USB Human Interface Device (HID) Library for Analog Devices ADSP-BF70x User's Guide Revision 2.00

Closed Loop Design, LLC

support@cld-llc.com

Table of Contents

Disclaimer

This software is supplied "AS IS" without any warranties, express, implied or statutory, including but not limited to the implied warranties of fitness for purpose, satisfactory quality and non-infringement. Closed Loop Design LLC extends you a royalty-free right to reproduce and distribute executable files created using this software for use on Analog Devices Blackfin family processors only. Nothing else gives you the right to use this software.

Introduction

The Closed Loop Design (CLD) HID library creates a simplified interface for developing a Human Interface Device (HID) using the Analog Devices ADSP-BF707 EZ-Board. The CLD BF70x HID library also includes support for a serial console and timer functions which facilitates creating timed events quickly and easily. The library's BF707 application interface is comprised of parameters used to customize the library's functionality as well as callback functions used to notify the User application of events. These parameters and functions are described in greater detail in the CLD BF70x HID Library API section of this document.

USB Background

The following is a very basic overview of some of the USB concepts which are necessary to use the CLD BF70x HID Library. However, it is still recommended that developers have at least a basic understanding of the USB 2.0 protocol as well as the HID 1.11 Protocol. The following are some resources to refer to when working with USB:

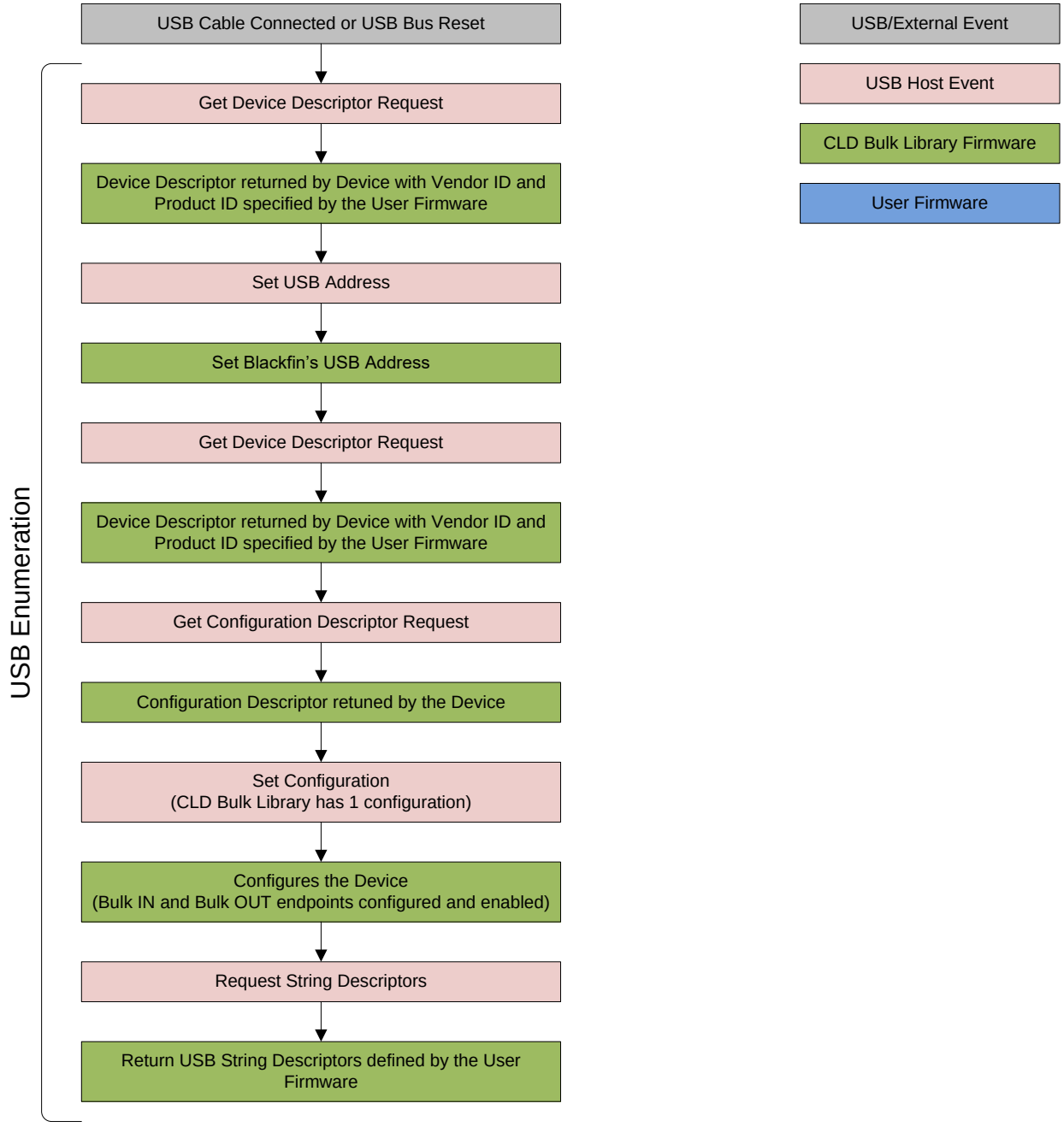
- The USB 2.0 Specification: http://www.usb.org/developers/docs/usb20_docs/
- The USB HID Class specification v1.11:http://www.usb.org/developers/hidpage/
- USB in a Nutshell: A free online wiki that explains USB concepts. http://www.beyondlogic.org/usbnutshell/usb1.shtml
- "USB Complete" by Jan Axelson ISBN: 1931448086

USB is a polling based protocol where the Host initiates all transfers, so all USB terminology is from the Host's perspective. For example, an 'IN' transfer is when data is sent from a Device to the Host, and an 'OUT' transfer is when the Host sends data to a Device.

The USB 2.0 protocol defines a basic framework devices must implement in order to work correctly. This framework is defined in the Chapter 9 of the USB 2.0 protocol, and is often referred to as the USB 'Chapter 9' functionality. Part of the Chapter 9 framework is standard USB requests used by a USB Host to control the Device. Another part of the Chapter 9 framework is the USB Descriptors. These USB

Descriptors are used to notify the Host of the Device's capabilities when the Device is attached. The USB Host uses the descriptors and the Chapter 9 standard requests to configure the Device. This process is called the USB Enumeration. The CLD BF70x HID Library includes support for the USB standard requests and USB Enumeration using some of the parameters specified by the User application when initializing the library. These parameters are discussed in the cld_bf70x_hid_lib_init section of this document. The CLD BF70x HID Library facilitates USB enumeration and is Chapter 9 compliant without User Application intervention as shown in the flow chart below. If you'd like additional information on USB Chapter 9 functionality or USB Enumeration please refer to one of the USB resources listed above.

CLD BF70x HID Library USB Enumeration Flow Chart



All USB data is transferred using Endpoints which act as a source or sink for data based on the endpoint's direction (IN or OUT). The USB protocol defines four types of Endpoints, each of which has unique characteristics that dictate how they are used. The four Endpoint types are: Control, Interrupt, Bulk and Isochronous. Data transmitted over USB is broken up into blocks of data called packets. For each endpoint type there are restrictions on the allowed max packet size. The allowed max packet sizes also vary based on the USB connection speed. Please refer to the USB 2.0 protocol for more information about the max packet size supported by the four endpoint types.

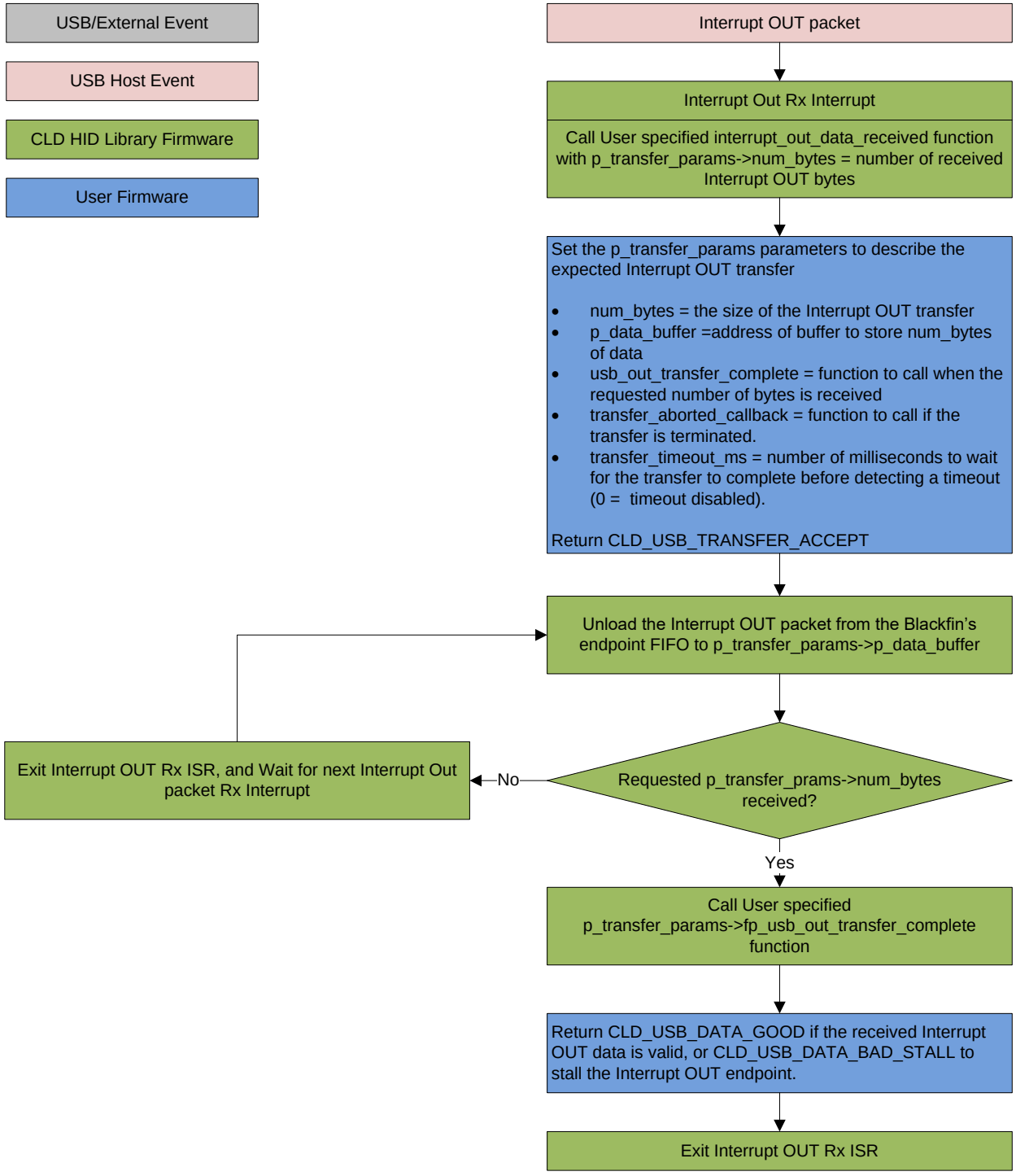
The CLD BF70x HID Library uses Control and Interrupt endpoints, so these endpoint types will be discussed in more detail below.

A Control Endpoint is the only bi-directional endpoint type, and is typically used for command and status transfers. A Control Endpoint transfer is made up of three stages (Setup Stage, Data Stage and Status Stage). The Setup Stage sets the direction and size of the optional Data Stage. The Data Stage is where any data is transferred between the Host and Device. The Status Stage gives the Device the opportunity to report if an error was detected during the transfer. All USB Devices are required to include a default Control Endpoint at endpoint number 0, referred to as Endpoint 0. Endpoint 0 is used to implement all the USB Protocol defined Chapter 9 framework and USB Enumeration. In the CLD BF70x HID Library Endpoint 0 is used for USB Chapter 9 requests, as well as HID Get/Set requests. These HID requests are discussed in more detail in the HID Background section of this document.

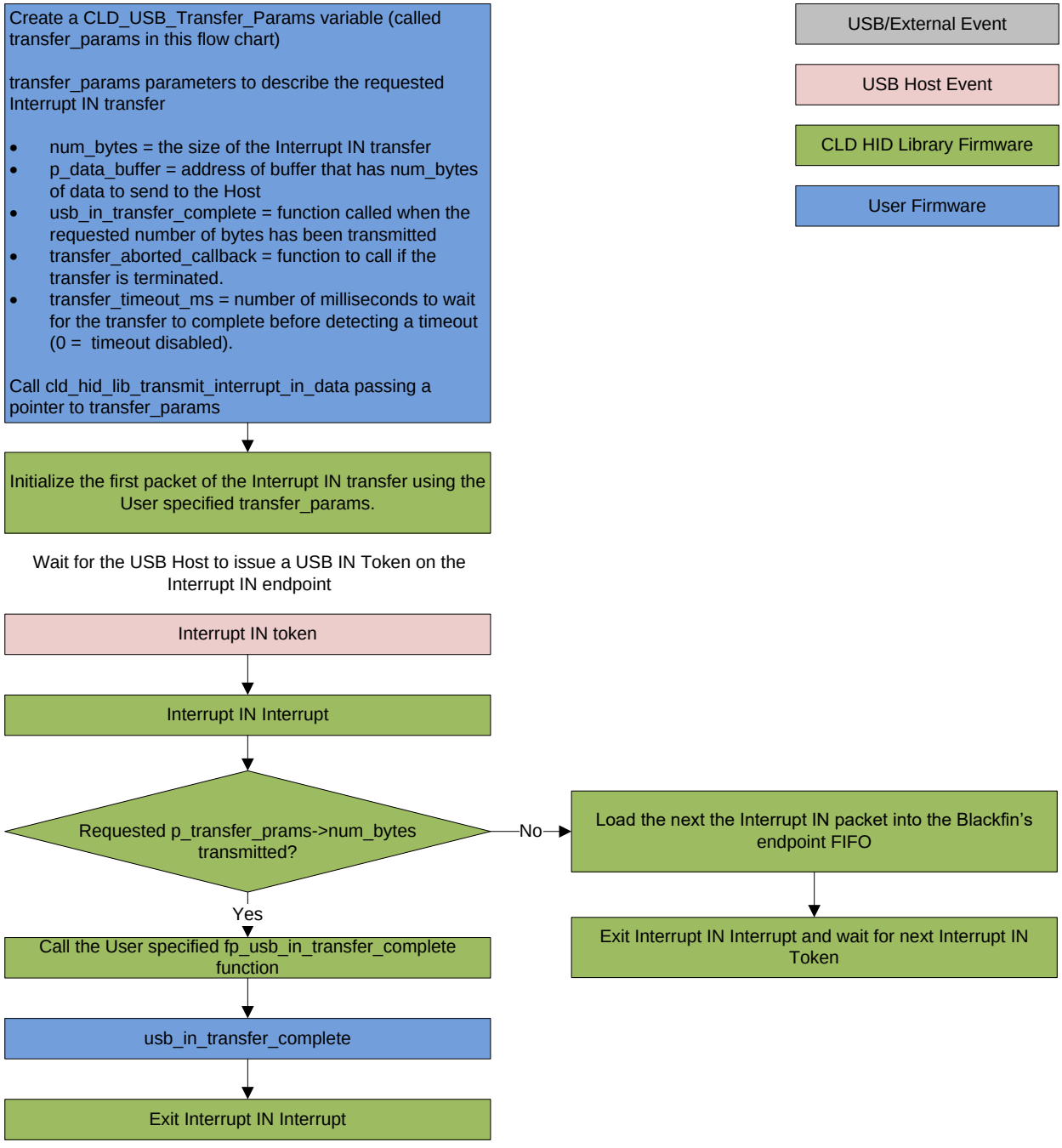
Interrupt Endpoints are used to transfer blocks of data where data integrity, and deterministic timing is required. Deterministic timing is achieved by allowing the Device to specify a requested interval used by the Host to initiate USB transfers, which gives the Device a guaranteed maximum time between opportunities to transfer data. Interrupt Endpoints are particularly useful when the Device needs to report to the Host when a change is detected without having to wait for the Host to ask for the information. An example of how this is used with HID is a USB Mouse. When a User moves the mouse or presses a button the mouse reports this change to the Host using the HID Interrupt IN endpoint. This is more efficient then requiring the host to repeatedly send Control Endpoint requests asking if the mouse inputs have changed.

The flow charts below give an overview of how the CLD BF70x HID Library and the User firmware interact to process Interrupt IN and Interrupt OUT transfers. Additionally, the User firmware code snippets included at the end of this document provide a basic framework for implementing the HID firmware using the CLD BF70x HID Library.

CLD BF70x HID Library Interrupt OUT Flow Chart



CLD BF70x HID Library Interrupt IN Flow Chart



HID Background

The USB Human Interface Device (HID) protocol is a USB Standard Class protocol released by the USB IF committee. The HID protocol was created to provide a standardized way USB devices that interface with a human could be controlled over USB. The HID protocol covers a wide range of uses including, but not limited to: keyboards, joysticks, button panels, touch screens, and alphanumeric displays.

In the HID protocol all data sent between the Host and Device is transferred using data structures called Reports, and each Report can include a variety data elements of various types and sizes. For example: a USB mouse has a single Report which it uses to report the mouse's position and button state. The format of this report is shown in the C structure below:

```
typedef struct
{
    unsigned char button;    /* Mouse button state  */
    signed char x;           /* X position */
    signed char y;           /* Y position */
} Mouse_Input_Report;
```

However, the Device needs to describe the structure and intended use of its Reports the Host. The HID protocol accomplishes this using the HID Report Descriptor which includes the information required by the Host to process the Device's Reports. The HID Report Descriptor uses identifiers defined in the HID protocol to describe the various elements which make up a Report, as well as how multiple data elements are organized in the Reports. The Report Descriptor also specifies if the Report is an INPUT, OUTPUT or FEATURE. An INPUT Report can only be sent from the Device to the Host. An OUTPUT Report can only be sent from the Host to the Device. While a FEATURE Report can be sent both directions (Device-to-Host and Host-to-Device). Below is an example HID Report Descriptor that describes the Mouse_Input_Report structure defined previously. In this example HID Report Descriptor the entries highlighted in blue define the unsigned char button element as an 8-bit bit-field where the least significant 3-bits are the three mouse buttons, and the remaining 5-bits are a constant. The entries highlighted in green define the signed char x and signed char y elements of the report. For additional information about what the various HID Report Descriptor identifiers are and how they are used please refer to the USB HID 1.11 specification.

```
static const unsigned char usb_hid_mouse_report_descriptor[] =
{
    0x05, 0x01,                  /* USAGE_PAGE (Generic Desktop)      */
    0x09, 0x02,                  /* USAGE (Mouse)                     */
    0xa1, 0x01,                  /* COLLECTION (Application)          */
    0x09, 0x01,                  /*   USAGE (Pointer)                 */
    0xa1, 0x00,                  /*   COLLECTION (Physical)           */
    0x05, 0x09,                  /*     USAGE_PAGE (Button)           */
    0x19, 0x01,                  /*     USAGE_MINIMUM (Button 1)      */
    0x29, 0x03,                  /*     USAGE_MAXIMUM (Button 3)      */
    0x15, 0x00,                  /*     LOGICAL_MINIMUM (0)           */
    0x25, 0x01,                  /*     LOGICAL_MAXIMUM (1)           */
    0x95, 0x03,                  /*     REPORT_COUNT (3)              */
    0x75, 0x01,                  /*     REPORT_SIZE (1)               */
    0x81, 0x02,                  /*     INPUT (Data,Var,Abs)          */
    0x95, 0x01,                  /*     REPORT_COUNT (1)              */
    0x75, 0x05,                  /*     REPORT_SIZE (5)               */
```

```
    0x81, 0x03,                    /*      INPUT (Cnst,Var,Abs)           */
    0x05, 0x01,                    /*      USAGE_PAGE (Generic Desktop)   */
    0x09, 0x30,                    /*      USAGE (X)                      */
    0x09, 0x31,                    /*      USAGE (Y)                      */
    0x15, 0x81,                    /*      LOGICAL_MINIMUM (-127)         */
    0x25, 0x7f,                    /*      LOGICAL_MAXIMUM (127)          */
    0x75, 0x08,                    /*      REPORT_SIZE (8)                */
    0x95, 0x02,                    /*      REPORT_COUNT (2)               */
    0x81, 0x06,                    /*      INPUT (Data,Var,Rel)           */
    0xc0,                          /*    END_COLLECTION                   */
    0xc0                           /*  END_COLLECTION                     */
};
```

HID Interrupt IN Endpoint

The HID protocol requires all Human Interface Devices include a Interrupt IN endpoint which is used to report when a INPUT or FEATURE report value changes. For the above mouse example this means the Mouse_Input_Report structure will be sent to the Host over the Interrupt IN endpoint anytime the the button_state, x or y values change.

HID Control Endpoint Requests

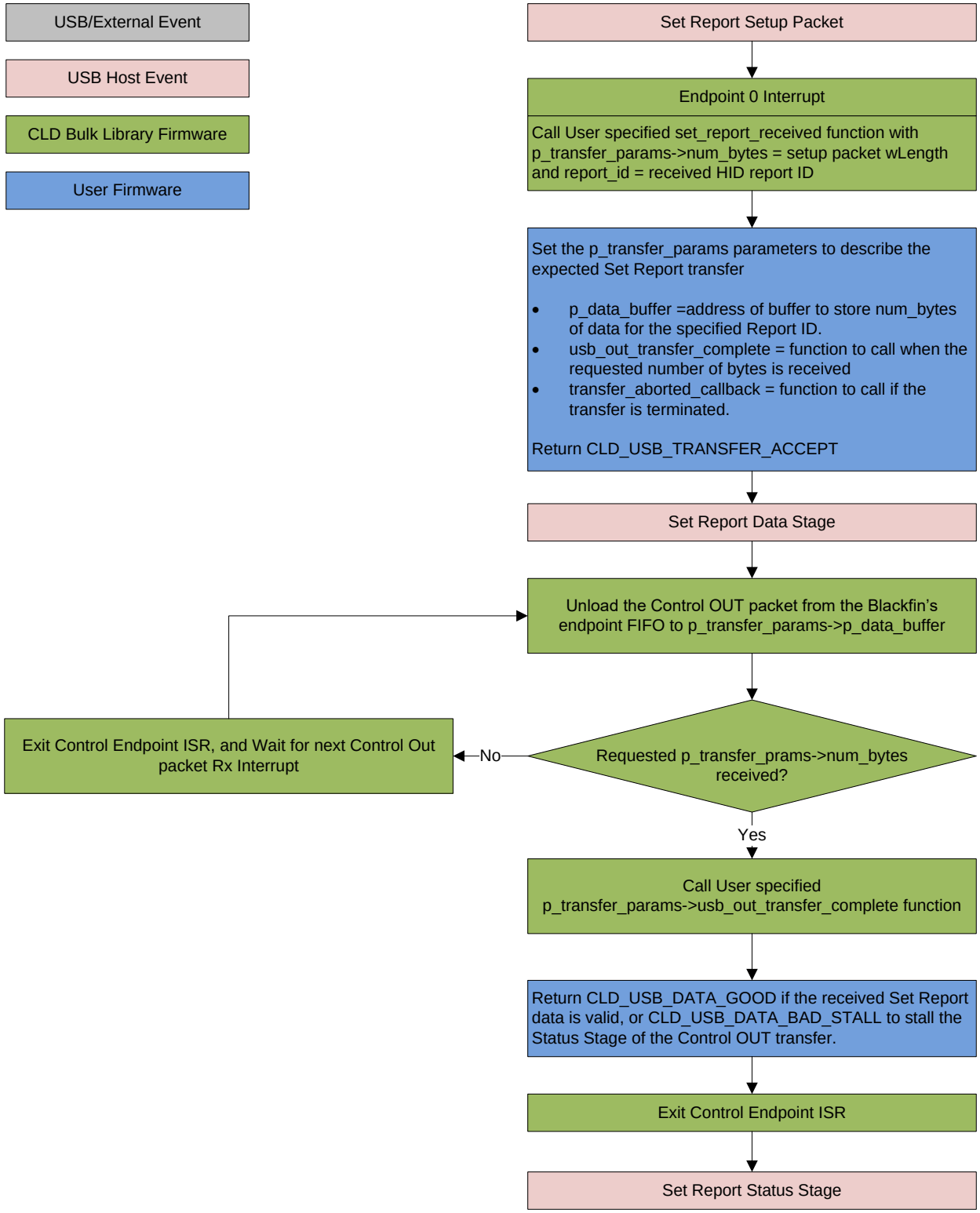
The HID protocol defines several Control Endpoint requests that a HID peripheral is required to support as well as some optional Control Endpoint requests. The Control Endpoint requests used by the CLD BF70x HID Library are explained in the following sections, and include flow charts showing how the CLD BF70x HID Library and the User firmware interact to the Control Endpoint requests.

Additionally, the User firmware code snippets included at the end of this document provide a basic framework for implementing the HID control requests using the CLD BF70x HID Library.

Set Report (required)

Set Report is a Control OUT request and is used by the Host to send data to the device using one of the Device's OUTPUT or FEATURE Reports

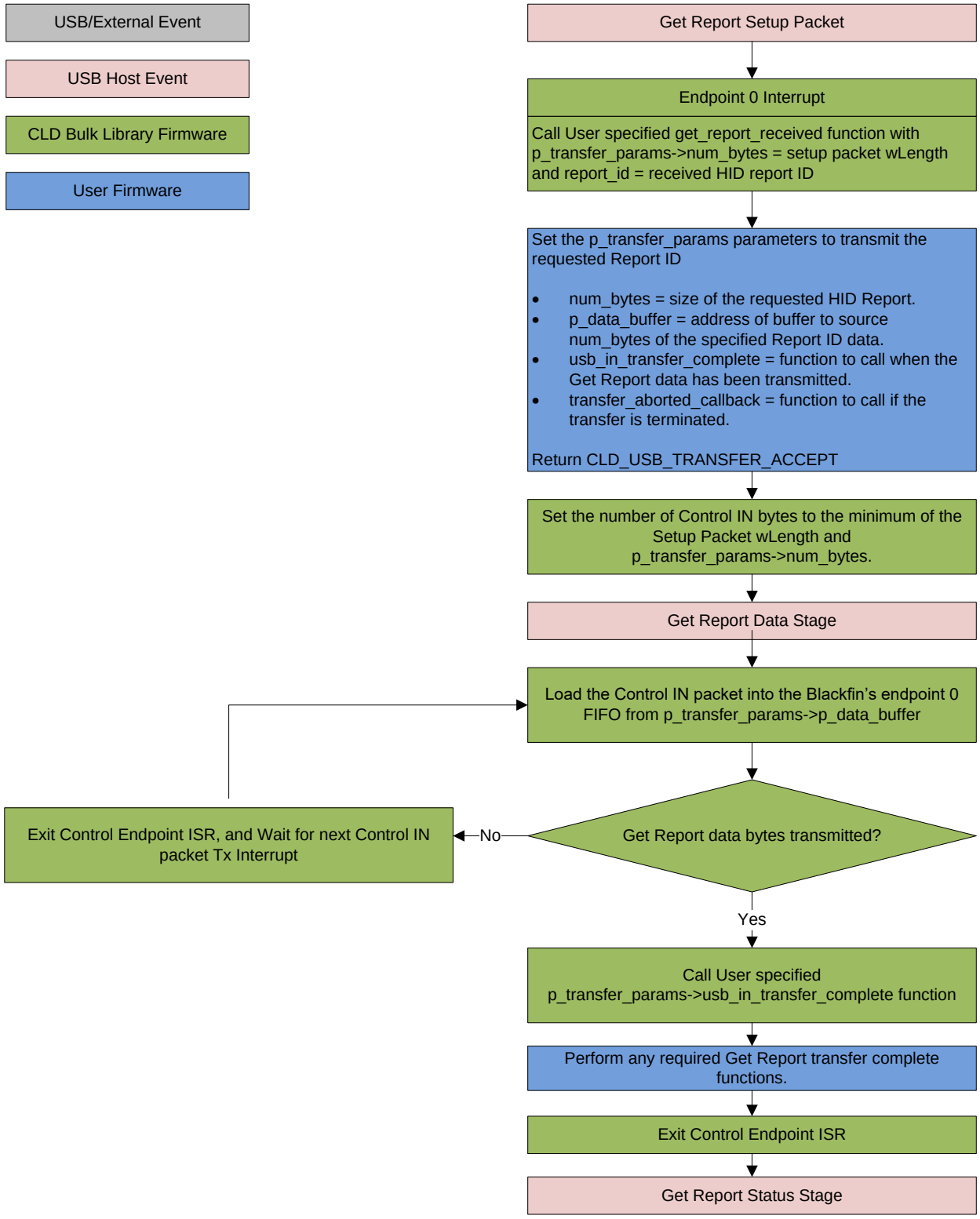
CLD BF70x HID Library Set Report Flow Chart



Get Report (optional)

Get Report is a Control IN request used by the Host to request the current state of one of the Device's INPUT or FEATURE Reports.

CLD BF70x HID Library Get Report Flow Chart

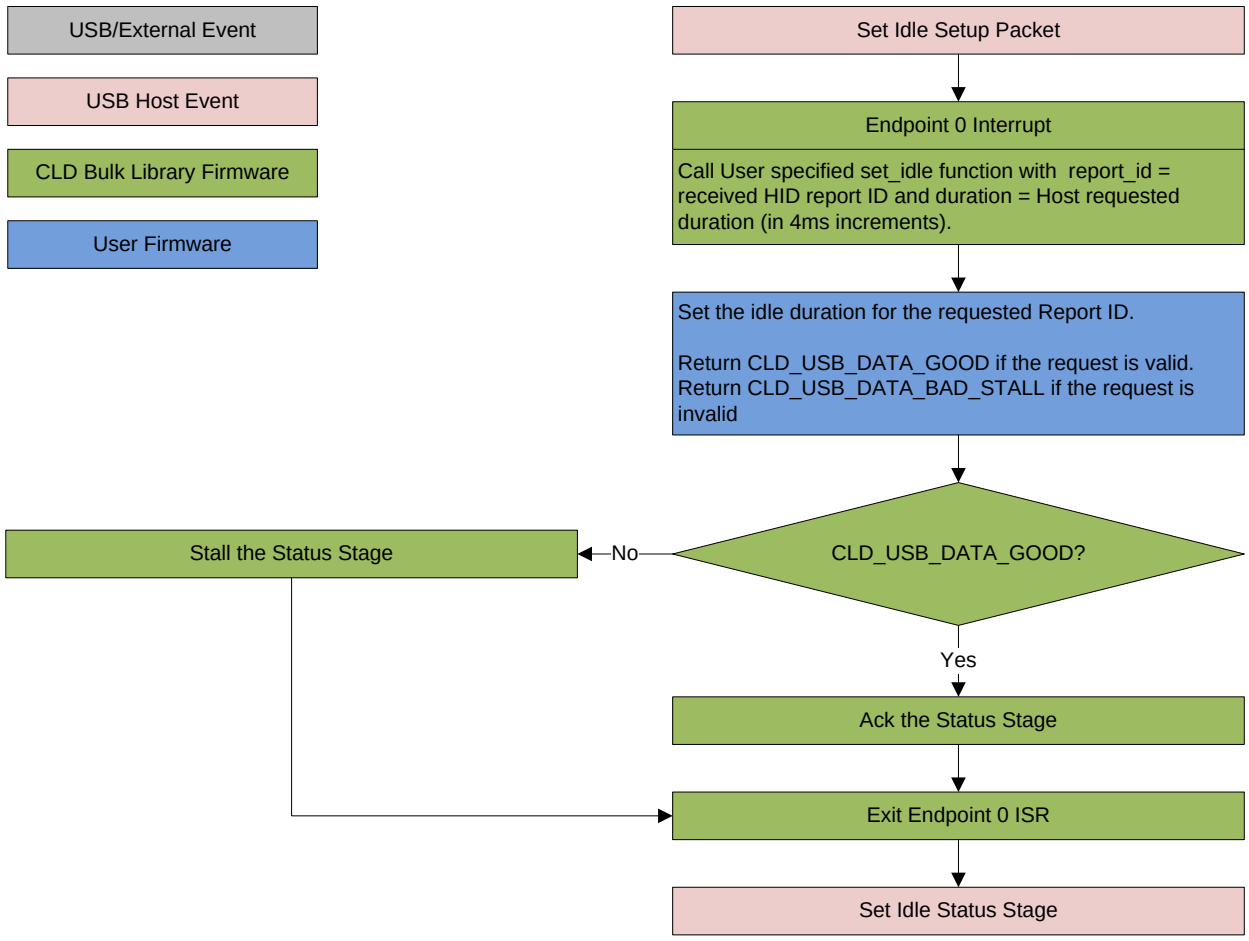


Set Idle (optional)

The Set Idle Control OUT request is used by the Host to specify the amount of time before the device will resend the current state of specified Report over the Interrupt IN endpoint when the reported data hasn't changed. The Set Idle duration is specified in 4 millisecond increments, where setting the duration to 0 tells the Device to only send the specified Report when it's data changes.

For example if the Host uses the Set Idle command and specifies a duration of 500ms the device is required to send the specified Report as soon as possible when the Report data changes, and every 500ms while the Report data remains constant.

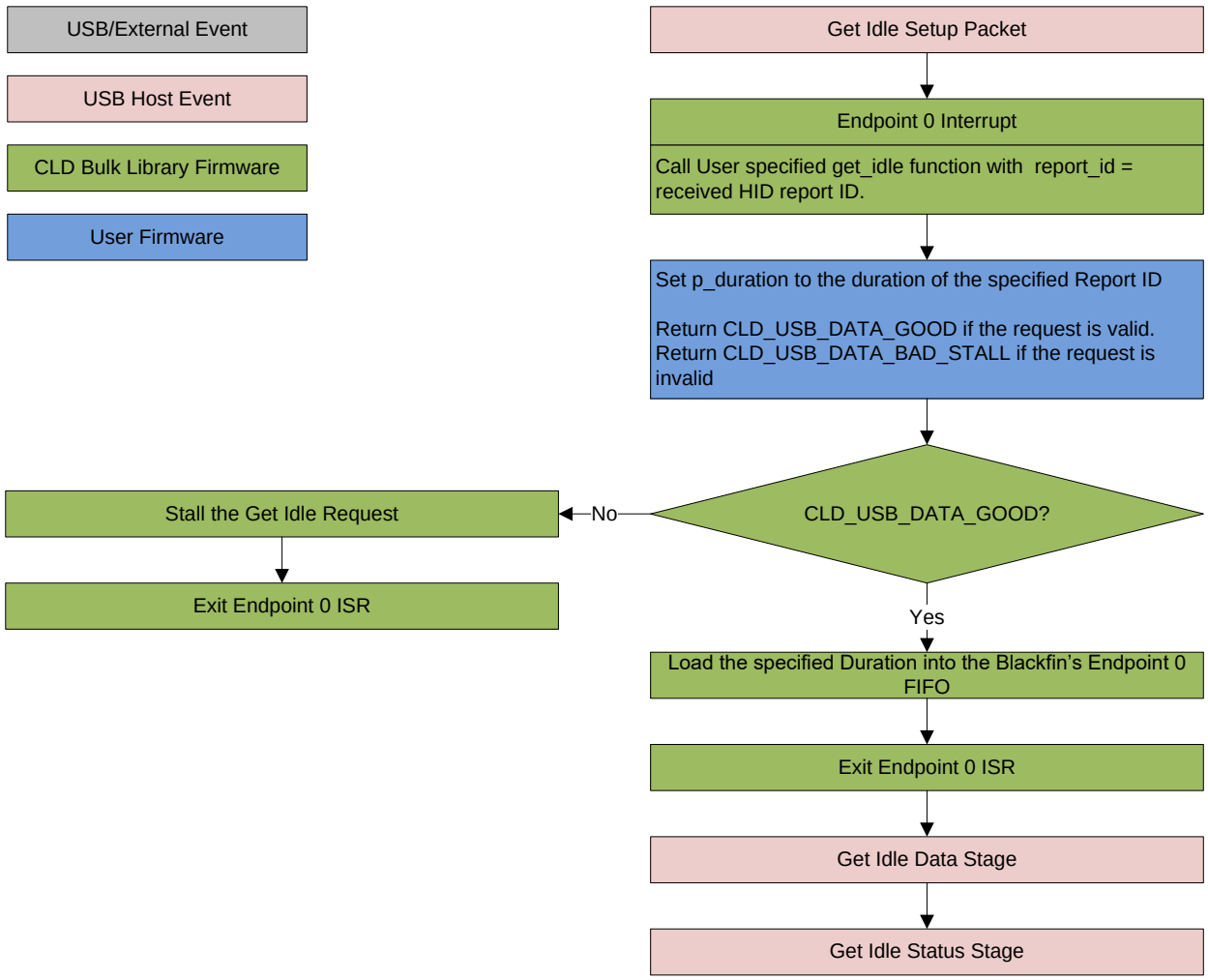
CLD BF70x HID Library Set Idle Flow Chart



Get Idle (optional)

The Get Idle Control IN request is used by the Host to get the current idle duration of the Report specified in the Get Idle request.

CLD BF70x HID Library Get Idle Flow Chart



Optional HID Interrupt OUT Endpoint

The USB HID Protocol includes an optional Interrupt OUT endpoint. When a Human Interface Device includes the Interrupt OUT endpoint the Host will use this endpoint to transmit OUTPUT Report data instead of using the Set Report Request.

Dependencies

In order to function properly the CLD BF70x HID Library requires the following Blackfin resources:

- 24Mhz clock input connected to the Blackfin USB0_CLKIN pin.
- Optionally the CLD BF70x HID Library can use one of the Blackfin UARTs to implement a serial console interface.
- The User firmware is responsible for setting up the Blackfin clocks, as well as enabling the Blackfin's System Event Controller (SEC) and configuring SEC Core Interface (SCI) interrupts to be sent to the Blackfin core.

Memory Footprint

The CLD BF70x HID Library approximate memory footprint is as follows:

Code memory:	26304 bytes
Data memory:	4404 bytes
Total:	30708 bytes or 29.98k
Heap memory:	1152 bytes (only malloc'ed if optional cld_console is enabled)

Note: The CLD BF70x HID Library is currently optimized for speed (not space).

CLD BF70x HID Library Scope and Intended Use

The CLD BF70x HID Library implements a USB Human Interface Device Class device, as well as providing time measurements and optional bi-directional UART console functionality. The CLD BF70x HID Library is designed to be added to an existing User project, and as such only includes the functionality needed to implement the above mentioned USB, timer and UART console features. All other aspects of Blackfin processor configuration must be implemented by the User code.

CLD HID Mouse Example v2.0 Description

The cld_hid_mouse_example_v2_0 project provided with the CLD BF70x HID Library implements a basic HID Mouse using the ADSP-BF707 EZ-Board. This example uses the EZ-Board's push buttons to generate mouse events that get reported to the Host using the CLD BF70x HID Library. This example is not indented to be a used as a complete stand alone project. Instead, this project only includes the User functionality required to create a basic USB mouse, and it is up to the User to include their own custom system initialization and any extra functionality they require.

CLD BF70x HID Library API

The following CLD library API descriptions include callback functions that are called by the library based on USB events. The following color code is used to identify if the callback function is called from the USB interrupt service routine, or from mainline. The callback functions called from the USB interrupt service routine are also italicized so they can be identified when printed in black and white.

Callback called from the mainline context
Callback called from the USB interrupt service routine

cld_bf70x_hid_lib_init

```
CLD_RV cld_bf70x_hid_lib_init (CLD_BF70x_HID_Lib_Init_Params *
cld_hid_lib_params)
```

Initialize the CLD BF70x HID Library.

Arguments

cld_hid_lib_params	Pointer to a CLD_BF70x_HID_Lib_Init_Params structure that has been initialized with the User Application specific data.

Return Value

This function returns the CLD_RV type which represents the status of the CLD BF70x HID initialization process. The CLD_RV type has the following values:

CLD_SUCCESS	The library was initialized successfully
CLD_FAIL	There was a problem initializing the library
CLD_ONGOING	The library initialization is being processed

Details

The cld_bf70x_hid_lib_init function is called as part of the device initialization and must be repeatedly called until the function returns CLD_SUCCESS or CLD_FAIL. If CLD_FAIL is returned the library will output an error message identifying the cause of the failure using the cld_console UART if enabled by the User application. Once the library has been initialized successfully the main program loop can start.

The CLD_BF70x_HID_Lib_Init_Params structure is described below:

```
typedef struct
{
    CLD_Uart_Num uart_num;
    unsigned long uart_baud;
    unsigned long sclk0;
    void (*fp_console_rx_byte) (unsigned char byte);

    unsigned short vendor_id;
    unsigned short product_id;
```

```
    unsigned short report_descriptor_size
    unsigned char * p_report_descriptor

    CLD_HID_Endpoint_Params * p_interrupt_in_endpoint_params;

    CLD_HID_Endpoint_Params * p_interrupt_out_endpoint_params;
    CLD_USB_Transfer_Request_Return_Type (*fp_interrupt_out_data_received)
                              (CLD_USB_Transfer_Params * p_transfer_data);

    unsigned char usb_bus_max_power;

    unsigned short device_descriptor_bcdDevice;

    const char * p_usb_string_manufacturer;
    const char * p_usb_string_product;
    const char * p_usb_string_serial_number;
    const char * p_usb_string_configuration;
    const char * p_usb_string_interface;

    unsigned short usb_string_language_id;

    CLD_USB_Transfer_Request_Return_Type (*fp_set_report_received) (unsigned
            char report_id, CLD_USB_Transfer_Params * p_transfer_data);

    CLD_USB_Transfer_Request_Return_Type (*fp_get_report_received) (unsigned
            char report_id, CLD_USB_Transfer_Params * p_transfer_data);

    CLD_USB_Data_Received_Return_Type (*fp_set_idle) (unsigned char
            report_id, unsigned char duration);

    CLD_USB_Data_Received_Return_Type (*fp_get_idle) (unsigned char
            report_id, unsigned char * p_duration);

    void (*fp_cld_usb_event_callback) (CLD_USB_Event event);
    void (*fp_cld_lib_status) (unsigned short status_code,
                               void * p_additional_data,
                               unsigned short additional_data_size);
} CLD_BF70x_HID_Lib_Init_Params;
```

A description of the CLD_BF70x_HID_Lib_Init_Params structure elements is included below:

Structure Element	Description
uart_num	Identifies which of the ADSP-BF707 UARTs should be used by the CLD BF70x HID Library to implement the cld_console (refer to the cld_console API description for additional information). The valid uart_num values are listed below: CLD_UART_0 CLD_UART_1 CLD_UART_DISABLE If uart_num is set to CLD_UART_DISABLE the CLD BF70x HID Library will not use a UART, and the cld_console functionality is disabled.
uart_baud	Sets the desired UART baud rate used for the cld_console. The remaining cld_console UART parameters are as follows:

<table>
<tr><td></td><td>Number of data bits: 8
Number of stop bits: 1
No Parity
No Hardware Flow Control</td></tr>
<tr><td>sclk0</td><td>Used to tell the CLD BF70x HID Library the frequency of the ADSP_BF707 SCLK0 clock.</td></tr>
<tr><td>fp_console_rx_byte</td><td>Pointer to the function that is called when a byte is received by the cld_console UART. This function has a single parameter ('byte') which is the value received by the UART.
Note: Set to NULL if not required by application</td></tr>
<tr><td>vendor_id</td><td>The 16-bit USB vendor ID returned to the USB Host in the USB Device Descriptor.
USB Vendor ID's are assigned by the USB-IF and can be purchased through their website (www.usb.org).</td></tr>
<tr><td>product_id</td><td>The 16-bit product ID returned to the USB Host in the USB Device Descriptor.</td></tr>
<tr><td>report_descriptor_size</td><td>The size of the User defined HID Report Descriptor.</td></tr>
<tr><td>p_report_descriptor</td><td>Pointer to the User defined HID Report Descriptor.</td></tr>
<tr><td>p_interrupt_in_endpoint_params</td><td>Pointer to a CLD_HID_Endpoint_Params structure that describes how the Interrupt IN endpoint should be configured. The CLD_HID_Endpoint_Params structure contains the following elements:

<table>
<tr><th>Structure Element</th><th>Description</th></tr>
<tr><td>endpoint_num</td><td>Sets the USB endpoint number of the Interrupt endpoint. The endpoint number must be within the following range: 1 ≤ endpoint_num ≤ 12. Any other endpoint number will result in the cld_bf70x_hid_lib_init function returning CLD_FAIL</td></tr>
<tr><td>max_packet_size_full_speed</td><td>Sets the Interrupt endpoint's max packet size when operating at Full Speed. The maximum max packet size is 64 bytes.</td></tr>
<tr><td>polling_interval_full_speed</td><td>Full-Speed polling interval in the USB Endpoint Descriptor. (See USB 2.0 section 9.6.6)</td></tr>
<tr><td>max_packet_size_high_speed</td><td>Sets the Interrupt endpoint's max packet size when operating at High Speed. The maximum max packet size 1024 bytes.</td></tr>
<tr><td>polling_interval_high_speed</td><td>High-Speed polling interval in the USB Endpoint Descriptor. (See USB 2.0 section 9.6.6)</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td>p_interrupt_out_endpoint_params</td><td>Pointer to a CLD_HID_Endpoint_Params structure that describes how the Interrupt Out endpoint should be configured. Refer to the p_interrupt_in_endpoint_params description for information about the CLD_HID_Endpoint_Params structure.

Set to CLD_NULL if the optional Interrupt OUT endpoint isn't used.</td></tr>
<tr><td>fp_interrupt_out_data_received</td><td>Pointer to the function that is called when the Interrupt OUT endpoint receives data. This function takes a pointer to the CLD_USB_Transfer_Params structure ('p_transfer_data')as a parameter.

The following CLD_USB_Transfer_Params structure elements are used to processed a Interrupt OUT transfer:

<table>
<tr><td>Structure Element</td><td>Description</td></tr>
<tr><td>num_bytes</td><td>The number of bytes to transfer to the p_data_buffer before calling the usb_out_transfer_complete callback function.

When the fp_interrupt_out_data_received function is called num_bytes is set the number of bytes in the current Interrupt OUT packet. If the Interrupt OUT total transfer size is known num_bytes can be set to the total transfer size, and the CLD BF70x HID Library will complete the entire transfer without calling fp_interrupt_out_data_received again. If num_bytes isn't modified the fp_interrupt_out_data_received function will be called for each Interrupt OUT packet.</td></tr>
<tr><td>p_data_buffer</td><td>Pointer to the data buffer to store the received Interrupt OUT data. The size of the buffer should be greater than or equal to the value in num_bytes.</td></tr>
<tr><td>fp_usb_out_transfer_complete</td><td>Function called when num_bytes of data has been transferred to the p_data_buffer memory.</td></tr>
<tr><td>fp_transfer_aborted_callback</td><td>Function called if there is a problem transferring the</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td></td><td>
<table>
<tr><td></td><td>requested Interrupt OUT data.</td></tr>
<tr><td>transfer_timeout_ms</td><td>Interrupt OUT transfer timeout in milliseconds. If the Interrupt out transfer takes longer then this timeout the transfer is aborted and the transfer_aborted_callback is called.
Setting the timeout to 0 disables the timeout</td></tr>
</table>

The interrupt_out_data_received function returns the CLD_USB_Transfer_Request_Return_Type, which has the following values:
<table>
<tr><td>Return Value</td><td>Description</td></tr>
<tr><td>CLD_USB_TRANSFER_ACCEPT</td><td>Notifies the CLD BF70x HID Library that the Interrupt OUT data should be accepted using the p_transfer_data values.</td></tr>
<tr><td>CLD_USB_TRANSFER_PAUSE</td><td>Requests that the CLD BF70x HID Library pause the current transfer. This causes the Interrupt OUT endpoint to be nak'ed until the transfer is resumed by calling cld_bf70x_hid_lib_resume_ paused_interrupt_out_transfer.</td></tr>
<tr><td>CLD_USB_TRANSFER_DISCARD</td><td>Requests that the CLD BF70x HID Library discard the number of bytes specified in p_transfer_params-> num_bytes. In this case the library accepts the Interrupt OUT data from the USB Host but discards the data. This is similar to the concepts of frame dropping in audio/video applications.</td></tr>
<tr><td>CLD_USB_TRANSFER_STALL</td><td>This notifies the CLD BF70x HID Library that there is an error and the Interrupt OUT endpoint should be stalled.</td></tr>
</table>
</td></tr>
<tr><td>usb_bus_max_power</td><td>USB Configuration Descriptor bMaxPower value (0 = self powered). Refer to the USB 2.0 protocol section 9.6.3.</td></tr>
<tr><td>device_descriptor_bcd_device</td><td>USB Device Descriptor bcdDevice value.
Refer to the USB 2.0 protocol section 9.6.1.</td></tr>
<tr><td>p_usb_string_manufacturer</td><td>Pointer to the null-terminated string. This string is used by the CLD BF70x HID Library to generate the Manufacturer USB String Descriptor. If the Manufacturer String Descriptor is not used set p_usb_string_manufacturer to NULL.</td></tr>
</table>

<table>
<tr><td>p_usb_string_product</td><td>Pointer to the null-terminated string. This string is used by the CLD BF70x HID Library to generate the Product USB String Descriptor. If the Product String Descriptor is not used set p_usb_string_product to NULL.</td></tr>
<tr><td>p_usb_string_serial_number</td><td>Pointer to the null-terminated string. This string is used by the CLD BF70x HID Library to generate the Serial Number USB String Descriptor. If the Serial Number String Descriptor is not used set p_usb_string_serial_number to NULL.</td></tr>
<tr><td>p_usb_string_configuration</td><td>Pointer to the null-terminated string. This string is used by the CLD BF70x HID Library to generate the Configuration USB String Descriptor. If the Configuration String Descriptor is not used set p_usb_string_configuration to NULL.</td></tr>
<tr><td>p_usb_string_interface</td><td>Pointer to the null-terminated string. This string is used by the CLD BF70x HID Library to generate the Interface 0 USB String Descriptor. If the Product String Descriptor is not used set p_usb_string_interface to NULL.</td></tr>
<tr><td>usb_string_language_id</td><td>16-bit USB String Descriptor Language ID Code as defined in the USB Language Identifiers (LANGIDs) document (www.usb.org/developers/docs/USB_LANGIDs.pdf).
0x0409 = English (United States)</td></tr>
<tr><td>fp_set_report_received</td><td>Pointer to the function that is called when a HID Set Report request is received. This function takes the requests Report ID and a pointer to the CLD_USB_Transfer_Params structure ('p_transfer_data') as its parameters.

The following CLD_USB_Transfer_Params structure elements are used to processed a Set Report transfer:

<table>
<tr><td>Structure Element</td><td>Description</td></tr>
<tr><td>num_bytes</td><td>The number of bytes from the Setup Packet wLength field, which is the number of bytes that will be transferred to p_data_buffer before calling the fp_usb_out_transfer_complete callback function.</td></tr>
<tr><td>p_data_buffer</td><td>Pointer to the data buffer to store the Set Report data. The size of the buffer should be greater than or equal to the value in num_bytes.</td></tr>
<tr><td>fp_usb_out_transfer_compelete</td><td>Function called when num_bytes of data has been written to the p_data_buffer memory.</td></tr>
<tr><td>fp_transfer_aborted_callback</td><td>Function called if there is a problem Set Report data.</td></tr>
<tr><td>transfer_timeout_ms</td><td>Not used.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td></td><td>The set_report_received function returns the CLD_USB_Transfer_Request_Return_Type, which has the following values:
<table>
<tr><th>Return Value</th><th>Description</th></tr>
<tr><td>CLD_USB_TRANSFER_ACCEPT</td><td>Notifies the CLD BF70x HID Library that the Set Report data should be accepted using the p_transfer_data values.</td></tr>
<tr><td>CLD_USB_TRANSFER_PAUSE</td><td>Requests that the CLD BF70x HID Library pause the Set Report transfer. This causes the Control Endpoint to be nak'ed until the transfer is resumed by calling cld_bf70x_hid_lib_resume_ paused_control_transfer.</td></tr>
<tr><td>CLD_USB_TRANSFER_DISCARD</td><td>Requests that the CLD BF70x HID Library discard the number of bytes specified in p_transfer_params-> num_bytes. In this case the library accepts the Set Report data from the USB Host but discards the data. This is similar to the concepts of frame dropping in audio/video applications.</td></tr>
<tr><td>CLD_USB_TRANSFER_STALL</td><td>This notifies the CLD BF70x HID Library that there is an error and the Set Report request should be stalled.</td></tr>
</table></td></tr>
<tr><td>fp_get_report_received</td><td>Pointer to the function that is called when a HID Get Report request is received. This function takes the requests Report ID and a pointer to the CLD_USB_Transfer_Params structure ('p_transfer_data') as its parameters.

The following CLD_USB_Transfer_Params structure elements are used to processed a Get Report request:
<table>
<tr><th>Structure Element</th><th>Description</th></tr>
<tr><td>num_bytes</td><td>The number of bytes from the Setup Packet wLength field. The User firmware sets num_bytes to the size of the requested Report ID.</td></tr>
<tr><td>p_data_buffer</td><td>Pointer to the data buffer to source the Get Report data. The size of the buffer should be greater than or equal to the value in num_bytes.</td></tr>
</table></td></tr>
</table>

<table>
<tr><td></td><td>
<table>
<tr><td>fp_usb_in_transfer_compelete</td><td>Function called when Get Report data has been transferred to the Host.</td></tr>
<tr><td>fp_transfer_aborted_callback</td><td>Function called if there is a problem transferring the Get Report data.</td></tr>
<tr><td>transfer_timeout_ms</td><td>Not used</td></tr>
</table>

The get_report_received function returns the CLD_USB_Transfer_Request_Return_Type, which has the following values:
<table>
<tr><td>Return Value</td><td>Description</td></tr>
<tr><td>CLD_USB_TRANSFER_ACCEPT</td><td>Notifies the CLD BF70x HID Library that the Get Report data should be transferred using the p_transfer_data values.</td></tr>
<tr><td>CLD_USB_TRANSFER_PAUSE</td><td>Requests that the CLD BF70x HID Library pause the Get Report transfer. This causes the Control Endpoint to be nak'ed until the transfer is resumed by calling cld_bf70x_hid_lib_resume_ paused_control_transfer.</td></tr>
<tr><td>CLD_USB_TRANSFER_DISCARD</td><td>Requests that the CLD BF70x HID Library to return a zero length packet in response to the Get Report request.</td></tr>
<tr><td>CLD_USB_TRANSFER_STALL</td><td>This notifies the CLD BF70x HID Library that there is an error and the Get Report request should be stalled.</td></tr>
</table>
</td></tr>
<tr><td>fp_set_idle</td><td>Pointer to the function that is called when a HID Set Idle request is received. This function takes the request's Report ID and requested duration as its parameters. The duration is specified in 4ms increments.

The set_idle function returns the CLD_USB_Data_Received_Return_Type, which has the following values:
<table>
<tr><td>Return Value</td><td>Description</td></tr>
<tr><td>CLD_USB_DATA_GOOD</td><td>Notifies the CLD BF70x HID Library that the Set Idle request is valid.</td></tr>
<tr><td>CLD_USB_DATA_BAD_STALL</td><td>Notifies the CLD BF70x HID Library that the Set Idle request is invalid, and should be stalled.</td></tr>
</table>
</td></tr>
<tr><td>fp_get_idle</td><td>Pointer to the function that is called when a HID Get Idle request is</td></tr>
</table>

<table>
<tr><td></td><td>received. This function takes the request's Report ID a pointer, p_duration as its parameters. p_duration should be set to the requested Report ID's duration in 4ms increments.

The get_idle function returns the CLD_USB_Data_Received_Return_Type, which has the following values:
<table>
<tr><th>Return Value</th><th>Description</th></tr>
<tr><td>CLD_USB_DATA_GOOD</td><td>Notifies the CLD BF70x HID Library that the Get Idle request is valid and the p_duration value should be returned to the Host.</td></tr>
<tr><td>CLD_USB_DATA_BAD_STALL</td><td>Notifies the CLD BF70x HID Library that the Get Idle request is invalid, and should be stalled.</td></tr>
</table></td></tr>
<tr><td>fp_cld_usb_event_callback</td><td>Function that is called when one of the following USB events occurs. This function has a single CLD_USB_Event parameter.

Note: This callback can be called from the USB interrupt or mainline context depending on which USB event was detected. The CLD_USB_Event values in the table below are highlighted to show the context the callback is called for each event.

The CLD_USB_Event has the following values:
<table>
<tr><th>Return Value</th><th>Description</th></tr>
<tr><td>CLD_USB_CABLE_CONNECTED</td><td>USB Cable Connected.</td></tr>
<tr><td>CLD_USB_CABLE_DISCONNECTED</td><td>USB Cable Disconnected</td></tr>
<tr><td>CLD_USB_ENUMERATED_CONFIGURED_HS</td><td>USB device enumerated at High-Speed (USB Configuration set to a non-zero value)</td></tr>
<tr><td>CLD_USB_ENUMERATED_CONFIGURED_FS</td><td>USB device enumerated at Full-Speed (USB Configuration set to a non-zero value)</td></tr>
<tr><td>CLD_USB_UN_CONFIGURED</td><td>USB Configuration set to 0</td></tr>
<tr><td>CLD_USB_BUS_RESET</td><td>USB Bus reset received</td></tr>
<tr><td>CLD_USB_BUS_SUSPEND</td><td>USB Suspend detected</td></tr>
<tr><td>CLD_USB_BUS_RESUME</td><td>USB Resume detected</td></tr>
</table>
Note: Set to CLD_NULL if not required by application</td></tr>
<tr><td>fp_cld_lib_status</td><td>Pointer to the function that is called when the CLD library has a status to report. This function has the following parameters:

<table>
<tr><th>Parameter</th><th>Description</th></tr>
</table></td></tr>
</table>

	status_code	16-bit status code. If the most significant bit is a '1' the status being reported is an Error.
	p_additional_data	Pointer to additional data included with the status.
	additional_data_size	The number of bytes in the specified additional data.
	If the User plans on processing outside of the fp_cld_lib_status function they will need to copy the additional data to a User buffer.	

cld_bf70x_hid_lib_main

```
void cld_bf70x_hid_lib_main (void)
```

CLD BF70x HID Library mainline function

Arguments

None

Return Value

None.

Details

The cld_bf70x_hid_lib_main function is the CLD BF70x HID Library mainline function which must be called in every iteration of the main program loop in order for the library to function properly.

cld_bf70x_hid_lib_transmit_interrupt_in_data

```
CLD_USB_Data_Transmit_Return_Type
cld_bf70x_hid_lib_transmit_interrupt_in_data    (CLD_USB_Transfer_Params *
p_transfer_data)
```

CLD BF70x HID Library function used to send data over the Interrupt IN endpoint.

Arguments

p_transfer_data	Pointer to a CLD_USB_Transfer_Params structure used to describe the data being transmitted.

Return Value

This function returns the CLD_USB_Data_Transmit_Return_Type type which reports if the Interrupt IN transmission request was started. The CLD_USB_Data_Transmit_Return_Type type has the following values:

CLD_USB_TRANSMIT_SUCCESSFUL	The library has started the requested Interrupt IN transfer.
CLD_USB_TRANSMIT_FAILED	The library failed to start the requested Interrupt IN transfer. This will happen if the Interrupt IN endpoint is busy, or if the p_transfer_data-> data_buffer is set to NULL

Details

The cld_bf70x_hid_lib_transmit_interrupt_in_data function transmits the data specified by the p_transfer_data parameter to the USB Host using the Device's Interrupt IN endpoint.

The CLD_USB_Transfer_Params structure is described below.

```
typedef struct
{
    unsigned long num_bytes;
    unsigned char * p_data_buffer;
    union
    {
        CLD_USB_Data_Received_Return_Type (*usb_out_transfer_complete)(void);
        void (*usb_in_transfer_complete) (void);
    }callback;
    void (*transfer_aborted_callback) (void);
} CLD_USB_Transfer_Params;
```

A description of the CLD_USB_Transfer_Params structure elements is included below:

Structure Element	Description
num_bytes	The number of bytes to transfer to the USB Host. Once the specified number of bytes have been transmitted the usb_in_transfer_complete callback function will be called.
p_data_buffer	Pointer to the data to be sent to the USB Host. This buffer must include the number of bytes specified by num_bytes.
fp_usb_out_transfer_complete	Not Used for Interrupt IN transfers

fp_usb_in_transfer_complete	Function called when the specified data has been transmitted to the USB host. This function pointer can be set to NULL if the User application doesn't want to be notified when the data has been transferred.
fp_transfer_aborted_callback	Function called if there is a problem transmitting the data to the USB Host. This function can be set to NULL if the User application doesn't want to be notified if a problem occurs.
transfer_timeout_ms	Interrupt OUT transfer timeout in milliseconds. If the Interrupt out transfer takes longer then this timeout the transfer is aborted and the transfer_aborted_callback is called. Setting the timeout to 0 disables the timeout

cld_bf70x_hid_lib_resume_paused_interrupt_out_transfer

```
void cld_bf70x_hid_lib_resume_paused_interrupt_out_transfer (void)
```

CLD BF70x HID Library function used to resume a paused Interrupt OUT transfer.

Arguments

None

Return Value

None.

Details

The cld_bf70x_hid_lib_resume_paused_interrupt_out_transfer function is used to resume a Interrupt OUT transfer that was paused by the `fp_interrupt_out_data_received` function returning CLD_USB_TRANSFER_PAUSE. When called the cld_bf70x_hid_lib_resume_paused_interrupt_out_transfer function will call the User application's `fp_interrupt_out_data_received` function passing the CLD_USB_Transfer_Params of the original paused transfer. The `fp_interrupt_out_data_`received function can then chose to accept, discard, or stall the interrupt out request.

cld_lib_usb_connect

```
void cld_lib_usb_connect (void)
```

CLD BF70x HID Library function used to connect to the USB Host.

Arguments
None

Return Value
None.

Details
The cld_lib_usb_connect function is called after the CLD BF70x HID Library has been initialized to connect the USB device to the Host.

cld_lib_usb_disconnect

```
void cld_lib_usb_disconnect (void)
```

CLD BF70x HID Library function used to disconnect from the USB Host.

Arguments
None

Return Value
None.

Details
The cld_lib_usb_disconnect function is called after the CLD BF70x HID Library has been initialized to disconnect the USB device to the Host.

cld_time_125us_tick

```
void cld_time_125us_tick (void)
```

CLD library timer function that should be called once per 125 microseconds.

Arguments
None

Return Value
None.

Details
This function should be called once every 125 microseconds in order to the CLD to processed periodic events.

cld_usb_isr_callback

```
void cld_usb_isr_callback (void)
```

CLD library USB interrupt service routines

Arguments
None

Return Value
None.

Details
These USB ISR functions should be called from the corresponding USB Port Interrupt Service Routine as shown in the CLD provided example projects.

cld_console_tx_isr_callback

```
void cld_console_tx_isr_callback (void)
```

CLD library console UART transmit interrupt service routines

Arguments
None

Return Value
None.

Details
These transmit ISR functions should be called from the corresponding UART transmit Interrupt Service Routine as shown in the CLD provided example projects.

cld_console_rx_isr_callback

```
void cld_console_rx_isr_callback (void)
```

CLD library console UART receive interrupt service routines

Arguments
None

Return Value
None.

Details
These receive ISR functions should be called from the corresponding UART receive Interrupt Service Routine as shown in the CLD provided example projects.

cld_time_get

```
CLD_Time cld_time_get(void)
```

CLD BF70x HID Library function used to get the current CLD time.

Arguments
None

Return Value
The current CLD library time.

Details
The cld_time_get function is used in conjunction with the cld_time_passed_ms function to measure how much time has passed between the cld_time_get and the cld_time_passed_ms function calls.

cld_time_passed_ms

```
CLD_Time cld_time_passed_ms(CLD_Time time)
```

CLD BF70x HID Library function used to measure the amount of time that has passed.

Arguments

`time`	A CLD_Time value returned by a cld_time_get function call.

Return Value
The number of milliseconds that have passed since the cld_time_get function call that returned the CLD_Time value passed to the cld_time_passed_ms function.

Details
The cld_time_passed_ms function is used in conjunction with the cld_time_get function to measure how much time has passed between the cld_time_get and the cld_time_passed_ms function calls.

If a one millisecond resolution is granular enough for your needs, you can have a virtually unlimited number of timed events when using cld_time_get and cld_time_passed_ms.

cld_time_get_125us

```
CLD_Time cld_time_get_125us(void)
```

CLD library function used to get the current CLD time in 125 microsecond increments.

Arguments

None

Return Value

The current CLD library time.

Details

The cld_time_get_125us function is used in conjunction with the cld_time_passed_125us function to measure how much time has passed between the cld_time_get_125us and the cld_time_passed_125us function calls in 125 microsecond increments.

cld_time_passed_125us

```
CLD_Time cld_time_passed_125us(CLD_Time time)
```

CLD library function used to measure the amount of time that has passed in 125 microsecond increments.

Arguments

`time`	A CLD_Time value returned by a cld_time_get_125us function call.

Return Value

The number of 125microsecond increments that have passed since the cld_time_get_125us function call that returned the CLD_Time value passed to the cld_time_passed_125us function.

Details

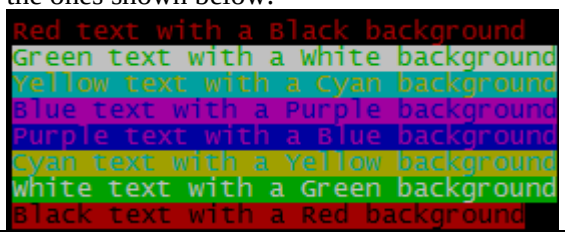
The cld_time_passed_125us function is used in conjunction with the cld_time_get_125us function to measure how much time has passed between the cld_time_get_125us and the cld_time_passed_125us function calls in 125 microsecond increments.

cld_console

```
CLD_RV cld_console(CLD_CONSOLE_COLOR foreground_color, CLD_CONSOLE_COLOR
      background_color, const char *fmt, ...)
```

CLD Library function that outputs a User defined message using the UART specified in the CLD_BF70x_HID_Lib_Init_Params structure.

Arguments

foreground_color	The CLD_CONSOLE_COLOR used for the console text. CLD_CONSOLE_BLACK CLD_CONSOLE_RED CLD_CONSOLE_GREEN CLD_CONSOLE_YELLOW CLD_CONSOLE_BLUE CLD_CONSOLE_PURPLE CLD_CONSOLE_CYAN CLD_CONSOLE_WHITE
background_color	The CLD_CONSOLE_COLOR used for the console background. CLD_CONSOLE_BLACK CLD_CONSOLE_RED CLD_CONSOLE_GREEN CLD_CONSOLE_YELLOW CLD_CONSOLE_BLUE CLD_CONSOLE_PURPLE CLD_CONSOLE_CYAN CLD_CONSOLE_WHITE The foreground and background colors allow the User to generate various color combinations like the ones shown below: 
fmt	The User defined ASCII message that uses the same format specifies as the printf function.
...	Optional list of additional arguments

Return Value

This function returns whether or not the specified message has been added to the cld_console transmit buffer.

CLD_SUCCESS	The message was added successfully.
CLD_FAIL	The message was not added, so the message will not be transmitted. This will occur if the CLD Console is disabled, or if the message will not fit into the transmit buffer.

Details

cld_console is similar in format to printf, and also natively supports setting a foreground and background color. A feature of cld_console is that it is non-blocking, i.e. long messages can be queued and the function call returns prior to the message draining from the buffer. Overly long messages are truncated to 128 bytes, and up to 1024 characters can be in escrow to be transmitted. Received characters can be processed by supplying a console_rx_byte function in the library init structure.

The following will output 'The quick brown fox' on a black background with green text:

```
cld_console(CLD_CONSOLE_GREEN, CLD_CONSOLE_BLACK, "The quick brown %s\n\r", "fox");
```

cld_lib_status_decode

```
char * cld_lib_status_decode (unsigned short status_cod,
                              void * p_additional_data,
                              unsigned short additional_data_size)
```

CLD Library function that returns a NULL terminated string describing the status passed to the function.

Arguments

status_code	16-bit status code returned by the CLD library. Note: If the most significant bit is a '1' the status is an error.
p_additional_data	Pointer to the additional data returned by the CLD library (if any).
additional_data_size	Size of the additional data returned by the CLD library.

Return Value

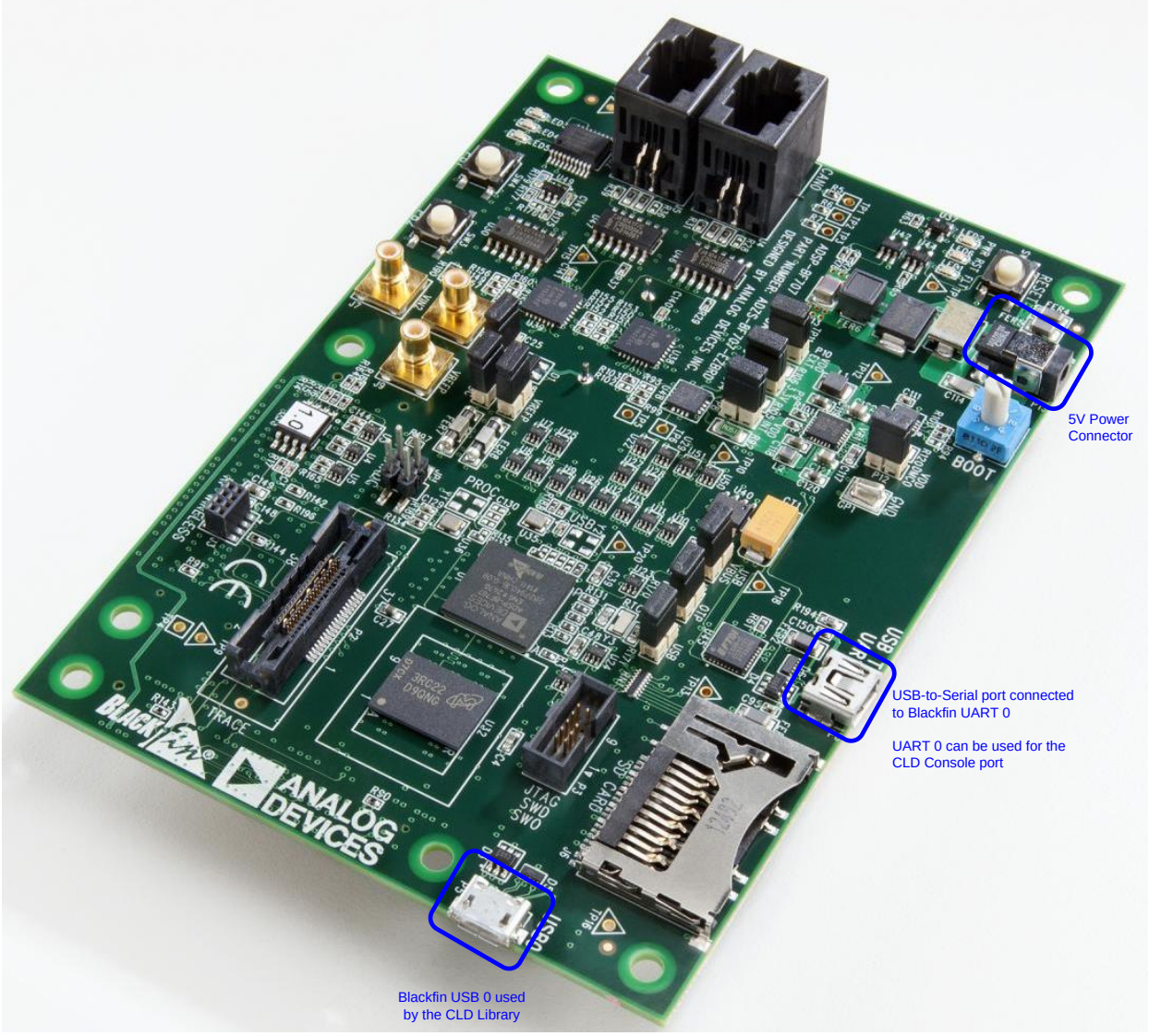
This function returns a decoded Null terminated ASCII string.

Details

The cld_lib_status_decode function can be used to generate an ASCII string which describes the CLD library status passed to the function. The resulting string can be used by the User to determine the meaning of the status codes returned by the CLD library.

Using the ADSP-BF707 Ez-Board

Connections:



Note about using UART0 and the FTDI USB to Serial Converter

On the ADSP-BF707 Ez-Board the Blackfin's UART0 serial port is connected to a FTDI FT232RQ USB-to-Serial converter. By default the UART 0 signals are connected to the FTDI chip. However, the demo program shipped on the Ez-Board disables the UART0 to FTDI connection. If the FTDI converter is used for the CLD BF70x HID Library console change the boot selection switch (located next to the power connector) so the demo program doesn't boot. Once this is done the FTDI USB-to-Serial converter can be used with the CLD BF70x HID Library console connected to UART0.

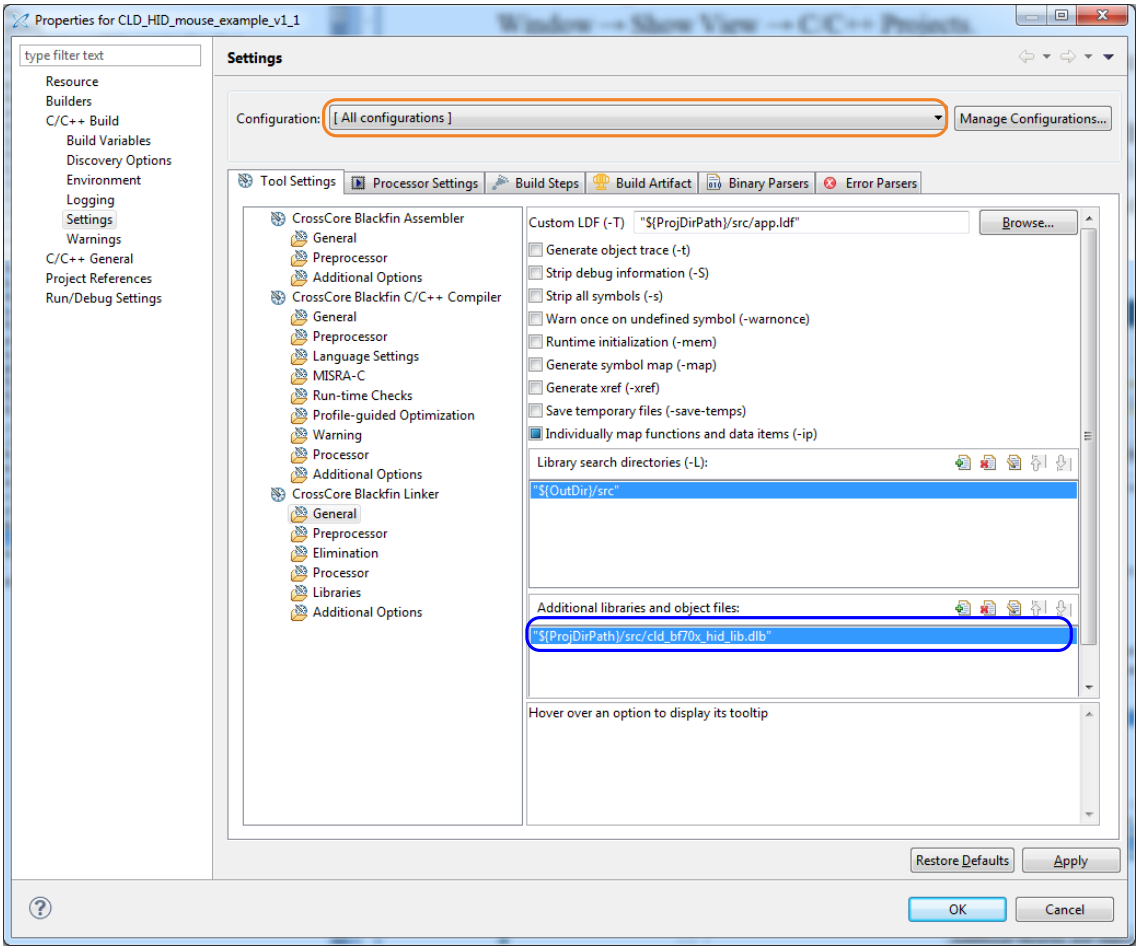
Adding the CLD BF70x HID Library to an Existing CrossCore Embedded Studio Project

In order to include the CLD BF70x HID Library in a CrossCore Embedded Studio (CCES) project you must configure the project linker settings so it can locate the library. The following steps outline how this is done.

1. Copy the cld_bf70x_hid_lib.h and cld_bf70x_hid_lib.dlb files to the project's src directory.
2. Open the project in CrossCore Embedded Studio.
3. Right click the project in the 'C/C++ Projects' window and select Properties.

   If you cannot find the 'C/C++ Projects" window make sure C/C++ Perspective is active. If the C/C++ Perspective is active and you still cannot locate the 'C/C++ Projects' window select Window → Show View → C/C++ Projects.
4. You should now see a project properties window similar to the one shown below.

   Navigate to the C/C++ Build → Settings page and select the CrossCore Blackfin Linker General page. The CLD BF70x HID Library needs to be included in the project's 'Additional libraries and object files' as shown in the diagram below (circled in blue). This lets the linker know where the cld_bf70x_hid_lib.dlb file is located.



5. The 'Additional libraries and object files' setting needs to be set for all configurations (Debug, Release, etc). This can be done individually for each configuration, or all at once by selecting the [All Configurations] option as shown in the previous figure (circled in orange).

User Firmware Code Snippets

The following code snippets are not complete, and are meant to be a starting point for the User firmware. For a functional User firmware example that uses the CLD BF70x HID Library please refer to the CLD HID Mouse Example v2.0 project included with the CLD BF70x HID Library. The CLD HID Mouse Example v2.0 project implements a basic USB Mouse using the Human Interface Device protocol.

main.c

```
void main(void)
{
    Main_States main_state = MAIN_STATE_SYSTEM_INIT;

    while (1)
    {
        switch (main_state)
        {
            case MAIN_STATE_SYSTEM_INIT:
                /* Enable and Configure the SEC. */

                /* sec_gctl - unlock the global lock  */
                pADI_SEC0->GCTL &= ~BITM_SEC_GCTL_LOCK;
                /* sec_gctl - enable the SEC in */
                pADI_SEC0->GCTL |= BITM_SEC_GCTL_EN;
                /* sec_cctl[n] - unlock */
                pADI_SEC0->CB.CCTL &= ~BITM_SEC_CCTL_LOCK;
                /* sec_cctl[n] - reset sci to default */
                pADI_SEC0->CB.CCTL |= BITM_SEC_CCTL_RESET;
                /* sec_cctl[n] - enable interrupt to be sent to core */
                pADI_SEC0->CB.CCTL = BITM_SEC_CCTL_EN;
                pADI_PORTA->DIR_SET = (3 << 0);
                pADI_PORTB->DIR_SET = (1 << 1);

                main_state = MAIN_STATE_USER_INIT;
            break;
            case MAIN_STATE_USER_INIT:
                rv = user_hid_init();
                if (rv == USER_HID_INIT_SUCCESS)
                {
                    main_state = MAIN_STATE_RUN;
                }
                else if (rv == USER_HID_INIT_FAILED)
                {
                    main_state = MAIN_STATE_ERROR;
                }
            break;

            case MAIN_STATE_RUN:
                 user_hid_main();
            break;

            case MAIN_STATE_ERROR:

            break;
        }
    }
}
```

user_hid.c

```
static const unsigned char user_hid_report_descriptor[] =
{
    /* Add custom HID Report Descriptor */
};

/* Interrupt IN endpoint parameters */
static CLD_HID_Endpoint_Params user_interrupt_in_endpoint_params =
{
     .endpoint_number                = 1,
     .max_packet_size_full_speed     = 64,
     .polling_interval_full_speed    = 1,
     .max_packet_size_high_speed     = 64,
     .polling_interval_high_speed    = 4,   /* 1ms */
};
/* Optional Interrupt OUT endpoint parameters */
static CLD_HID_Endpoint_Params user_interrupt_out_endpoint_params =
{
     .endpoint_number                = 1,
     .max_packet_size_full_speed     = 64,
     .polling_interval_full_speed    = 1,
     .max_packet_size_high_speed     = 64,
     .polling_interval_high_speed    = 4,   /* 1ms */
};


/* CLD BF50x HID library initialization data. */
static CLD_BF70x_HID_Lib_Init_Params user_hid_init_params =
{
    .uart_num        = CLD_UART_0,
    .uart_baud       = 115200,
    .sclk0           = 100000000u,
    .fp_console_rx_byte = user_hid_console_rx_byte,
    .vendor_id       = 0x064b,
    .product_id      = 0x0001,
    .report_descriptor_size = sizeof(user_hid_report_descriptor),
    .p_report_descriptor = (unsigned char *)user_hid_report_descriptor,

    .p_interrupt_in_endpoint_params = &user_interrupt_in_endpoint_params,

    /* Optional Interrupt OUT endpoint if not being used set endpoint params and data
       received callback set to CLD_NULL */
    .p_interrupt_out_endpoint_params = &user_interrupt_out_endpoint_params,
    .fp_interrupt_out_data_received = user_interrupt_out_data_received,

    .usb_bus_max_power = 0,

    .device_descriptor_bcdDevice = 0x0100,

    /* USB string descriptors - Set to CLD_NULL if not required */
    .p_usb_string_manufacturer  = "Analog Devices Inc",
    .p_usb_string_product       = "Example HID",
    .p_usb_string_serial_number = CLD_NULL,
    .p_usb_string_configuration = CLD_NULL,
    .p_usb_string_interface     = "BF707 HID Interface",

    .usb_string_language_id     = 0x0409,      /* English (US) language ID */

    .set_report_received = user_hid_set_report_received,
    .get_report_received = user_hid_get_report_received,
```

```
    .get_idle = user_hid_get_idle,
    .set_idle = user_hid_set_idle,

    .fp_cld_usb_event_callback = user_hid_usb_event_callback,
    .fp_cld_lib_status         = user_audio_status,
};

typedef enum
{
    USER_HID_INIT_SUCCESS = 0,
    USER_HID_INIT_ONGOING,
    USER_HID_INIT_FAILED,
} User_HID_Init_Return_Code;

User_HID_Init_Return_Code user_hid_init (void)
{
    static unsigned char user_init_state = 0;
    CLD_RV cld_rv = CLD_ONGOING;
    User_HID_Init_Return_Code init_return_code = USER_HID_INIT_ONGOING;

    switch (user_init_state)
    {
        case 0:
             /* TODO: Configure a timer to generate an interrupt every 125
                      microseconds, and call cld_time_125us_tick from interrupt. */
            /* TODO: Install USB and optionally the Console UART ISRs. */
            /* TODO: add any custom User firmware initialization */

            user_init_state++;
        break;
        case 1:
            /* Initialize the CLD BF50x HID Library */
            cld_rv = cld_bf70x_hid_lib_init(&user_hid_init_params);

            if (cld_rv == CLD_SUCCESS)
            {
                /* Connect to the USB Host */
                cld_lib_usb_connect();

                init_return_code = USER_HID_INIT_SUCCESS;
            }
            else if (cld_rv == CLD_FAIL)
            {
                init_return_code = USER_HID_INIT_FAILED;
            }
            else
            {
                init_return_code = USER_HID_INIT_ONGOING;
            }
    }
    return init_return_code;
}

void user_hid_main (void)
{
    cld_bf70x_hid_lib_main();

}
```

```
/* Function called when a Interrupt OUT packet is received */
static CLD_USB_Transfer_Request_Return_Type
      user_hid_interrupt_out_data_received(CLD_USB_Transfer_Params * p_transfer_data)
{
    p_transfer_data->num_bytes = /* TODO: Set number of Interrupt OUT bytes to
                                    transfer */
    p_transfer_data->p_data_buffer = /* TODO: address to store Interrupt OUT data */

    /* User Interrupt transfer complete callback function. */
    p_transfer_data->callback.usb_out_transfer_complete =
                                    user_hid_interrupt_out_transfer_done;
    p_transfer_params->transfer_aborted_callback = /* TODO: Set to User callback
                                                      function or NULL */
    p_transfer_params->transfer_timeout_ms = /* TODO: Set interrupt OUT transfer
                                                timeout */

    /* TODO: Return how the Interrupt OUT transfer should be handled (Accept, Pause,
             Discard, or Stall */
}

/* The function below is an example of the interrupt out transfer done callback
   specified in the CLD_USB_Transfer_Params structure. */
static CLD_USB_Data_Received_Return_Type user_hid_interrupt_out_transfer_done (void)
{
    /* TODO: Process the received Interrupt OUT transfer and return if the received
       data is good(CLD_USB_DATA_GOOD) or if there is an error
       (CLD_USB_DATA_BAD_STALL)*/
}

/* Function called when a Set Report request is received */
static CLD_USB_Transfer_Request_Return_Type user_hid_set_report_received
             (unsigned char report_id, CLD_USB_Transfer_Params * p_transfer_data)
{
    if (/* TODO: Check if report_id is valid */)
    {
        p_transfer_data->p_data_buffer = /* TODO: address to store Set Report data */
        p_transfer_data->callback.usb_out_transfer_complete =
                                    user_hid_set_report_transfer_complete;
        p_transfer_data->transfer_aborted_callback = /* TODO: Set to User callback
                                                        function or NULL */

        return CLD_USB_TRANSFER_ACCEPT;
    }
    else
    {
        return CLD_USB_TRANSFER_STALL;
    }
}

/* Function called when The Set Report data is received */
static CLD_USB_Data_Received_Return_Type user_hid_set_report_transfer_complete(void)
{
    if ( /* TODO: Check if Set Report data is valid */ )
    {
        return CLD_USB_DATA_GOOD;
    }
    else
    {
        return CLD_USB_DATA_BAD_STALL;
    }
}
```

```
/* Function called when a Get Report request is received */
static CLD_USB_Transfer_Request_Return_Type user_hid_get_report_received
             (unsigned char report_id, CLD_USB_Transfer_Params * p_transfer_data)
{
    if (/* TODO: Check if report_id is valid */)
    {
        p_transfer_data->num_bytes = /* TODO: Set to size of requested Report ID */
        p_transfer_data->p_data_buffer = /* TODO: address to store Get Report data */
        p_transfer_data->callback.usb_in_transfer_complete =
                                                user_hid_get_report_transfer_complete;
        p_transfer_data->transfer_aborted_callback = /* TODO: Set to User callback
                                                        function or NULL */

        return CLD_USB_TRANSFER_ACCEPT;
    }
    else
    {
        return CLD_USB_TRANSFER_STALL;
    }
}

/* Function called when a Get Report has been transmitted */
static void user_hid_get_report_transfer_complete (void)
{
    /* TODO: The Get Report data has been send to the Host, add any User
             functionality. */
}


CLD_USB_Data_Received_Return_Type user_hid_set_idle (unsigned char report_id,
                                                     unsigned char duration)
{
    if ( /* TODO: Check if report_id is valid */ )
    {
        /* TODO: Save the requested duration and process it accordingly */
        return CLD_USB_DATA_GOOD;
    }
    else
    {
        return CLD_USB_DATA_BAD_STALL;
    }
}

CLD_USB_Data_Received_Return_Type user_hid_get_idle (unsigned char report_id,
                                                     unsigned char * p_duration)
{
    if ( /* TODO: Check if report_id is valid */ )
    {
        *p_duration = /* TODO: Set to the current idle duration of the requested
                                Report ID. */
        return CLD_USB_DATA_GOOD;
    }
    else
    {
        return CLD_USB_DATA_BAD_STALL;
    }
}


static void user_hid_usb_event_callback (CLD_USB_Event event)
{
    switch (event)
    {
```

```
        case CLD_USB_CABLE_CONNECTED:
            /* TODO: Add any User firmware processed when a USB cable is connected. */
        break;
        case CLD_USB_CABLE_DISCONNECTED:
            /* TODO: Add any User firmware processed when a USB cable is
               disconnected.*/
        break;
        case CLD_USB_ENUMERATED_CONFIGURED:
            /* TODO: Add any User firmware processed when a Device has been
               enumerated.*/
        break;
        case CLD_USB_UN_CONFIGURED:
            /* TODO: Add any User firmware processed when a Device USB Configuration
               is set to 0.*/
        break;
        case CLD_USB_BUS_RESET:
            /* TODO: Add any User firmware processed when a USB Bus Reset occurs. */
        break;
    }
}


static void user_hid_console_rx_byte (unsigned char byte)
{
    /* TODO: Add any User firmware to process data received by the CLD Console UART.*/
}

/* The following function will transmit the specified memory using
   the Interrupt IN endpoint. */
static void user_hid_transmit_interrupt_in_data (void)
{
    static CLD_USB_Transfer_Params transfer_params;

    transfer_params.num_bytes = /* TODO: Set number of Interrupt IN bytes */
    transfer_params.p_data_buffer = /* TODO: address Interrupt IN data */
    transfer_params.callback.usb_in_transfer_complete = /* TODO: Set to User callback
                                                           function or NULL */;
    transfer_params.callback.transfer_aborted_callback = /* TODO: Set to User callback
                                                            function or NULL */;
    transfer_params.transfer_timeout_ms = /* TODO: Set interrupt OUT transfer
                                          timeout */

    if (cld_bf70x_hid_lib_transmit_interrupt_in_data(&transfer_params) ==
            CLD_USB_TRANSMIT_SUCCESSFUL)
    {
        /* Interrupt IN transfer initiated successfully */
    }
    else
    {
        /* Interrupt IN transfer was unsuccessful */
    }
}

static void user_cld_lib_status (unsigned short status_code, void * p_additional_data,
                                 unsigned short additional_data_size)
{
    /* TODO: Process the library status if needed.  The status can also be decoded to
             a USB readable string using cld_lib_status_decode as shown below: */

    char * p_str = cld_lib_status_decode(status_code, p_additional_data,
                                         additional_data_size);
}
```