USB Core for Altera DE-Series Boards

For Quartus II 13.0

1 Core Overview

The USB chip on Altera DE2/DE2-115 board is intended for connecting any USB device or for connecting the board to a computer. The USB Core provides a connection to the on-board USB controllers and presents an easy-to-use communication interface to the USB device.
The on-board USB chip contains the Host Controller (HC), the Device Controller (DC), and the On-The-Go (OTG) controller. The HC and DC share the same data bus, but use different I/O locations. There are two ports and a 4KB built-in buffer on the USB chip.

2 Functional Description

The USB Core's hardware provides a basic connection to the Host Controller and Device Controller ports of the on-board USB Controller. The USB Core's main function is to provide a set of useful software functions for communicating with the on-board USB Controller. These software functions can be easily included into a software program using the Altera Monitor Program or the Nios II IDE.

3 Instantiating the Core in Qsys

Designers can instantiate a USB Core by using the Qsys tool in Quartus II software. There is no need to configure the core.

4 Software Programming Model

4.1 Register Map

Device drivers control and communicate with the USB Core through four 32-bit registers. These registers are directly mapped to the four ports of the on-board USB Controller. Table 1 shows the details for the registers.

4.2 Software Functions

The USB Core is packaged with C-language functions accessible through the hardware abstraction layer (HAL) as listed below. These functions implement common operations that users need for the USB Core.

To use the functions, the C code must include the statements:

Table 1. USB Core register map			
Offset in bytes	**Register Name**	**R/W**	**Bit description**
			31...0
0	HcAddr	R	Host Controller Address
4	HcData	R/W	Host Controller Data
8	DcAddr	R	Device Controller Address
12	DcData	R/W	Device Controller Data

```
#include "altera_up_avalon_usb.h"
#include "altera_up_avalon_usb_ptd.h"
#include "altera_up_avalon_usb_regs.h"
#include "altera_up_avalon_usb_low_level_driver.h"
#include "altera_up_avalon_usb_high_level_driver.h"
```

In addition, some sample functions for specific communication with the mouse are also provided. They serve as a good starting point if the user wishes to develop more features with the USB. To use the mouse communication functions, the corresponding header file, `altera_up_avalon_usb_mouse_driver.h`, has to be included. These functions are described in Sections 4.7 - 4.11.

4.3 Writing programs using USB device driver functions

General steps for writing a program to communicate with the USB controller are:

1. Set up the corresponding controllers and buffers for USB operations.
2. Query the USB device and assign addresses to the connected devices.
3. Choose a configuration for the device.
4. Start transmitting data packets.

Step 1 configures the operational mode of the USB controller. Steps 2 and3 establish the connection between the HC and the device. Step 4 is the actual data transfer process. All the steps will be further clarified with the example functions shown in following sections.

4.4 Example Program: USB Mouse

A complete example of C code that uses the HAL device drivers is shown in Figure 1. The program displays the x and y locations of the mouse on the red LEDs and green LEDs respectively. It also shows the button status on the HEX displays. A '0' will appear on HEX0 if the right button is clicked, a '0' will appear on HEX1 if the center button is clicked, and a '0' will appear on HEX2 if the left button is clicked.
As shown in the Figure 1, the program first opens the USB device by calling the function `alt_up_usb_open_dev (...)` in line 4. The argument to be passed to the function should be the USB component name in the Qsys tool prefixed with the string */dev/*. In our example, the name of the USB component in the Qsys tool is *USB*, so the string

included in the function is */dev/USB*.
After the device pointer is returned, the program iteratively calls the function `alt_up_usb_setup(...)` in line 17 to set up a connection between the host and the device. Details about the setup function are given in the Section 4.5. The way we set up the USB controller here may not be optimal, it is a simple demonstration of the USB controller setup process.
Once the program confirms that the connected device is a mouse, it will choose a configuration on the device by calling the function `alt_up_usb_set_config(...)` in line 22.
Then it calls the function `alt_up_usb_play_mouse(...)` in line 25 to set up the mouse, and receives the mouse-movement packets. The function is explained in detail in Section 4.6.

```
1  int main(void){
2    // 1.Open the USB device
3      alt_up_usb_dev * usb_device;
4      usb_device = alt_up_usb_open_dev("/dev/USB");
5      if (usb_device != NULL) {
6          printf("usb_device->base %08x, please check if this matches the USB's base
               address in Qsys\n", usb_device->base);
7          unsigned int mycode;
8          int port = -1;
9          int addr = -1;
10         int config = -1;
11         int HID = -1; //Human Interface Device Descriptor number.
12         while (1) {
13             port = -1;
14             addr = -1;
15
16             // 2. Set up the USB and get the connected port number and its address
17             HID = alt_up_usb_setup(usb_device, &addr, &port);
18
19             if (port != -1 && HID == 0x0209) {
20                 // 3. After confirming that the device is connected is a mouse, the
                       host must choose a configuration on the device
21                 config = 1;
22                 mycode = alt_up_usb_set_config(usb_device, addr, port, config);
23                 if (mycode == 0) {
24                     // 4. Set up and play mouse
25                     alt_up_usb_play_mouse(usb_device, addr, port);
26                 }
27             }
28         }
29     }else {
30         printf("Error: could not open USB device\n");
31     }
32   return 0;
33 }
```

Figure 1. Example C code for the USB mouse demo.

4.5 Example Setup Function: alt_up_usb_setup

The function in Figure 2 shows a simple way to set up the USB controllers. It performs the following operations:

1. Reset and configure the controller by calling the functions in lines 16 and 17. The function `alt_up_usb_hc_initialize_defaults(...)` in line 17 configures the buffer sizes, disables all interrupts, and stops processing of all buffers.
2. Set the HC to the operational state by calling the function `alt_up_usb_hc_set_operational(...)` in line 20.
3. Enable the port if a connection has been detected using the function `alt_up_usb_enable_ports(...)` in line 21.
4. Assign new addresses to the ports by calling the function `alt_up_usb_assign_address(...)` in line 32; in our example, address 1 is assigned to port 1, address 2 is assigned to port 2.
5. Get the required descriptors by calling the function `alt_up_usb_get_control(...)` in lines 50 and 56.
6. Return the Human Interface Device (HID) number upon successful setup. The newly assigned port number and address will also be returned by pointers.

The function could be modified to fit specific requirements; more information can be found to ISP1362 USB datasheet online.

```
1  unsigned int alt_up_usb_setup(alt_up_usb_dev * usb_device, int * addr_ptr, int *
       port_ptr) {
2      unsigned int rbuf[128];
3      unsigned int mycode;
4      unsigned int iManufacturer, iProduct;
5      unsigned int status;
6      unsigned int new_port1_addr, new_port2_addr, print_port_info;
7      unsigned int extra, HID;
8
9      // 1 means port1 is connected; 2 means port2 is connected;
10     *port_ptr = -1;
11     HID = -1;
12
13     while (1) {
14
15         // Configure and Set up the controls of the ATL buffer
16         alt_up_usb_reset(usb_device);
17         alt_up_usb_hc_initialize_defaults(usb_device);
18
19         // Change the HC to operational state and Enable the port
20         alt_up_usb_hc_set_operational(usb_device);
21         alt_up_usb_enable_ports(usb_device);
22
23         // Suspend the host controller, if the system doesn't need it
24         alt_up_usb_hc_reg_write_16(usb_device, ALT_UP_USB_HcControl, 0x6c0);
```

```
25          alt_up_usb_hc_reg_write_16(usb_device, ALT_UP_USB_HcuPInterrupt, 0x1a9);
26
27          // Assign new addresses for port 1 and port 2, maximum addr number is 7
28          new_port1_addr = 1;
29          new_port2_addr = 2;
30          print_port_info = 0;
31
32          status = alt_up_usb_assign_address(usb_device, new_port1_addr,
                new_port2_addr, print_port_info);
33
34          // Enable ALT_IRQ and HC suspended
35          alt_up_usb_hc_reg_write_16(usb_device, ALT_UP_USB_HcuPInterruptEnable,
                0x120);
36
37          *port_ptr = -1;
38          extra = 0;
39
40          if ((status & 0x0001) != 0) {//port 2 active
41              *port_ptr = 2;
42              *addr_ptr = new_port2_addr;
43          } else if ((status & 0x0100) != 0) {//port 1 active
44              *port_ptr = 1;
45              *addr_ptr = new_port1_addr;
46          }
47
48          if (*port_ptr != -1) {
49              // Check port for device
50              mycode = alt_up_usb_get_control(usb_device, rbuf, *addr_ptr, 'D',
                    extra, *port_ptr);
51              if (mycode == 0x0300) {
52                  iManufacturer = rbuf[7]&0xFF;
53                  iProduct = (rbuf[7]&0xFF00) >> 8;
54                  alt_up_usb_addr_info(*addr_ptr, 'W', 'O', iManufacturer);
55                  alt_up_usb_addr_info(*addr_ptr, 'W', 'P', iProduct);
56                  mycode = alt_up_usb_get_control(usb_device, rbuf, *addr_ptr, 'H',
                        alt_up_usb_addr_info(*addr_ptr, 'R', 'P', 0), *port_ptr);
57
58                  HID = *(rbuf + 1);
59                  if (HID == 0x0209) { //it must be 0x0209, if connected device is a
                        mouse
60                      printf("\nMouse Detected...\n");
61                  } else if (HID == 0x0609) { //it must be 0x0609, if connected
                        device is a keyboard
62                      printf("\nKeyboard Detected...\n");
63                  } else {
64                      printf("\nUSB Device with HID 0x%04x Detected...\n", HID);
65                  }
66                  return HID;
67
68              }
69          }
```

```
70
71          {
72              volatile int usleep;
73              for (usleep = 0; usleep < 20000; usleep++);
74          }
75      }
76  }
```

Figure 2. Example HAL function for USB setup: alt_up_usb_setup.

4.6 Example Function for mouse communication: alt_up_usb_play_mouse

The function `alt_up_usb_play_mouse(...)` shown in Figure 3 demonstrates how to set up and retrieve mouse packets using polling. It keeps calling the function `alt_up_usb_retrieve_mouse_packet(...)` in line 57 to acquire the data from the mouse device.

```
1  unsigned int alt_up_usb_play_mouse(alt_up_usb_dev * usb_device, int addr, int port)
       {
2
3      printf("ISP1362 USB Mouse Demo.....\n");
4      alt_up_usb_mouse_setup(usb_device, addr, port);
5
6      alt_up_usb_mouse_packet usb_mouse_packet;
7
8      usb_mouse_packet.x_movement = 0;
9      usb_mouse_packet.y_movement = 0;
10     usb_mouse_packet.buttons = 0;
11
12     unsigned int pX = 320, pY = 240;
13
14     do {
15         pX = pX + usb_mouse_packet.x_movement;
16         pY = pY + usb_mouse_packet.y_movement;
17
18         if (pX > 639) {
19             pX = 639;
20         }
21         if (pX < 0) {
22             pX = 0;
23         }
24         if (pY > 479) {
25             pY = 479;
26         }
27         if (pY < 0) {
28             pY = 0;
29         }
30
31         alt_up_parallel_port_dev * Green_LEDs_dev;
32         alt_up_parallel_port_dev * Red_LEDs_dev;
33         alt_up_parallel_port_dev * HEX3_HEX0_dev;
34
```

```
35          Green_LEDs_dev = alt_up_parallel_port_open_dev("/dev/Green_LEDs");
36          Red_LEDs_dev = alt_up_parallel_port_open_dev("/dev/Red_LEDs");
37          HEX3_HEX0_dev = alt_up_parallel_port_open_dev("/dev/HEX3_HEX0");
38
39          alt_up_parallel_port_write_data(Red_LEDs_dev, pX);
40          alt_up_parallel_port_write_data(Green_LEDs_dev, pY);
41
42          if ((usb_mouse_packet.buttons & 0x1) == 1) { //left button
43              alt_up_parallel_port_write_data(HEX3_HEX0_dev, 0x3f0000);
44          }
45          if (((usb_mouse_packet.buttons & 0x2) >> 1) == 1) { //right button
46              alt_up_parallel_port_write_data(HEX3_HEX0_dev, 0x3f);
47          }
48          if (((usb_mouse_packet.buttons & 0x4) >> 2) == 1) { //center button
49              alt_up_parallel_port_write_data(HEX3_HEX0_dev, 0x3f00);
50          }
51
52          usb_mouse_packet.x_movement = 0;
53          usb_mouse_packet.y_movement = 0;
54          usb_mouse_packet.buttons = 0;
55
56      // Polling and get the data from the mouse
57      } while (alt_up_usb_retrieve_mouse_packet(usb_device, &usb_mouse_packet) !=
            ALT_UP_USB_MOUSE_NOT_CONNECTED);
58      printf("Mouse Not Detected\n");
59      return 0;
60  }
```

Figure 3. Example HAL function for USB data transfer: alt_up_usb_play_mouse

4.7 USB Device Driver Details

4.7.1 alt_up_usb_dev

Prototype:

```
typedef struct alt_up_usb_dev {
    alt_dev dev;
    unsigned int base;
    unsigned int irq_base;
    unsigned int irq_id;
} alt_up_usb_dev;
```

Include: `<altera_up_avalon_usb.h>`
Fields: `dev` – the device structure
`base` – the base address of the device's data port
`irq_base` – the base address of the device's interrupt port
`irq_id` – the interrupt ID of the device
Description: Define the device structure. Each instance of the driver uses one of these structures to hold its associated state.

4.7.2 usb_open_dev

Prototype: `alt_up_usb_dev* alt_up_usb_open_dev(const char* name)`
Include: `<altera_up_avalon_usb.h>`
Parameters: `name` – the specified name of the device in Qsys prefixed with the string `/dev/`.
Returns: the pointer to the USB device structure
Description: Open a USB device structure with `name` in Qsys.

4.7.3 struct ptd_struct

Prototype:

```
struct ptd_struct {
    alt_u16 actualBytes;
    alt_u8 completionCode;
    alt_u8 active;
    alt_u8 toggle;
    alt_u16 maxPacketSize;
    alt_u8 endpointNumber;
    alt_u8 last;      // For isochronous(ISO) transfer only
    alt_u8 speed;
    alt_u16 totalBytes;
    alt_u8 paired;
    alt_u8 pingPong;
    alt_u8 dirToken;
    alt_u8 functionAddress;
    alt_u8 pollingRate;   // For Interrupt transfer only
    alt_u8 startingFrame;
};
```

Include: `<altera_up_avalon_usb_ptd.h>`

Fields:
`actualBytes` – the actual amount of data transferred at the moment
`completionCode` – the completion code that reports success or errors in transaction. Details can be referred to Table3 in the appendix.
`active` – This is set to logic 1 by firmware to enable the execution of transactions by the HC. The Host Controller(HC) sets this bit to logic 0 when the transaction associated with this descriptor is completed to indicate that a transaction for this element should not be executed when it is next encountered in the schedule.
`toggle` – This bit is used to generate or compare the data Packet ID value (DATA0 or DATA1) for IN and OUT transactions. It is updated after each successful transmission or reception of a data packet.
`maxPacketSize` – the maximum amount of data per packet
`endpointNumber` – the USB address of the target endpoint within the function
`last` – This is 1 if this PTD is the last PTD. `last` is used only for ISO transfers. The last PTD is indicated by the HcINTLLastPTD and HcATLLastPTD registers.
`speed` – the speed of the port. 0 indicates low-speed, 1 indicates high-speed.
`totalBytes` – the total amount of data to be transferred
`paired` – This bit determines whether this PTD is a normal bulk PTD or a paired-PTD.
`pingPong` – the identification of the paired buffer. 0 indicates this is the ping buffer; 1 indicates this is the pong buffer. `pingPong` is used only when `paired` is set to 1.
`dirToken` – the token type to specify IN, OUT or setup token
`functionAddress` – the address of target device
`pollingRate` – the polling rate
`startingFrame` – the start frame number

Description: Define the Philips Transfer Descriptor(PTD). PTD is an 8-byte data structure to provide communication between the processor and USB Host Controller, it gives order to the Host Controller(HC) and reflects the status of USB transaction.

4.7.4 usb_init_ptd

Prototype: `void alt_up_usb_init_ptd(struct ptd_struct * ptd, unsigned int dirToken, alt_u8 endpointNumber, alt_u16 packetSize, alt_u8 toggle, alt_u8 functionAddress, alt_u8 portNumber)`
Include: `<altera_up_avalon_usb_ptd.h>`
Parameters: `portNumber` – the port number
`functionAddress` – the address assigned to the USB port
`toggle` – This bit is used to generate or compare the data Packet ID value (DATA0 or DATA1) for IN and OUT transactions. It could be 1 or 0.
`packetSize` – the size of the packet which specifies the maximum amount of data per packet
`endpointNumber` – the target endpoint number
`dirToken` – the direction token, which can be `ALT_UP_USB_DIR_TOKEN_SETUP`, `ALT_UP_USB_DIR_TOKEN_OUT` or `ALT_UP_USB_DIR_TOKEN_IN`.
`ptd` – the pointer to the Philips Transfer Descriptor(PTD) structure
Returns: nothing
Description: Initialize the the parameters in the PTD structure.

4.7.5 usb_convert_ptd_to_array

Prototype: `void alt_up_usb_convert_ptd_to_array(unsigned int * ptd_array, struct ptd_struct * ptd)`
Include: `<altera_up_avalon_usb_ptd.h>`
Parameters: `ptd` – the pointer to the Philips Transfer Descriptor(PTD) structure
`ptd_array` – the pointer to the array to store the PTD data
Returns: nothing
Description: Convert the data stored in the ptd_struct into an array.

4.8 Addresses of Registers in the USB Device Driver

Register Name	Address
Control and Status Registers	
ALT_UP_USB_HcRevision	0x00
ALT_UP_USB_HcControl	0x01
ALT_UP_USB_HcCommandStatus	0x02
ALT_UP_USB_HcInterruptStatus	0x03
ALT_UP_USB_HcInterruptEnable	0x04
ALT_UP_USB_HcInterruptDisable	0x05
Frame Counter Registers	
ALT_UP_USB_HcFmInterval	0x0D
ALT_UP_USB_HcFmRemaining	0x0E
ALT_UP_USB_HcFmNumber	0x0F
ALT_UP_USB_HcLSThreshold	0x11
Root Hub Registers	
ALT_UP_USB_HcRhDescriptorA	0x12
ALT_UP_USB_HcRhDescriptorB	0x13
ALT_UP_USB_HcRhStatus	0x14
ALT_UP_USB_HcRhPortStatus1	0x15
ALT_UP_USB_HcRhPortStatus2	0x16
DMA and Interrupt Control Registers	
ALT_UP_USB_HcHardwareConfiguration	0x20
ALT_UP_USB_HcDMAConfiguration	0x21
ALT_UP_USB_HcTransferCounter	0x22
ALT_UP_USB_HcuPInterrupt	0x24
ALT_UP_USB_HcuPInterruptEnable	0x25
Miscellaneous Registers	
ALT_UP_USB_HcChipID	0x27
ALT_UP_USB_HcScratch	0x28
ALT_UP_USB_HcSoftwareReset	0x29
Buffer RAM Control Registers	
ALT_UP_USB_HcBufferStatus	0x2C
ALT_UP_USB_HcDirectAddressLength	0x32
ALT_UP_USB_HcDirectAddressData	0x45
ISO Transfer Registers	
ALT_UP_USB_HcISTLBufferSize	0x30
ALT_UP_USB_HcISTL0BufferPort	0x40
ALT_UP_USB_HcISTL1BufferPort	0x42
ALT_UP_USB_HcISTLToggleRate	0x47
Interrupt Transfer Registers	
ALT_UP_USB_HcINTLBufferSize	0x33
ALT_UP_USB_HcINTLBufferPort	0x43

Continued on next page

Table 2 – continued from previous page

Register Name	Address
ALT_UP_USB_HcINTLBlkSize	0x53
ALT_UP_USB_HcINTLPTDDoneMap	0x17
ALT_UP_USB_HcINTLPTDSkipMap	0x18
ALT_UP_USB_HcINTLLastPTD	0x19
ALT_UP_USB_HcINTLCurrentActivePTD	0x1A
Aperiodic Transfer Registers	
ALT_UP_USB_HcATLBufferSize	0x34
ALT_UP_USB_HcATLBufferPort	0x44
ALT_UP_USB_HcATLBlkSize	0x54
ALT_UP_USB_HcATLPTDDoneMap	0x1B
ALT_UP_USB_HcATLPTDSkipMap	0x1C
ALT_UP_USB_HcATLLastPTD	0x1D
ALT_UP_USB_HcATLCurrentActivePTD	0x1E
ALT_UP_USB_HcATLPTDDoneThresholdCount	0x51
ALT_UP_USB_HcATLPTDDoneThresholdTimeOut	0x52
OTG Control Registers	
ALT_UP_USB_OTGControl	0x62
ALT_UP_USB_OTGStatus	0x67
ALT_UP_USB_OTGInterrupt	0x68
ALT_UP_USB_OTGInterruptEnable	0x69
ALT_UP_USB_OTGTimer	0x6A
ALT_UP_USB_OTGAltTimer	0x6C

Section 4.8 shows the names of the USB Host Control Data Registers and their corresponding addresses. Details about the functionality of each register can be found in the ISP1362 USB datasheet online.

4.9 USB_low_level_driver Details

4.9.1 usb_hc_reg_write_16

Prototype: `void alt_up_usb_hc_reg_write_16(alt_up_usb_dev * usb_device, unsigned char reg, unsigned int value)`
Include: `<altera_up_avalon_usb_low_level_driver.h>`
Parameters: `value` – the 16-bit value to be written to the Host Controller(HC) register
`reg` – the address of the HC register. The register addresses can be found in Section 4.8.
`usb_device` – the USB device structure
Returns: nothing
Description: Write a 16-bit value to the HC register.

4.9.2 usb_hc_reg_write_32

Prototype: `void alt_up_usb_hc_reg_write_32(alt_up_usb_dev * usb_device, unsigned char reg, unsigned long value)`
Include: `<altera_up_avalon_usb_low_level_driver.h>`
Parameters: `value` – the 32-bit value to be written to the Host Controller(HC) register.
`reg` – the address of the HC register. The register addresses can be found in Section 4.8.
`usb_device` – the USB device structure
Returns: nothing
Description: Write a 32-bit value to the HC register

4.9.3 usb_hc_reg_read_16

Prototype: `unsigned int alt_up_usb_hc_reg_read_16(alt_up_usb_dev * usb_device, unsigned char reg)`
Include: `<altera_up_avalon_usb_low_level_driver.h>`
Parameters: `reg` – the address of the Host Controller(HC) register. The register addresses can be found in Section 4.8.
`usb_device` – the USB device structure
Returns: the 16-bit value stored in the HC register
Description: Read a 16-bit value from the HC register

4.9.4 usb_hc_reg_read_32

Prototype: `unsigned long alt_up_usb_hc_reg_read_32(alt_up_usb_dev * usb_device, unsigned char reg)`
Include: `<altera_up_avalon_usb_low_level_driver.h>`
Parameters: `reg` – the address of the Host Controller(HC) register. The register addresses can be found in Section 4.8.
`usb_device` – the USB device structure
Returns: the 32-bit value stored in the HC register
Description: Read a 32-bit value from the HC register

4.9.5 usb_hc_write_atl

Prototype: `void alt_up_usb_hc_write_atl(alt_up_usb_dev * usb_device, unsigned int * a_ptr, unsigned int data_size)`
Include: `<altera_up_avalon_usb_low_level_driver.h>`
Parameters: `data_size` – the number of 16-bit words to be written
`a_ptr` – the pointer that points to the memory array that holds data to be written to the ATL buffer
`usb_device` – the USB device structure
Returns: nothing
Description: Write data to the ATL buffer.

4.9.6 usb_hc_read_atl

Prototype: `void alt_up_usb_hc_read_atl(alt_up_usb_dev * usb_device, unsigned int * a_ptr, unsigned int data_size)`
Include: `<altera_up_avalon_usb_low_level_driver.h>`
Parameters: `data_size` – the number of 16-bit words to be read
`a_ptr` – the pointer that points to the memory array to hold data from the ATL buffer
`usb_device` – the USB device structure
Returns: nothing
Description: Read data from the ATL buffer.

4.9.7 usb_hc_write_int

Prototype: `void alt_up_usb_hc_write_int(alt_up_usb_dev * usb_device, unsigned int * a_ptr, unsigned int data_size)`
Include: `<altera_up_avalon_usb_low_level_driver.h>`
Parameters: `data_size` – the number of 16-bit words to be written
`a_ptr` – the pointer that points to the memory array that holds data to be written to the INTL buffer
`usb_device` – the USB device structure
Returns: nothing
Description: Write data to the INTL buffer.

4.9.8 usb_hc_read_int

Prototype: `void alt_up_usb_hc_read_int(alt_up_usb_dev * usb_device, unsigned int * a_ptr, unsigned int data_size)`
Include: `<altera_up_avalon_usb_low_level_driver.h>`
Parameters: `data_size` – the number of 16-bit words to be read
`a_ptr` – the pointer that points to the memory array to hold data from the INTL buffer
`usb_device` – the USB device structure
Returns: nothing
Description: Read data from the INTL buffer.

4.9.9 usb_hc_direct_address_write

Prototype: `void alt_up_usb_hc_direct_address_write(alt_up_usb_dev * usb_device, unsigned int *a_ptr, unsigned int start_addr, unsigned int data_size)`
Include: `<altera_up_avalon_usb_low_level_driver.h>`
Parameters: `data_size` – the number of 16-bit words to be written
`start_addr` – the starting byte address of the data to be written in the Host Controller(HC) buffer
`a_ptr` – the pointer that points to the memory array that holds data to be written directly to the address specified
`usb_device` – the USB device structure
Returns: nothing
Description: Write data to the Host Controller(HC) buffer using direct addressing. This addressing method views the entire buffer as a single linear array of 4096 bytes instead of 4 sperate buffer areas(`ATL buffer`, `INTL buffer`, `ISTL0 buffer`, and `ISTL1 buffer`).

4.9.10 usb_hc_direct_address_read

Prototype: `void alt_up_usb_hc_direct_address_read(alt_up_usb_dev * usb_device, unsigned int *a_ptr, unsigned int start_addr, unsigned int data_size)`
Include: `<altera_up_avalon_usb_low_level_driver.h>`
Parameters: `data_size` – the number of 16-bit words to be read
`start_addr` – the starting byte address of the data to be read in the Host Controller(HC) buffer
`a_ptr` – the pointer that points to the memory array to hold the data read from the address specified
`usb_device` – the USB device structure
Returns: nothing
Description: Read data from the Host Controller(HC) buffer using direct addressing. This addressing method views the entire buffer as a single linear array of 4096 bytes instead of 4 sperate buffer areas(`ATL buffer`, `INTL buffer`, `ISTL0 buffer`, and `ISTL1 buffer`).

4.9.11 usb_dc_reg_write_16

Prototype: `void alt_up_usb_dc_reg_write_16(alt_up_usb_dev * usb_device, unsigned char reg, unsigned int value)`
Include: `<altera_up_avalon_usb_low_level_driver.h>`
Parameters: `value` – the 16-bit value to be written to the Device Controller(DC) register
`reg` – the address of the DC register.
`usb_device` – the USB device structure
Returns: nothing
Description: Write a 16-bit value to the DC register.

4.9.12 usb_dc_reg_write_32

Prototype: `void alt_up_usb_dc_reg_write_32(alt_up_usb_dev * usb_device, unsigned char reg, unsigned long value)`
Include: `<altera_up_avalon_usb_low_level_driver.h>`
Parameters: `value` – the 32-bit value to be written to the Device Controller(DC) register
`reg` – the address of the DC register.
`usb_device` – the USB device structure
Returns: nothing
Description: Write a 32-bit value to the DC register.

4.9.13 usb_dc_reg_read_16

Prototype: `unsigned int alt_up_usb_dc_reg_read_16(alt_up_usb_dev * usb_device, unsigned char reg)`
Include: `<altera_up_avalon_usb_low_level_driver.h>`
Parameters: `reg` – the address of the Device Controller(DC) register.
`usb_device` – the USB device structure
Returns: the 16-bit value stored in the DC register
Description: Read a 16-bit value from the DC register

4.9.14 usb_dc_reg_read_32

Prototype: `unsigned long alt_up_usb_dc_reg_read_32(alt_up_usb_dev * usb_device, unsigned char reg)`
Include: `<altera_up_avalon_usb_low_level_driver.h>`
Parameters: `reg` – the address of the Device Controller(DC) register.
`usb_device` – the USB device structure
Returns: the 32-bit value stored in the DC register
Description: Read a 32-bit value from the DC register

4.10 USB_high_level_driver Details

4.10.1 usb_setup

Prototype: `unsigned int alt_up_usb_setup(alt_up_usb_dev * usb_device, int * addr_ptr, int * port_ptr)`
Include: `<altera_up_avalon_usb_high_level_driver.h>`
Parameters: `port_ptr` – the pointer to the port number
`addr_ptr` – the pointer to the address
`usb_device` – the USB device structure
Returns: the Human Interface Device(HID) number. It must be 0x0209, if connected device is a mouse.
Description: Set up the USB, get the connected port number and its address. It is an simple example function of how to set up the USB. The USB can be set up in different modes to fit your own need.

4.10.2 usb_reset

Prototype: `int alt_up_usb_reset(alt_up_usb_dev * usb_device)`
Include: `<altera_up_avalon_usb_high_level_driver.h>`
Parameters: `usb_device` – the USB device structure
Returns: 0 on success
Description: Reset the USB operation.

4.10.3 usb_enable_ports

Prototype: `int alt_up_usb_enable_ports(alt_up_usb_dev * usb_device)`
Include: `<altera_up_avalon_usb_high_level_driver.h>`
Parameters: `usb_device` – the USB device structure
Returns: 0 on success
Description: Enable a port if a USB device is found to be connected to it and update the USB port status.
Notes: Port 2 is always used as a host port.

4.10.4 usb_hc_set_operational

Prototype: `int alt_up_usb_hc_set_operational(alt_up_usb_dev * usb_device)`
Include: `<altera_up_avalon_usb_high_level_driver.h>`
Parameters: `usb_device` – the USB device structure
Returns: 0 on success
Description: Set the Host Controller to the operational mode. In the operational mode, the bits in the HcBufferStatus register can request the Host Controller to start processing the data in buffers.

4.10.5 usb_hc_setup_atl_buffer

Prototype: `int alt_up_usb_hc_setup_atl_buffer(alt_up_usb_dev * usb_device)`
Include: `<altera_up_avalon_usb_high_level_driver.h>`
Parameters: `usb_device` – the USB device structure
Returns: 0 on success
Description: Set up the controls of the ATL buffer.

4.10.6 usb_hc_initialize_defaults

Prototype: `int alt_up_usb_hc_initialize_defaults(alt_up_usb_dev * usb_device)`
Include: `<altera_up_avalon_usb_high_level_driver.h>`
Parameters: `usb_device` – the USB device structure
Returns: 0 on success
Description: Initialize the host controller. It sets the ATL, INTL, ISTL buffer size to 1536, 1024 and 512 respectively, disables all the interrupts, sets up the controls of the ATL buffer and disables processing of all buffers.

4.10.7 usb_hc_disable_all_interrupts

Prototype: `void alt_up_usb_hc_disable_all_interrupts(alt_up_usb_dev * usb_device)`
Include: `<altera_up_avalon_usb_high_level_driver.h>`
Parameters: `usb_device` – the USB device structure
Returns: 0 on success
Description: Disable all interrupts.

4.10.8 usb_hc_set_istl_buffer_size

Prototype: `void alt_up_usb_hc_set_istl_buffer_size(alt_up_usb_dev * usb_device, unsigned int buffer_size)`
Include: `<altera_up_avalon_usb_high_level_driver.h>`
Parameters: `buffer_size` – the size of ISTL buffer in bytes
`usb_device` – the USB device structure
Returns: nothing
Description: Configure the ISTL buffer size.

4.10.9 usb_hc_set_intl_buffer_size

Prototype: `void alt_up_usb_hc_set_intl_buffer_size(alt_up_usb_dev * usb_device, unsigned int buffer_size)`
Include: `<altera_up_avalon_usb_high_level_driver.h>`
Parameters: `buffer_size` – the size of INTL buffer in bytes
`usb_device` – the USB device structure
Returns: nothing
Description: Configure the INTL buffer size.

4.10.10 usb_hc_set_atl_buffer_size

Prototype: `void alt_up_usb_hc_set_atl_buffer_size(alt_up_usb_dev * usb_device, unsigned int buffer_size)`
Include: `<altera_up_avalon_usb_high_level_driver.h>`
Parameters: `buffer_size` – the size of ATL buffer in bytes
`usb_device` – the USB device structure
Returns: nothing
Description: Configure the ATL buffer size.

4.10.11 usb_get_connection_status

Prototype: `int alt_up_usb_get_connection_status(alt_u8 portNumber)`
Include: `<altera_up_avalon_usb_high_level_driver.h>`
Parameters: `portNumber` – the port number
Returns: 1 if the port is connected, 0 otherwise.
Description: Get the connection status of the port specified.

4.10.12 usb_get_port_speed

Prototype: `int alt_up_usb_get_port_speed(alt_u8 portNumber)`
Include: `<altera_up_avalon_usb_high_level_driver.h>`
Parameters: `portNumber` – the port number
Returns: 1 for low-speed device, or 0 for high-speed device
Description: Get the speed of the port specified.

4.10.13 usb_send_control

Prototype: `unsigned int alt_up_usb_send_control(alt_up_usb_dev * usb_device, unsigned int * control_packet, unsigned int * return_packet)`
Include: `<altera_up_avalon_usb_high_level_driver.h>`
Parameters: `return_packet` – the pointer to the location that stores the data to be returned from the ATL buffer to the host
`control_packet` – the pointer to the location that stores the data to be sent from the host to the ATL buffer
`usb_device` – the USB device structure
Returns: a number. It should be greater than 0, otherwise it indicates a time out in the polling loop.
Description: Copy the PTD and payload (if any) into the ATL buffer and activate the buffer. Once the transaction is completed, the routine terminates and returns a number greater than zero together with the processed PTD in the return_packet to the calling function.

4.10.14 usb_set_address

Prototype: `unsigned int alt_up_usb_set_address(alt_up_usb_dev * usb_device, int port, int old_addr, int new_addr)`
Include: `<altera_up_avalon_usb_high_level_driver.h>`
Parameters: `new_addr` – the new address of the port
`old_addr` – the old address of the port. The initialized address for port is 0 by default.
`port` – the port number
`usb_device` – the USB device structure
Returns: 0 on success
Description: Allocate an available address to the newly connected USB device.
Notes: The routine has only two stages: Setup and Status. If an error is encountered in the Setup stage, it will not proceed to the Status stage to send out the data IN packet.

4.10.15 usb_set_config

Prototype: `unsigned int alt_up_usb_set_config(alt_up_usb_dev * usb_device, int addr, int port, int config)`
Include: `<altera_up_avalon_usb_high_level_driver.h>`
Parameters: `config` – the configuration to be set on the device
`port` – the port number
`addr` – the address assigned to the USB port
`usb_device` – the USB device structure
Returns: 0 on success
Description: Choose a configuration on the device after confirming the device identity.

4.10.16 usb_send_int_packet

Prototype: `unsigned int alt_up_usb_send_int_packet(alt_up_usb_dev * usb_device, unsigned int * control_packet, unsigned int * return_packet)`
Include: `<altera_up_avalon_usb_high_level_driver.h>`
Parameters: `return_packet` – the pointer to the location that stores the data to be returned from the INTL buffer to the host
`control_packet` – the pointer to the location that stores the data to be sent from the host to the INTL buffer
`usb_device` – the USB device structure
Returns: a number. It should be greater than 0, otherwise it indicates a time out.
Description: Send packet to the INTL buffer and activate the buffer. Once the transaction is completed, the routine terminates and returns a number greater than zero together with the the return_packet containing device information to the calling function.

4.10.17 usb_get_control

Prototype: `unsigned int alt_up_usb_get_control(alt_up_usb_dev * usb_device, unsigned int * return_packet, unsigned int addr, char control_type, unsigned int extra, int port)`
Include: `<altera_up_avalon_usb_high_level_driver.h>`
Parameters: `port` – the port number
`extra` – the extra information for string processing. It is ORed with the 5th element in alt_control_packet.
`control_type` – the type of descriptor you want to get from the USB device.
Input `'D'` for Device Descriptor, `'C'` for Configuration Descriptor, `'S'` for String Descriptor, `'E'` for Endpoint Descriptor, or `'H'` for Human Interface Device (HID) Descriptor.
`addr` – the address assigned to the USB port
`return_packet` – the pointer to the location that stores the data to be returned from the device to the host
`usb_device` – the USB device structure
Returns: the completion code. Bit 0-3 indicates the error code, bit 8-11 indicates at which stage the error was encountered, bit 12-15 is 0xF if time runs out, else 0.
Description: Get a descriptor from the device.

4.10.18 usb_addr_info

Prototype: `unsigned int alt_up_usb_addr_info(int addr, int mode, int dtype, int para)`
Include: `<altera_up_avalon_usb_high_level_driver.h>`
Parameters: `para` – the input parameter for the Write mode
`dtype` – the data type. Input `'S'` for speed, `'M'` for maximum packet size, `'O'` for manufacturer, or `'P'` for product.
`mode` – the mode of the operation. Input `'D'` for displaying data, `'W'` for writing data, or `'R'` for reading data.
`addr` – the address assigned to the USB port
Returns: the acquired data in read mode, 0 in other modes
Description: Perform display, write, or read operation on the USB data bank.

4.10.19 usb_assign_address

Prototype: `unsigned int alt_up_usb_assign_address(alt_up_usb_dev * usb_device, unsigned int new_port1_addr, unsigned int new_port2_addr, unsigned int print_port_info)`
Include: `<altera_up_avalon_usb_high_level_driver.h>`
Parameters: `print_port_info` – the parameter to enable printing the port information. Input 1 to enable the feature, 0 to disable it.
`new_port2_addr` – the new address for port 2
`new_port1_addr` – the new address for port 1
`usb_device` – the USB device structure
Returns: the status. The status &0x0100 becomes TRUE if port 1 has a device attached and an address has been successfully assigned to it. The status &0x0001 becomes TRUE if port 2 has a device attached and an address has been successfully assigned to it.
Description: Assign address to the connected port and return the status. It calls the `usb_set_address` function.

4.10.20 usb_hc_buffer_size_info

Prototype: `unsigned int alt_up_usb_hc_buffer_size_info(alt_up_usb_dev * usb_device)`
Include: `<altera_up_avalon_usb_high_level_driver.h>`
Parameters: `usb_device` – the USB device structure
Returns: 0 on success
Description: Print out the buffer size information.

4.11 USB_mouse_driver Details

4.11.1 alt_up_usb_mouse_packet

Prototype:

```
typedef struct alt_up_usb_mouse_packet {
    signed char x_movement;
    signed char y_movement;
    signed char buttons;
} alt_up_usb_mouse_packet;
```

Include: `<altera_up_avalon_usb_mouse_driver.h>`
Fields: `x_movement` – the mouse movement in x direction
`y_movement` – the mouse movement in y direction
`buttons` – the button status. Bit 0 indicates the left button is clicked, bit 1 indicates the right button is clicked, and bit 2 indicates the center button is clicked.
Description: Define the mouse packet structure.

4.11.2 usb_mouse_setup

Prototype: `int alt_up_usb_mouse_setup(alt_up_usb_dev * usb_device, int addr, int port)`
Include: `<altera_up_avalon_usb_mouse_driver.h>`
Parameters: `port` – the port number
`addr` – the address assigned to the USB port
`usb_device` – the USB device structure
Returns: 0 on success
Description: Set up the INTL buffer for the mouse packet transaction. This function should be called after confirming that the port is connected to a mouse device and an address has been assigned to the port.

4.11.3 usb_retrieve_mouse_packet

Prototype: `int alt_up_usb_retrieve_mouse_packet(alt_up_usb_dev * usb_device, struct alt_up_usb_mouse_packet * mouse_packet)`
Include: `<altera_up_avalon_usb_mouse_driver.h>`
Parameters: `mouse_packet` – the pointer to the mouse packet structure to be retrieved
`usb_device` – the USB device structure
Returns: 0 on success, 1 if no packet is retrieved, or 2 if the mouse is not connected
Description: Retrieve mouse packet.

4.11.4 usb_play_mouse

Prototype: `unsigned int alt_up_usb_play_mouse(alt_up_usb_dev * usb_device, int addr, int port)`
Include: `<altera_up_avalon_usb_mouse_driver.h>`
Parameters: `port` – the port number
`addr` – the address assigned to the USB port
`usb_device` – the USB device structure
Returns: 0 on success
Description: Retrieves mouse packet and displays the mouse information.

4.11.5 usb_mouse_example

Prototype: `void alt_up_usb_mouse_example(alt_up_usb_dev * usb_device)`
Include: `<altera_up_avalon_usb_mouse_driver.h>`
Parameters: `usb_device` – the USB device structure
Returns: nothing
Description: Demonstrate how the USB mouse works. By calling this function, it sets up the USB host controller, configures the mouse device, retrieves mouse packet and displays the mouse information on the FPGA boards (DE2 and DE2-115) by calling the `usb_setup`, `usb_set_config`, and `usb_play_mouse` functions. The LEDRs on board display the X location of the mouse, the LEDGs display the Y location of the mouse, and the HEX0-HEX2 display the status of the mouse buttons.

A Appendix

Table 3. Transaction Completion Codes

Completion codes	Value
ALT_UP_USB_PTD_CC_NoError	0x0
ALT_UP_USB_PTD_CC_CRC	0x1
ALT_UP_USB_PTD_CC_BitStuffing	0x2
ALT_UP_USB_PTD_CC_DataToggleMismatch	0x3
ALT_UP_USB_PTD_CC_Stall	0x4
ALT_UP_USB_PTD_CC_DeviceNotResponding	0x5
ALT_UP_USB_PTD_CC_PIDCheckFailure	0x6
ALT_UP_USB_PTD_CC_UnexpectedPID	0x7
ALT_UP_USB_PTD_CC_DataOverrun	0x8
ALT_UP_USB_PTD_CC_DataUnderrun	0x9
ALT_UP_USB_PTD_CC_BufferOverrun	0xC
ALT_UP_USB_PTD_CC_BufferUnderrun	0xD
ALT_UP_USB_PTD_CC_Initial	0xF

■