packages can then be installed. After rebooting the machine, driver signing enforcement will be re-enabled, but Windows 8 will continue to allow the unsigned driver(s) that were installed to be loaded for the hardware, without requiring the system to be repeatedly rebooted into the driver signing enforcement disabled mode.

1.7.5.4 Driver Signatures in the Microchip Libraries for Applications (MLA) Projects

Driver Signatures in the Microchip Libraries for Applications (MLA) Projects

Projects based on WinUSB: WinUSB is a Microsoft created/supplied driver. All Microsoft supplied drivers contain an embedded signature from Microsoft. Additionally, WinUSB driver packages supplied in the February 2013 MLA release (or later) also contain a full driver package Microsoft WHQL signature.

In operating systems prior to Windows 8, WinUSB based devices require the user to install a driver package for the hardware. However, starting with Windows 8, it is possible to make WinUSB based devices that are fully plug and play, and do not require any user supplied driver package. Windows 8 allows for automatic installation of the WinUSB driver, when the device firmware implements the correct Microsoft specific “OS” and related USB descriptors. These special descriptors are optional, but when implemented, allow for automatic driver installation using the in-box provided WinUSB driver that is distributed with the operating system installation. These special OS descriptors are implemented starting the WinUSB based firmware projects in the 2012-10-15 MLA Release. For all new application designs, it is recommended to include these special descriptors as they will result in a better end user experience, free of any driver package/signing/installation concerns.

Projects based on CDC: When used with Windows, the CDC projects in the MLA use the Microsoft created/supplied “usbser.sys” driver. This driver contains an embedded signature from Microsoft. Additionally, CDC driver packages supplied in the February 2013 MLA release (or later), also contain a full driver package Microsoft WHQL signature.

Projects based on MCHPUSB: The MCHPUSB driver is a Microchip created/supplied driver. This driver contains an embedded signature from Microchip. Additionally, this driver package contains a WHQL signature. However, when designing a new application, it is suggested to consider using the WinUSB driver instead.

Projects based on libusb: There are multiple versions of the “libusb driver”. Libusb version 0.1 devices rely on a custom driver that can be signed with “libwdi”. Libusb version 1.0 devices, when attached to Windows based machines, relies on the Microsoft supplied WinUSB driver (see WinUSB section).

Projects based on PHDC: The PHDC projects in the MLA rely on the libusb driver (see libusb section).

Projects based on HID, MSD, audio class, CCID: These USB device classes/projects rely on Microsoft supplied drivers that are distributed with the operating system, and do not require any user supplied driver packages or .inf files. Therefore, driver package signing is usually not relevant for these types of applications, as the drivers will normally get installed automatically when the hardware is attached to the machine.

1.7.5.5 Obtaining a Microsoft Authenticode Code Signing Certificate

Obtaining a Microsoft Authenticode Code Signing Certificate

There are several Certificate Authority (CA) companies that can sell your organization a signing certificate that will allow you to sign your own driver packages. However, when submitting a driver package to Microsoft for WHQL certification, either as a new device/driver, or by reusing a previous submission through the “Driver Update Acceptable” (DUA) process, Microsoft currently requires that the submitted files be signed with an authenticode signing certificate issued by VeriSign.

Therefore, it is generally preferred to obtain the Microsoft Authenticode code signing certificate from VeriSign (now a part of Symantec Corporation). Before purchasing the certificate, it is recommended to search for possible promotional/discounted rates. Historically Microsoft has run a program providing for discounted prices for first time purchasers of VeriSign certificates.

Authenticode code signing certificates are usually sold on an annual or multi-year basis. Once purchased, the signing certificate can normally be used to sign an unlimited number of driver package security catalog files (ex: .cat files), along with other types of files (ex: .exe executable programs). The certificate itself (ex: typically a .pvk file, though other extensions are possible) needs to be kept physically secure, and should never be distributed publicly.

1.7.5.6 Code Signing Certificates (Other Uses)

Code Signing Certificates – Other Uses

In addition to signing driver packages, a Microsoft Authenticode signing certificate can be used to sign certain other types of files, such as executable (.exe) programs. Windows, especially Windows 8, does not trust unsigned executables as much as signed executables. In Windows 8, an unsigned executable that has “no history” and has no reputation established with Microsoft will be treated as relatively untrustworthy, and is blocked from execution, unless the user manually overrides the OS behavior, through an advanced options dialogue that is typically hard for new users to find.

Additionally, some virus scanning applications also rely on executable signatures, to help establish relative trustworthiness. In some cases, unsigned executables, free of malware/viruses, can still be blocked from execution by the virus scanning software, until a history/reputation is built up establishing the executable as trustworthy. Signing the executable with a Microsoft Authenticode signing certificate will generally make the executable more trustworthy and less likely to be (incorrectly) flagged as malware.

1.7.5.7 Using a Code Signing Certificate to Sign Driver Packages

Using a Code Signing Certificate to Sign Driver Packages

If you make modifications to a driver package and need to re-sign the package, the easiest method is to sign it with a Microsoft Authenticode code signing certificate. This can be done with the following procedure:

1. Start from a known working driver package .inf file from the latest MLA release.
2. Modify the .inf as desired. The .inf file is a plain text (ex: editable with Notepad) installation instruction/information file that tells the OS what driver needs to be used for the hardware, and anything else that may need to happen during the driver installation process. When changing the .inf file device list sections, please remove all existing Microchip VID/PIDs, before replacing them with your own. The manufacturer and product strings should also be updated as applicable for your device.
3. Delete the security catalog file (.cat) that is already supplied with the package. After modifying the .inf file, the security catalog file will no longer be valid and you will need to create a new one.
4. Download the latest version of the Windows Driver Kit (WDK) from Microsoft (this is currently at: <http://msdn.microsoft.com/en-us/library/windows/hardware/gg487428.aspx>). Version 8.0 or later is needed (prior versions don't have awareness of Windows 8 specifics).
5. Use the "Inf2Cat" utility in the WDK to re-generate a new .cat file from the modified .inf file.

1. Inf2Cat is a command line utility. Open a command prompt, navigate to the directory of the inf2cat tool, and then run it at the command line to get a small help/explanation of usage syntax. The program is typically located in the following location: C:\Program Files\Windows Kits\8.0\bin\x64 (or \x86 folder for 32-bit OS)
2. Typical usage syntax would be similar to the following (all on one line):

```
inf2cat /driver:C:[path] to dir with .inf file]
/os:XP_X86,XP_X64,Vista_X86,Vista_X64,7_X86,7_X64,8_X86,8_X64,Server2003_X86,Server2003_X64,Server2008_X86,
Server2008_X64,Server2008R2_X64,Server8_X64
```

Assuming the inf2cat utility runs successfully, it will generate a "raw" .cat file. The .cat file will still need to be signed, in order to be useful.

6. If your organization does not already have one, purchase a code signing certificate from a Certificate Authority (CA) such as VeriSign (now Symantec Corporation). See the section "Obtaining a Microsoft Authenticode Code Signing Certificate" for more details.
7. Use the "signtool.exe" utility, along with the signing certificate purchased from the CA, to sign the .cat file. The signtool utility is small Microsoft program that is distributed in the Windows SDK (and/or in older versions of the WDK, prior to v8.0). The Windows SDK can currently be obtained from:

<http://msdn.microsoft.com/en-us/windows/desktop/hh852363.aspx>

Typical syntax when using the signtool would be as follows (when executed in the directory of the .cat file, assuming directory to the signtool is in the path, and the certificate has a .pfx extension without a password, and that the certificate resides on "E:", like a typical USB flash drive):

```
signtool sign /v /f "E:[path to certificate][certificate file name].pfx" /t http://timestamp.verisign.com/scripts/timestamp.dll
[FileNameToSign.cat]
```

8. Verify that the signature has been properly applied using the verify command line option:

```
signtool verify /a /pa [FileNameToSign.cat]
```

The verify step should report success. The driver package should now be correctly signed with a Microsoft Authenticode signature. Test it on all target OSes. Distribute both the .inf file and .cat file together to the end consumer (along with any other driver package files that may be necessary, which may include .dll files, particularly in the case of the WinUSB driver package). Never distribute the signing certificate that you purchased from the CA, this should be kept in a safe place, out of the hands of the public (the certificate can be re-used to sign any number of driver packages, as well as .exe files, which will have some benefits).

Index

- _USBHOST_H_ 133
 - _USBHOST_H_ macro 133
 - _CLIENT_DRIVER_TABLE 124
 - _CLIENT_DRIVER_TABLE structure 124
 - _COMM_INTERFACE_DETAILS 144
 - _COMM_INTERFACE_DETAILS structure 144
 - _DATA_INTERFACE_DETAILS 145
 - _DATA_INTERFACE_DETAILS structure 145
 - _HID_COLLECTION 180
 - _HID_COLLECTION structure 180
 - _HID_DATA_DETAILS 180
 - _HID_DATA_DETAILS structure 180
 - _HID_GLOBALS 181
 - _HID_GLOBALS structure 181
 - _HID_ITEM_INFO 182
 - _HID_ITEM_INFO structure 182
 - _HID_REPORT 183
 - _HID_REPORT structure 183
 - _HID_REPORTITEM 183
 - _HID_REPORTITEM structure 183
 - _HID_STRINGITEM 184
 - _HID_STRINGITEM structure 184
 - _HID_TRANSFER_DATA 184
 - _HID_TRANSFER_DATA structure 184
 - _HID_USAGEITEM 185
 - _HID_USAGEITEM structure 185
 - _HOST_TRANSFER_DATA 125
 - _HOST_TRANSFER_DATA structure 125
 - _USB_CDC_ACM_FN_DSC 146
 - _USB_CDC_ACM_FN_DSC structure 146
 - _USB_CDC_CALL_MGT_FN_DSC 146
 - _USB_CDC_CALL_MGT_FN_DSC structure 146
 - _USB_CDC_CONTROL_SIGNAL_BITMAP 146
 - _USB_CDC_CONTROL_SIGNAL_BITMAP union 146
 - _USB_CDC_DEVICE_INFO 147
 - _USB_CDC_DEVICE_INFO structure 147
 - _USB_CDC_HEADER_FN_DSC 148
 - _USB_CDC_HEADER_FN_DSC structure 148
 - _USB_CDC_LINE_CODING 148
 - _USB_CDC_LINE_CODING union 148
 - _USB_CDC_UNION_FN_DSC 149
 - _USB_CDC_UNION_FN_DSC structure 149
 - _USB_DEVICE_H 71
 - _USB_DEVICE_H macro 71
 - _USB_HID_DEVICE_ID 186
 - _USB_HID_DEVICE_ID structure 186
 - _USB_HID_DEVICE_RPT_INFO 186
 - _USB_HID_DEVICE_RPT_INFO structure 186
 - _USB_HID_ITEM_LIST 188
 - _USB_HID_ITEM_LIST structure 188
 - _USB_HOST_CDC_H_ 166
 - _USB_HOST_CDC_H_ macro 166
 - _USB_HOST_CDC_INTERFACE_H_ 166
 - _USB_HOST_CDC_INTERFACE_H_ macro 166
 - _USB_HOST_HID_PARSER_H_ 196
 - _USB_HOST_HID_PARSER_H_ macro 196
 - _USBHOSTMSD_H_ 214
 - _USBHOSTMSD_H_ macro 214
- A**
- Adding a boot loader to your project 254
 - Android 3.1+ 280
 - Appendix (FAQs, Important Information, Reference Material, etc.) 290
 - Application Version Information 237
 - Audio Function Driver 75
- B**
- Boot Loader Entry 243
 - BOOT_INTF_SUBCLASS 96
 - BOOT_INTF_SUBCLASS macro 96
 - BOOT_PROTOCOL 96
 - BOOT_PROTOCOL macro 96
- C**
- C18 Compiler 249
 - CDC Client Driver 133
 - CDC Function Driver 76
 - CDC_H 90
 - CDC_H macro 90

CDCInitEP	80	Device - Boot Loader - HID	234
CDCInitEP function	80	Device - CDC - Serial Emulator	273
CDCNotificationHandler	80	Device - CDC Basic Demo	265
CDCNotificationHandler function	80	Device - HID - Custom Demo	268
CDCSetBaudRate	85	Device - HID - Digitizer Demos	271
CDCSetBaudRate macro	85	Device - HID - Joystick Demo	272
CDCSetCharacterFormat	86	Device - HID - Keyboard Demo	274
CDCSetCharacterFormat macro	86	Device - HID - Mouse Demo	274
CDCSetDataSize	86	Device - HID - Uninterruptible Power Supply	275
CDCSetDataSize macro	86	Device - Mass Storage - Internal Flash Demo	275
CDCSetLineCoding	87	Device - Mass Storage - SD Card Reader	277
CDCSetLineCoding macro	87	Device - Vendor Driver Basic Demo	277
CDCSetParity	87	Device - Vendor High Bandwidth Demo	282
CDCSetParity macro	87	Device (Slave) Demo Board Support and Limitations	30
CDCTxService	81	Device Stack	40
CDCTxService function	81	Device/Peripheral	40
Changing the Memory Footprint	249	DEVICE_CLASS_CDC	149
Changing the memory footprint of the boot loader	262	DEVICE_CLASS_CDC macro	149
CLIENT_DRIVER_TABLE	124	DEVICE_CLASS_HID	190
CLIENT_DRIVER_TABLE structure	124	DEVICE_CLASS_HID macro	190
Code Signing Certificates (Other Uses)	298	DEVICE_CLASS_MASS_STORAGE	206
COMM_INTERFACE_DETAILS	144	DEVICE_CLASS_MASS_STORAGE macro	206
COMM_INTERFACE_DETAILS structure	144	DEVICE_INTERFACE_PROTOCOL_BULK_ONLY	206
Command Set	238	DEVICE_INTERFACE_PROTOCOL_BULK_ONLY macro	206
Configuration Bits	236	DEVICE_SUBCLASS_CD_DVD	206
Configuring the Demo	286	DEVICE_SUBCLASS_CD_DVD macro	206
Customizing for an Application	235	DEVICE_SUBCLASS_FLOPPY_INTERFACE	206
		DEVICE_SUBCLASS_FLOPPY_INTERFACE macro	206
		DEVICE_SUBCLASS_RBC	207
		DEVICE_SUBCLASS_RBC macro	207
		DEVICE_SUBCLASS_REMOVABLE	207
		DEVICE_SUBCLASS_REMOVABLE macro	207
		DEVICE_SUBCLASS_SCSI	207
		DEVICE_SUBCLASS_SCSI macro	207
		DEVICE_SUBCLASS_TAPE_DRIVE	207
		DEVICE_SUBCLASS_TAPE_DRIVE macro	207
		deviceRptInfo	189
		deviceRptInfo variable	189
		Driver Signatures in the Microchip Libraries for Applications (MLA) Projects	297
		Driver Signing and Windows 8	294
		DSC_HID	190
		DSC_HID macro	190

D

Data Types and Constants	68, 90, 95, 99, 123, 142, 177, 205		
DATA_INTERFACE_DETAILS	145	DEVICE_SUBCLASS_REMOVABLE	207
DATA_INTERFACE_DETAILS structure	145	DEVICE_SUBCLASS_REMOVABLE macro	207
Demo Board Information	215	DEVICE_SUBCLASS_SCSI	207
Demos	228	DEVICE_SUBCLASS_SCSI macro	207
DESC_CONFIG_uint32_t	69	DEVICE_SUBCLASS_TAPE_DRIVE	207
DESC_CONFIG_uint32_t macro	69	DEVICE_SUBCLASS_TAPE_DRIVE macro	207
DESC_CONFIG_uint8_t	69		
DESC_CONFIG_uint8_t macro	69		
DESC_CONFIG_WORD	70		
DESC_CONFIG_WORD macro	70		
Device - Audio Microphone Basic Demo	228		
Device - Audio MIDI Demo	230		

DSC_PHY 190	EVENT_HID_RESET_ERROR 192
DSC_PHY macro 190	EVENT_HID_RESET_ERROR macro 192
DSC_RPT_wValue 198	EVENT_HID_RPT_DESC_PARSED 192
DSC_RPT_wValue macro 198	EVENT_HID_RPT_DESC_PARSED macro 192
dsPIC33EP512MU810 Plug-In-Module (PIM) 226	EVENT_HID_WRITE_DONE 192
	EVENT_HID_WRITE_DONE macro 192
E	EVENT_MSD_ATTACH 207
Embedded Host API 104	EVENT_MSD_ATTACH macro 207
Embedded Host Stack 104	EVENT_MSD_MAX_LUN 208
ERASE_DEVICE 240	EVENT_MSD_MAX_LUN macro 208
EVENT_CDC_ATTACH 149	EVENT_MSD_NONE 208
EVENT_CDC_ATTACH macro 149	EVENT_MSD_NONE macro 208
EVENT_CDC_COMM_READ_DONE 150	EVENT_MSD_OFFSET 208
EVENT_CDC_COMM_READ_DONE macro 150	EVENT_MSD_OFFSET macro 208
EVENT_CDC_COMM_WRITE_DONE 150	EVENT_MSD_RESET 208
EVENT_CDC_COMM_WRITE_DONE macro 150	EVENT_MSD_RESET macro 208
EVENT_CDC_DATA_READ_DONE 150	EVENT_MSD_TRANSFER 208
EVENT_CDC_DATA_READ_DONE macro 150	EVENT_MSD_TRANSFER macro 208
EVENT_CDC_DATA_WRITE_DONE 150	Explorer 16 224
EVENT_CDC_DATA_WRITE_DONE macro 150	
EVENT_CDC_NAK_TIMEOUT 151	
EVENT_CDC_NAK_TIMEOUT macro 151	
EVENT_CDC_NONE 151	Flash Signature 264
EVENT_CDC_NONE macro 151	From v2.10 to v2.11 35
EVENT_CDC_OFFSET 151	From v2.11 to v2.12 35
EVENT_CDC_OFFSET macro 151	From v2.12 to v2.13 34
EVENT_CDC_RESET 151	From v2.13 to v2.14+ 34
EVENT_CDC_RESET macro 151	From v2.5 to v2.6 38
EVENT_HID_ATTACH 190	From v2.6 to v2.6a 38
EVENT_HID_ATTACH macro 190	From v2.6a to v2.7 38
EVENT_HID_BAD_REPORT_DESCRIPTOR 191	From v2.7 to v2.7a 37
EVENT_HID_BAD_REPORT_DESCRIPTOR macro 191	From v2.7a to v2.8 37
EVENT_HID_DETACH 191	From v2.8 to v2.9 37
EVENT_HID_DETACH macro 191	From v2.9 to v2.9a 37
EVENT_HID_NONE 191	From v2.9a to v2.9b 37
EVENT_HID_NONE macro 191	From v2.9b to v2.9c 37
EVENT_HID_OFFSET 191	From v2.9c to v2.9d 37
EVENT_HID_OFFSET macro 191	From v2.9d to v2.9e 37
EVENT_HID_READ_DONE 191	From v2.9e to v2.9f 36
EVENT_HID_READ_DONE macro 191	From v2.9f to v2.9g 36
EVENT_HID_RESET 192	From v2.9g to v2.9h 36
EVENT_HID_RESET macro 192	From v2.9h to v2.9i 36
	From v2.9i to v2.9j 36

From v2.9j to v2.10	35	HIDRxHandleBusy	93
Functions	40, 75, 78, 92, 97, 101, 104, 135, 167, 199	HIDRxHandleBusy macro	93
G		HIDRxPacket	93
Garage Band '08 [Macintosh Computers]	230	HIDRxPacket macro	93
GET_DATA	241	HIDTxHandleBusy	94
getsUSBUSART	82	HIDTxHandleBusy macro	94
getsUSBUSART function	82	HIDTxPacket	95
		HIDTxPacket macro	95
		Host - CDC Serial Demo	283
H		Host - HID - Keyboard Demo	284
HID Client Driver	166	Host - HID - Mouse Demo	284
HID Function Driver	92	Host - Mass Storage - Thumb Drive Data Logger	285
HID_COLLECTION	180	Host - Mass Storage (MSD) - Simple Demo	286
HID_COLLECTION structure	180	Host Application Responsibilities	237
HID_DATA_DETAILS	180	Host Demo Board Support and Limitations	31
HID_DATA_DETAILS structure	180	HOST_TRANSFER_DATA	125
HID_DESIGITEM	181	HOST_TRANSFER_DATA structure	125
HID_DESIGITEM structure	181		
HID_GLOBALS	181	I	
HID_GLOBALS structure	181	Implementation Details	238
HID_ITEM_INFO	182	Importance of Change the VID/PID	235
HID_ITEM_INFO structure	182	INIT_CL_SC_P	127
HID_PROTOCOL_KEYBOARD	96	INIT_CL_SC_P macro	127
HID_PROTOCOL_KEYBOARD macro	96	INIT_VID_PID	127
HID_PROTOCOL_MOUSE	96	INIT_VID_PID macro	127
HID_PROTOCOL_MOUSE macro	96	Input Button/Hardware Entry	243
HID_PROTOCOL_NONE	97	Interrupt Remapping	257
HID_PROTOCOL_NONE macro	97	Introduction	16
HID_REPORT	183	itemListPtrs	190
HID_REPORT structure	183	itemListPtrs variable	190
HID_REPORTITEM	183		
HID_REPORTITEM structure	183	L	
HID_STRINGITEM	184	Legal Information	17
HID_STRINGITEM structure	184	Library Interface	40
HID_TRANSFER_DATA	184	Library Migration	34
HID_TRANSFER_DATA structure	184	Linking Options for PIC16 Devices	250
HID_USAGEITEM	185	Linking Options for PIC18 Devices	252
HID_USAGEITEM structure	185	Linux	266, 279
HID_USER_DATA_SIZE	185	Low Pin Count USB Development Board	215
HID_USER_DATA_SIZE type	185	LUN_FUNCTIONS	99
HIDReportTypeEnum	185	LUN_FUNCTIONS structure	99
HIDReportTypeEnum enumeration	185		

M

Macintosh 267
 Mass Storage Client Driver 199
 Memory Map 245, 254
 Memory Region Definitions 258
 Merging Bootloader and Application Project Output 253
 Minimum Driver Signature Requirements 295
 MSD Function Driver 97
 MSD_COMMAND_FAILED 209
 MSD_COMMAND_FAILED macro 209
 MSD_COMMAND_PASSED 209
 MSD_COMMAND_PASSED macro 209
 MSD_PHASE_ERROR 209
 MSD_PHASE_ERROR macro 209
 MSDTasks 98
 MSDTasks function 98
 MSDTransferTerminated 98
 MSDTransferTerminated function 98
 mUSBUSARTIsTxTrfReady 88
 mUSBUSARTIsTxTrfReady macro 88
 mUSBUSARTTxRam 88
 mUSBUSARTTxRam macro 88
 mUSBUSARTTxRom 89
 mUSBUSARTTxRom macro 89

N

Notes on .inf Files 293
 NUM_STOP_BITS_1 91
 NUM_STOP_BITS_1 macro 91
 NUM_STOP_BITS_1_5 91
 NUM_STOP_BITS_1_5 macro 91
 NUM_STOP_BITS_2 91
 NUM_STOP_BITS_2 macro 91

O

Obtaining a Microsoft Authenticode Code Signing Certificate 298
 Online Reference and Resources 29
 Operating System Support and Limitations 32
 Optimization Mode Requirements 250

P

PARITY_EVEN 91
 PARITY_EVEN macro 91
 PARITY_MARK 91
 PARITY_MARK macro 91
 PARITY_NONE 92
 PARITY_NONE macro 92
 PARITY_ODD 92
 PARITY_ODD macro 92
 PARITY_SPACE 92
 PARITY_SPACE macro 92
 PIC16 and PIC18 245
 PIC18 Starter Kit 218
 PIC18F46J50 Plug-In-Module (PIM) 219
 PIC18F47J53 Plug-In-Module (PIM) 220
 PIC18F87J50 Plug-In-Module (PIM) Demo Board 221
 PIC24EP512GU810 Plug-In-Module (PIM) 226
 PIC24F 254
 PIC24F Starter Kit 222
 PIC24FJ256DA210 Development Board 222
 PIC24FJ256GB110 Plug-In-Module (PIM) 225
 PIC24FJ256GB210 Plug-In-Module (PIM) 225
 PIC24FJ64GB004 Plug-In-Module (PIM) 225
 PICDEM FS USB Board 217
 Processor Specific Implementation Details 245
 PROGRAM_COMPLETE 241
 PROGRAM_DEVICE 241
 putrsUSBUSART 82
 putrsUSBUSART function 82
 putsUSBUSART 83
 putsUSBUSART function 83
 putUSBUSART 84
 putUSBUSART function 84

Q

QUERY_DEVICE 238
 QUERY_EXTENDED_INFO 243
 QUERY_EXTENDED_INFO Response 248

R

Release Notes 18
RESET_DEVICE 242
Revision History 18
Running the Demo 287

UNLOCK_CONFIG 240
USB Library 15
USB PICTail Plus Daughter Board 226
USB_APPLICATION_EVENT_HANDLER 43
USB_APPLICATION_EVENT_HANDLER function 43
USB_CDC_ABSTRACT_CONTROL_MODEL 151
USB_CDC_ABSTRACT_CONTROL_MODEL macro 151
USB_CDC_ACN_FN_DSC 146
USB_CDC_ACN_FN_DSC structure 146

S

Safe Boot Loading Considerations 236
SIGN_FLASH 242
Software/Application Entry 244
Special Region Creation 259
Startup Sequence and Reset Remapping 256
Support 29
Supported Demo Boards 288

USB_CDC_ATM_NETWORKING_CONTROL_MODEL 152
USB_CDC_ATM_NETWORKING_CONTROL_MODEL macro 152
USB_CDC_CALL_MGT_FN_DSC 146
USB_CDC_CALL_MGT_FN_DSC structure 146
USB_CDC_CAPI_CONTROL_MODEL 152
USB_CDC_CAPI_CONTROL_MODEL macro 152
USB_CDC_CLASS_ERROR 152
USB_CDC_CLASS_ERROR macro 152

T

Tool Information 34
TPL_ALLOW_HNP 127
TPL_ALLOW_HNP macro 127
TPL_CLASS_DRV 128
TPL_CLASS_DRV macro 128
TPL_EPO_ONLY_CUSTOM_DRIVER 128
TPL_EPO_ONLY_CUSTOM_DRIVER macro 128
TPL_IGNORE_CLASS 128
TPL_IGNORE_CLASS macro 128
TPL_IGNORE_PID 128
TPL_IGNORE_PID macro 128
TPL_IGNORE_PROTOCOL 128
TPL_IGNORE_PROTOCOL macro 128
TPL_IGNORE_SUBCLASS 129
TPL_IGNORE_SUBCLASS macro 129
TPL_SET_CONFIG 129
TPL_SET_CONFIG macro 129
TRANSFER_ATTRIBUTES 125
TRANSFER_ATTRIBUTES type 125
Troubleshooting 276

USB_CDC_COMM_INTF 152
USB_CDC_COMM_INTF macro 152
USB_CDC_COMMAND_FAILED 152
USB_CDC_COMMAND_FAILED macro 152
USB_CDC_COMMAND_PASSED 153
USB_CDC_COMMAND_PASSED macro 153
USB_CDC_CONTROL_LINE_LENGTH 153
USB_CDC_CONTROL_LINE_LENGTH macro 153
USB_CDC_CONTROL_SIGNAL_BITMAP 146
USB_CDC_CONTROL_SIGNAL_BITMAP union 146
USB_CDC_CS_ENDPOINT 153
USB_CDC_CS_ENDPOINT macro 153
USB_CDC_CS_INTERFACE 153
USB_CDC_CS_INTERFACE macro 153
USB_CDC_DATA_INTF 153
USB_CDC_DATA_INTF macro 153
USB_CDC_DEVICE_BUSY 154
USB_CDC_DEVICE_BUSY macro 154
USB_CDC_DEVICE_DETACHED 154
USB_CDC_DEVICE_DETACHED macro 154
USB_CDC_DEVICE_HOLDING 154
USB_CDC_DEVICE_HOLDING macro 154
USB_CDC_DEVICE_INFO 147
USB_CDC_DEVICE_INFO structure 147
USB_CDC_DEVICE_MANAGEMENT 154

U

Understanding and Customizing the Boot Loader Implementation 258

USB_CDC_DEVICE_MANAGEMENT macro 154	USB_CDC_INITIALIZING macro 158
USB_CDC_DEVICE_NOT_FOUND 154	USB_CDC_INTERFACE_ERROR 158
USB_CDC_DEVICE_NOT_FOUND macro 154	USB_CDC_INTERFACE_ERROR macro 158
USB_CDC_DIRECT_LINE_CONTROL_MODEL 155	USB_CDC_LINE_CODING 148
USB_CDC_DIRECT_LINE_CONTROL_MODEL macro 155	USB_CDC_LINE_CODING union 148
USB_CDC_DSC_FN_ACN 155	USB_CDC_LINE_CODING_LENGTH 158
USB_CDC_DSC_FN_ACN macro 155	USB_CDC_LINE_CODING_LENGTH macro 158
USB_CDC_DSC_FN_CALL_MGT 155	USB_CDC_MAX_PACKET_SIZE 159
USB_CDC_DSC_FN_CALL_MGT macro 155	USB_CDC_MAX_PACKET_SIZE macro 159
USB_CDC_DSC_FN_COUNTRY_SELECTION 155	USB_CDC_MOBILE_DIRECT_LINE_MODEL 159
USB_CDC_DSC_FN_COUNTRY_SELECTION macro 155	USB_CDC_MOBILE_DIRECT_LINE_MODEL macro 159
USB_CDC_DSC_FN_DLM 155	USB_CDC_MULTI_CHANNEL_CONTROL_MODEL 159
USB_CDC_DSC_FN_DLM macro 155	USB_CDC_MULTI_CHANNEL_CONTROL_MODEL macro 159
USB_CDC_DSC_FN_HEADER 156	USB_CDC_NO_PROTOCOL 159
USB_CDC_DSC_FN_HEADER macro 156	USB_CDC_NO_PROTOCOL macro 159
USB_CDC_DSC_FN_RPT_CAPABILITIES 156	USB_CDC_NO_REPORT_DESCRIPTOR 159
USB_CDC_DSC_FN_RPT_CAPABILITIES macro 156	USB_CDC_NO_REPORT_DESCRIPTOR macro 159
USB_CDC_DSC_FN_TEL_OP_MODES 156	USB_CDC_NORMAL_RUNNING 160
USB_CDC_DSC_FN_TEL_OP_MODES macro 156	USB_CDC_NORMAL_RUNNING macro 160
USB_CDC_DSC_FN_TELEPHONE_RINGER 156	USB_CDC_OBEX 160
USB_CDC_DSC_FN_TELEPHONE_RINGER macro 156	USB_CDC_OBEX macro 160
USB_CDC_DSC_FN_UNION 156	USB_CDC_PHASE_ERROR 160
USB_CDC_DSC_FN_UNION macro 156	USB_CDC_PHASE_ERROR macro 160
USB_CDC_DSC_FN_USB_TERMINAL 157	USB_CDC_REPORT_DESCRIPTOR_BAD 160
USB_CDC_DSC_FN_USB_TERMINAL macro 157	USB_CDC_REPORT_DESCRIPTOR_BAD macro 160
USB_CDC_ETHERNET_EMULATION_MODEL 157	USB_CDC_RESET_ERROR 161
USB_CDC_ETHERNET_EMULATION_MODEL macro 157	USB_CDC_RESET_ERROR macro 161
USB_CDC_ETHERNET_NETWORKING_CONTROL_MODEL 157	USB_CDC_RESETTING_DEVICE 161
USB_CDC_ETHERNET_NETWORKING_CONTROL_MODEL macro 157	USB_CDC_RESETTING_DEVICE macro 161
USB_CDC_GET_COMM_FEATURE 157	USB_CDC_SEND_BREAK 161
USB_CDC_GET_COMM_FEATURE macro 157	USB_CDC_SEND_BREAK macro 161
USB_CDC_GET_ENCAPSULATED_REQUEST 157	USB_CDC_SEND_ENCAPSULATED_COMMAND 161
USB_CDC_GET_ENCAPSULATED_REQUEST macro 157	USB_CDC_SEND_ENCAPSULATED_COMMAND macro 161
USB_CDC_GET_LINE_CODING 158	USB_CDC_SET_COMM_FEATURE 161
USB_CDC_GET_LINE_CODING macro 158	USB_CDC_SET_COMM_FEATURE macro 161
USB_CDC_HEADER_FN_DSC 148	USB_CDC_SET_CONTROL_LINE_STATE 162
USB_CDC_HEADER_FN_DSC structure 148	USB_CDC_SET_CONTROL_LINE_STATE macro 162
USB_CDC_ILLEGAL_REQUEST 158	USB_CDC_SET_LINE_CODING 162
USB_CDC_ILLEGAL_REQUEST macro 158	USB_CDC_SET_LINE_CODING macro 162
USB_CDC_INITIALIZING 158	USB_CDC_TELEPHONE_CONTROL_MODEL 162
	USB_CDC_UNION_FN_DSC 149

USB_CDC_UNION_FN_DSC structure	149	USB_HID_RPT_DESC_ERROR enumeration	188
USB_CDC_V25TER	162	USB_HID_TRANSFER_IN	198
USB_CDC_V25TER macro	162	USB_HID_TRANSFER_IN macro	198
USB_CDC_WIRELESS_HANDSET_CONTROL_MODEL	162	USB_HID_TRANSFER_OUT	198
USB_CDC_WIRELESS_HANDSET_CONTROL_MODEL macro	162	USB_HID_TRANSFER_OUT macro	198
usb_host.h	130		
USB_CLIENT_EVENT_HANDLER	126	USB_HOST_APP_DATA_EVENT_HANDLER	106, 129
USB_CLIENT_EVENT_HANDLER type	126	USB_HOST_APP_DATA_EVENT_HANDLER function	106
USB_CLIENT_INIT	126	USB_HOST_APP_DATA_EVENT_HANDLER macro	129
USB_CLIENT_INIT type	126	USB_HOST_APP_EVENT_HANDLER	106, 129
usb_device.h	72	USB_HOST_APP_EVENT_HANDLER function	106
usb_device_audio.h	76	USB_HOST_APP_EVENT_HANDLER macro	129
usb_device_cdc.h	77	usb_host_cdc.h	163
usb_device_generic.h	103	usb_host_cdc_interface.h	165
usb_device_hid.h	97	usb_host_hid.h	193
usb_device_msdu.h	100	usb_host_hid_parser.h	196
USB_DEVICE_STACK_EVENTS	69	USB_HOST_HID_RETURN_CODES	195
USB_DEVICE_STACK_EVENTS type	69	USB_HOST_HID_RETURN_CODES enumeration	195
USB_DEVICE_STATE	69	usb_host_msdu.h	213
USB_DEVICE_STATE type	69	USB_HostInterruptHandler	107
USB_EP0_BUSY	70	USB_HostInterruptHandler function	107
USB_EP0_BUSY macro	70	USB_MSD_CBW_ERROR	209
USB_EP0_INCLUDE_ZERO	70	USB_MSD_CBW_ERROR macro	209
USB_EP0_INCLUDE_ZERO macro	70	USB_MSD_COMMAND_FAILED	209
USB_EP0_NO_DATA	70	USB_MSD_COMMAND_FAILED macro	209
USB_EP0_NO_DATA macro	70	USB_MSD_COMMAND_PASSED	210
USB_EP0_NO_OPTIONS	71	USB_MSD_COMMAND_PASSED macro	210
USB_EP0_NO_OPTIONS macro	71	USB_MSD_CSW_ERROR	210
USB_EP0_RAM	71	USB_MSD_CSW_ERROR macro	210
USB_EP0_RAM macro	71	USB_MSD_DEVICE_BUSY	210
USB_EP0_ROM	71	USB_MSD_DEVICE_BUSY macro	210
USB_EP0_ROM macro	71	USB_MSD_DEVICE_DETACHED	210
USB_HANDLE	71	USB_MSD_DEVICE_DETACHED macro	210
USB_HANDLE macro	71	USB_MSD_DEVICE_NOT_FOUND	211
USB_HID_CLASS_ERROR	193	USB_MSD_DEVICE_NOT_FOUND macro	211
USB_HID_CLASS_ERROR macro	193	USB_MSD_ERROR	211
USB_HID_DEVICE_ID	186	USB_MSD_ERROR macro	211
USB_HID_DEVICE_ID structure	186	USB_MSD_ERROR_STATE	211
USB_HID_DEVICE_RPT_INFO	186	USB_MSD_ERROR_STATE macro	211
USB_HID_DEVICE_RPT_INFO structure	186	USB_MSD_ILLEGAL_REQUEST	211
USB_HID_ITEM_LIST	188	USB_MSD_ILLEGAL_REQUEST macro	211
USB_HID_ITEM_LIST structure	188	USB_MSD_INITIALIZING	211
USB_HID_RPT_DESC_ERROR	188		

USB_MSD_INITIALIZING macro 211	USBCtrlEPAllowStatusStage function 44
USB_MSD_INVALID_LUN 212	USBDeferINDataStage 45
USB_MSD_INVALID_LUN macro 212	USBDeferINDataStage function 45
USB_MSD_MEDIA_INTERFACE_ERROR 212	USBDeferOUTDataStage 46
USB_MSD_MEDIA_INTERFACE_ERROR macro 212	USBDeferOUTDataStage function 46
USB_MSD_NORMAL_RUNNING 212	USBDeferStatusStage 47
USB_MSD_NORMAL_RUNNING macro 212	USBDeferStatusStage function 47
USB_MSD_OUT_OF_MEMORY 212	USBDeviceAttach 48
USB_MSD_OUT_OF_MEMORY macro 212	USBDeviceAttach function 48
USB_MSD_PHASE_ERROR 212	USBDeviceDetach 48
USB_MSD_PHASE_ERROR macro 212	USBDeviceDetach function 48
USB_MSD_RESET_ERROR 213	USBDeviceInit 50
USB_MSD_RESET_ERROR macro 213	USBDeviceInit function 50
USB_MSD_RESETTING_DEVICE 213	USBDeviceTasks 50
USB_MSD_RESETTING_DEVICE macro 213	USBDeviceTasks function 50
USB_NUM_BULK_NAKS 129	USBEnableEndpoint 51
USB_NUM_BULK_NAKS macro 129	USBEnableEndpoint function 51
USB_NUM_COMMAND_TRIES 130	USBEP0Receive 52
USB_NUM_COMMAND_TRIES macro 130	USBEP0Receive function 52
USB_NUM_CONTROL_NAKS 130	USBEP0SendRAMPtr 53
USB_NUM_CONTROL_NAKS macro 130	USBEP0SendRAMPtr function 53
USB_NUM_ENUMERATION_TRIES 130	USBEP0SendROMPptr 53
USB_NUM_ENUMERATION_TRIES macro 130	USBEP0SendROMPptr function 53
USB_NUM_INTERRUPT_NAKS 130	USBEP0Transmit 54
USB_NUM_INTERRUPT_NAKS macro 130	USBEP0Transmit function 54
USB_TPL 127	USBGEN_H 102
USB_TPL type 127	USBGEN_H macro 102
USBCancelIO 44	USBGenRead 102
USBCancelIO function 44	USBGenRead macro 102
USBCDCEventHandler 84	USBGenWrite 103
USBCDCEventHandler function 84	USBGenWrite macro 103
USBCheckAudioRequest 76	USBGet1msTickCount 54
USBCheckAudioRequest function 76	USBGet1msTickCount function 54
USBCheckCDCRequest 85	USBGetDeviceState 56
USBCheckCDCRequest function 85	USBGetDeviceState function 56
USBCheckMSDRequest 98	USBGetNextHandle 57
USBCheckMSDRequest function 98	USBGetNextHandle function 57
USBCheckVendorRequest 101	USBGetRemoteWakeupsStatus 58
USBCheckVendorRequest function 101	USBGetRemoteWakeupsStatus function 58
USBCtrlEPAllowDataStage 44	USBGetSuspendState 59
USBCtrlEPAllowDataStage function 44	USBGetSuspendState function 59
USBCtrlEPAllowStatusStage 44	USBGetTicksSinceSuspendEnd 55

USBGetTicksSinceSuspendEnd function 55	USBHostGetDeviceDescriptor macro 120
USBHandleBusy 60	USBHostGetStringDescriptor 120
USBHandleBusy function 60	USBHostGetStringDescriptor macro 120
USBHandleGetAddr 61	USBHostHID_ApiFindBit 168
USBHandleGetAddr function 61	USBHostHID_ApiFindBit function 168
USBHandleGetLength 61	USBHostHID_ApiFindValue 168
USBHandleGetLength function 61	USBHostHID_ApiFindValue function 168
USBHostCDC_Api_ACM_Request 136	USBHostHID_ApiGetCurrentInterfaceNum 169
USBHostCDC_Api_ACM_Request function 136	USBHostHID_ApiGetCurrentInterfaceNum function 169
USBHostCDC_Api_Get_IN_Data 136	USBHostHID_ApiImportData 169
USBHostCDC_Api_Get_IN_Data function 136	USBHostHID_ApiImportData function 169
USBHostCDC_Api_Send_OUT_Data 137	USBHostHID_GetCurrentReportInfo 170
USBHostCDC_Api_Send_OUT_Data function 137	USBHostHID_GetCurrentReportInfo macro 170
USBHostCDC_ApiDeviceDetect 137	USBHostHID_GetItemListPointers 171
USBHostCDC_ApiDeviceDetect function 137	USBHostHID_GetItemListPointers macro 171
USBHostCDC_ApiTransferIsComplete 137	USBHostHID_HasUsage 170
USBHostCDC_ApiTransferIsComplete function 137	USBHostHID_HasUsage function 170
USBHostCDCDeviceStatus 138	USBHostHIDDeviceDetect 171
USBHostCDCDeviceStatus function 138	USBHostHIDDeviceDetect function 171
USBHostCDCEventHandler 138	USBHostHIDDeviceStatus 172
USBHostCDCEventHandler function 138	USBHostHIDEEventHandler function 172
USBHostCDCInitAddress 139	USBHostHIDEEventHandler function 172
USBHostCDCInitAddress function 139	USBHostHIDInitialize 173
USBHostCDCInitialize 139	USBHostHIDInitialize function 173
USBHostCDCInitialize function 139	USBHostHIDRead 173
USBHostCDCResetDevice 140	USBHostHIDRead function 173
USBHostCDCResetDevice function 140	USBHostHIDReadIsComplete 174
USBHostCDCTasks 140	USBHostHIDReadIsComplete function 174
USBHostCDCTasks function 140	USBHostHIDReadTerminate 174
USBHostCDCTransfer 141	USBHostHIDReadTerminate function 174
USBHostCDCTransfer function 141	USBHostHIDResetDevice 174
USBHostCDCTransferIsComplete 142	USBHostHIDResetDevice function 174
USBHostCDCTransferIsComplete function 142	USBHostHIDTasks 175
USBHostClearEndpointErrors 107	USBHostHIDTasks function 175
USBHostClearEndpointErrors function 107	USBHostHIDWrite 176
USBHostDeviceSpecificClientDriver 108	USBHostHIDWrite function 176
USBHostDeviceSpecificClientDriver function 108	USBHostHIDWritelsComplete 176
USBHostDeviceStatus 108	USBHostHIDWritelsComplete function 176
USBHostDeviceStatus function 108	USBHostHIDWriteTerminate 177
USBHostGetCurrentConfigurationDescriptor 119	USBHostHIDWriteTerminate function 177
USBHostGetCurrentConfigurationDescriptor macro 119	USBHostInit 109
USBHostGetDeviceDescriptor 120	

USBHostInit function 109	USBHostShutdown function 115
USBHostIsochronousBuffersCreate 110	USBHostSuspendDevice 116
USBHostIsochronousBuffersCreate function 110	USBHostSuspendDevice function 116
USBHostIsochronousBuffersDestroy 110	USBHostTasks 116
USBHostIsochronousBuffersDestroy function 110	USBHostTasks function 116
USBHostIsochronousBuffersReset 111	USBHostTerminateTransfer 117
USBHostIsochronousBuffersReset function 111	USBHostTerminateTransfer function 117
USBHostIssueDeviceRequest 111	USBHostTransferIsComplete 117
USBHostIssueDeviceRequest function 111	USBHostTransferIsComplete function 117
USBHostMSDDeviceStatus 200	USBHostVbusEvent 118
USBHostMSDDeviceStatus function 200	USBHostVbusEvent function 118
USBHostMSDEventHandler 200	USBHostWrite 119
USBHostMSDEventHandler function 200	USBHostWrite function 119
USBHostMSDInitialize 201	USBHostWriteIsochronous 122
USBHostMSDInitialize function 201	USBHostWriteIsochronous macro 122
USBHostMSDRead 204	USBIncrement1msInternalTimers 62
USBHostMSDRead macro 204	USBIncrement1msInternalTimers function 62
USBHostMSDResetDevice 201	USBINDataStageDeferred 62
USBHostMSDResetDevice function 201	USBINDataStageDeferred function 62
USBHostMSDTasks 202	USBIsBusSuspended 63
USBHostMSDTasks function 202	USBIsBusSuspended function 63
USBHostMSDTerminateTransfer 202	USBIsDeviceSuspended 63
USBHostMSDTerminateTransfer function 202	USBIsDeviceSuspended function 63
USBHostMSDTransfer 203	USBMSDInit 99
USBHostMSDTransfer function 203	USBMSDInit function 99
USBHostMSDTransferIsComplete 203	USBOUTDataStageDeferred 64
USBHostMSDTransferIsComplete function 203	USBOUTDataStageDeferred function 64
USBHostMSDWrite 204	USBRxOnePacket 65
USBHostMSDWrite macro 204	USBRxOnePacket function 65
USBHostRead 112	USBSoftDetach 65
USBHostRead function 112	USBSoftDetach function 65
USBHostReadIsochronous 122	USBStallEndpoint 66
USBHostReadIsochronous macro 122	USBStallEndpoint function 66
USBHostResetDevice 113	USBTransferOnePacket 66
USBHostResetDevice function 113	USBTransferOnePacket function 66
USBHostResumeDevice 113	USBTxOnePacket 67
USBHostResumeDevice function 113	USBTxOnePacket function 67
USBHostSetDeviceConfiguration 114	USBUSARTIsTxTrfReady 89
USBHostSetDeviceConfiguration function 114	USBUSARTIsTxTrfReady macro 89
USBHostSetNAKTimeout 115	Using a Code Signing Certificate to Sign Driver Packages 298
USBHostSetNAKTimeout function 115	Using a Diff Tool 293
USBHostShutdown 115	Using breakpoints in USB host applications 290

Using Linux MultiMedia Studio (LMMS) [Linux and Windows Computers] 232

Using Older Drivers with Windows 8 295

V

v2.10 20

v2.11 20

v2.12 20

v2.13 19

v2.14 19

v2.15 18

v2.16 18

v2.17 18

v2.7 27

v2.7a 27

v2.8 26

v2.9 25

v2.9a 25

v2.9b 24

v2.9c 24

v2.9d 23

v2.9e 23

v2.9f 22

v2.9g 22

v2.9h 21

v2.9i 21

v2.9j 20

Vendor Class (Generic) Function Driver 101

Vendor IDs (VID) and Product IDs (PID) 293

W

What are "Signed" Drivers? 294

Windows 266, 278

X

XC8 Compiler 250