Altera DE2:
ISP1362 USB Controller
Application Notes

By: G16
Tarek Kaddoura
Jigar Nahar

Table of Contents

Introduction

The Altera DE2 Board contains a ISP1362 USB OTG Controller. The controller supports a wide range of functions as described in the reference manual. It can act as a USB host or act as a USB device. This document describes the basic procedure to set up the ISP1362 in Quartus, as well as the software drivers necessary to get it working.

Hardware Configuration

Terasic has provided the HDL and TCL files necessary to add the component to a system through SOPC builder. These files can be downloaded from:
https://www.ualberta.ca/~delliott/local/ece492/appnotes/2013w/USB ISP1362/ISP1362.tar

SOPC Builder

Note: The following steps only show what needs to be added to get the ISP1362 working. They assume that a NIOSII processor and other components (such as RAM) have already been added to the system.

1. In the downloaded file, there is a folder named ISP1362. Copy this folder to the root of your Quartus project.
2. In Quartus, go to Tools → SOPC Builder.
3. In the component list in SOPC Builder, there should be a new category: Terasic Technologies. Under this category is the ISP1362 component. Add this component to the system.
4. Generate the system.
5. Add the generated files to your project.

Top Level Modifications

Now that the component has been added, it needs to be connected through the top level HDL file.

Note: The following instructions assume that the ISP1362 component was named “isp1362” in SOPC Builder. Also, all necessary code is shown in VHDL.

1. Add the pins necessary in your top level entity.
2. Add the necessary signals into your system component.
3. Connect the pins as shown below in the VHDL code.

The ISP1362 component should now be connected. The final step is to program the board with the new configuration.

Software Configuration

After adding the component to the system, the software drivers are not added automatically in the NIOS II IDE. The following additions and initializations need to be done in software.

1. Add all the files under the folder ISP1362/software into your root project directory.
2. If you named your component something other than "ISP1362" in SOPC Builder, then open the ISP1362_HAL.H file and add the line:

#define ISP1362_BASE SOPCNAME_BASE

After the software modifications are complete, the ISP1362 should be working. The following section will test the chip.

Hardware Test

In order to test if the interface between the NIOS II and the ISP1362 is working, the chip ID of the USB Chip will simply be read in software. In the reference manual of the ISP1362, the chip ID register is supposed to return 3630 in hex. Hence, if this shows up then the interface is working.

1. Start by creating a simple Hello World project in the NIOS II IDE.
2. Perform the software modifications mentioned above.
3. Call the following function wherever possible to watch the output on stdout

Hal4D13_RegAccess();

4. Compile and run the program on the board

The function Hal4D13_RegAccess() performs several functions. It will reset the USB controller, print out the chip id, check the chip RAM, and print which configuration modes the chip is in.

Once the program runs, if the chip ID is readable and equal to 3630, then the interface to the ISP1362 is set up and working.

USB Device Hardware Test

This section will attempt to summarize how to initialize a connection with a USB device, and how to read the device's USB descriptor. It will also introduce how to talk to a USB device with interrupts. The instructions below assume that the software drivers have already been added as explained in the Software Configuration section.

Initializing a USB Device

In the ISP1362, there are three different types of transfers. Isochronous transfers, interrupt transfers, and control/bulk transfers. Consequently, each transfer has its own buffer.

Initializing the USB device involves sending control packets to the device, which means the ATL buffer will be used.

The basic procedure in initializing a USB device is:

1. Initialize USB Hardware
    a. Write into the HcReset register 0x00F6. This resets the USB hardware.
        i. *w16(HcReset, 0x00F6);*

b. Call the reset_usb() function.
    i. reset_usb();

2. Setup the ATL buffer parameters for control transfer
    w16(HcControl,0x6c0);
    w16(HcUpInt,0x1a9);
    w16(HcBufStatus,0x00);
3. Set the port the USB device is on as operational
    a. The drivers include a function set_operational() that queries the ports and sets any port as operational if it contains a device.
4. Enable the port
    a. The function enable_port() enables the ports that have an active device on them.
5. Assign an address to the USB device
    a. The assign_address() function will assign addresses to ports 1 and 2 respectively if a device is active on either port.
    b. assign_address(1, 2, 0);
        i. This will assign addresses 1 and 2 (from parameters) to ports 1 and 2 respectively.

Reading the USB Device Descriptor

Now that both the hardware and the device are initialized, the USB device descriptor of the device can be read. The USB device contains many descriptors. The device descriptor is only one of the descriptors that can be read.

In order to read the device descriptor, simply use the get_control() function in the drivers. This function has a prototype of

*unsigned int get_control(unsigned int *rptr,unsigned int addr,char control_type,unsigned int extra,int port);*

The parameters are:

- rptr: buffer to store the descriptor in
- addr: The address of the USB device
- control_type: option to choose which descriptor to read. Can be one of the following characters:
    - ‘D’: Device Descriptor
    - ‘C’: Configuration Descriptor
    - ‘S’: String Descriptor
    - ‘I’: Interface Descriptor
    - ‘E’: Endpoint Descriptor
    - ‘H’: HID Descriptor (for human interface devices)
- extra: Extra parameters to be passed. This depends on the descriptor used. For example, to get a string descriptor, the string index needs to be passed as the extra parameters. The string index can be read from the device descriptor. On the other hand, reading the device descriptor does not need any extra parameters.
- port: The port the device is on.

For example, we can call the function to read the device descriptor for a device with address of 2 and a port of 2:

- ret = get_control(rbuf, 2, 'D', 0, 2);
- The return value should be 0x300 only for a successful read. Now, the descriptor should be stored in rbuf.
- The result obtained is in the form of a whole device descriptor. To see the structure of each descriptor, read the following page: http://www.beyondlogic.org/usbnutshell/usb5.shtml

Writing to the USB Device

In order to talk to the USB device, the USB device configuration needs to be set first. The USB device can contain many configurations. These configurations can be extracted from the configuration descriptors of the USB device. Also, the device descriptor contains a bNumConfigurations field which is how many configurations there are on the device. Each configuration's configuration value can then be read from the configuration descriptor. In this section, it is assumed that the device is using configuration 0.

In order to set the configuration of the device, call the following function: *set_config(address, config);* This function sets the configuration for device on address "address".

- *set_config(2, 0);*
  - This sets the configuration as 0 for the device on address 2.

In addition to the configuration, the device's endpoints need to be obtained. These are addresses than define where the data goes on the device. To read the write or read endpoint addresses, the endpoint descriptor needs to be read. All control packets usually use the endpoint 0x0 (e.g. for initializing the device). In this example, it is assumed that the write endpoint for data transfer is at 0x1.

Interrupts will be used in order to write to the USB device. There are other methods not explained here, such as ATL bulk transfers and isochronous double-buffering transfers. Note that the name of Interrupt Write is misleading. The interrupt write is actually a blocking process.

The procedure for interrupt writes is as follows:

1. Setup the interrupt parameters. There are three parameters that need to be set up:
   a. HcIntSkip: Defines a map for the buffer to choose which PTDs to send and which to not send.
   b. HcIntLast: Defines the last PTD in the buffer
   c. HcIntBlkSize: Defines the size of the interrupt bulk
   d. For example, for sending only one PTD we can use the following parameters:
      i. unsigned long int_skip=0xFFFFFFFE;
      ii. unsigned long int_last=0x00000001;
      iii. unsigned int int_blk_size=64;
   e. Then, the interrupt parameters can be setup:
      i. w32(HcIntSkip,int_skip);
      ii. w32(HcIntLast,int_last);
      iii. w16(HcIntBlkSize,int_blk_size);
2. Create an interrupt PTD. The PTD contains the header of the USB packet.

a. make_int_ptd(cbuf, OUT, EP, pbytes, 0, address, port, freq);
    i. The function creates an OUT packet (for writing), with an endpoint of EP, a payload of size pbytes in bytes, and a frequency of freq. It then stores the result in cbuf. The frequency can be 0 for this example.

3. Add the payload to the PTD. Simply append the payload to the address "cbuf+4" (assuming cbuf is an integer array).
4. Send the interrupt using the send_int() function.
    a. send_int(cbuf, rbuf);
        i. This sends the PTD from cbuf that was just created. If there is any reply from the device, it will be stored in rbuf. But, there doesn't have to be a reply in order for the data to have been sent.

At this point, the data should be successfully sent.

In order to ease the configuration of a USB device, we have included a framework (Framework.c) in the drivers that include wrapper functions around common tasks, such as initializing the device, reading it's descriptors, setting its configuration, or writing to it.

Further Reading

Altera provides two different examples on USB integration. The projects are the DE2_NIOS_HOST_MOUSE_VGA and DE2_NIOS_DEVICE_LED.

The first project uses USB Host in order to interface with a mouse. It performs different actions on left or right mouse click. The second project uses USB in order to communicate with a computer.

Both projects provide a multitude of example code on how to communicate through the USB. They also contain implementations of Chapter 9 of the USB Device Specifications in Chap_9.c.

Note: The demonstration projects provided on the University labs are quite old, and will likely lead to hardware or other compilation errors. A newer version of the demonstrations updated to support Quatus II 10.0 are the DE2_70 demonstrations. These demonstrations are not for the Cylone II, but they contain excellent examples of updated code and drivers for the newer Quartus. These demonstrations can be downloaded from here: http://www.terasic.com/downloads/cd-rom/de2_70/DE2_70_demonstrations_V10.rar

References

Datasheet for the ISP1362: http: //www.cs.columbia.edu/~sedwards/classes/2013/4840/Philips-ISP1362-USB-controller.pdf

Initial drivers and HDL files:
http://www.terasic.com/downloads/cdrom/de2_70/DE2_70_demonstrations_V10.rar