Building an EZ-Host/OTG Project From Start to Finish

Introduction

The CY3663 EZ-Host and EZ-OTG Development Kit contains a full set of standard GNU-based tools. These tools have been ported to our CY16 processor-based EZ-Host and EZ-OTG products. This application note describes how to build a simple stand-alone project with these tools that will run in either debug mode or out of an EEPROM. This document is intended to help those less familiar with the GNU development environment. It is not intended to be a complete tools reference, but rather a quick example to get you familiar with the tools.

The example in this document is presented from start to finish and is all-inclusive. To make things easier to follow, this example does not have any dependencies on ROM BIOS code (except for initialization), Frameworks, Makefiles, or any other presupplied code that the design examples in our CY3663 development kit use. It is strongly recommended that production designs do not start from scratch as this example does. Instead, production designs should rely heavily on our BIOS and Frameworks code. For more information on developing around the BIOS and Frameworks, please see the book *USB Multi-Role Device Design by Example* included in the CY3663 development kit.

Required Tools Components

Below is a list of tools that are required to build and run the example in this application note (or any other project, for that matter). All of the required tools listed below, including many other tools, are installed automatically when the CY3663 CD is installed. In fact, these tools have many environmental dependencies in order to work properly that are also automatically handled by the install, so please properly install the CY3663 CD before attempting to build an EZ-Host or EZ-OTG project. For complete documentation on all utilities below, including standard RedHat GNUPro documentation, please refer to the CY3663 documentation folder.

- **cy16-elf-gcc:** Port of standard GNU GCC compiler
- **cy16-elf-as:** Port of standard GNU GAS assembler
- **cy16-elf-ld:** Port of standard GNU LD linker
- **cy16-elf-objdump:** Port of standard GNU objdump
- **cy16-elf-objcopy:** Port of standard GNU objcopy
- **cy16-elf-gdb:** Port of standard GNU GDB debugger
- **cy16-elf-libremote:** Port of standard GNU libremote
- **scanwrap:** Cypress-developed utility
- **qtui2c:** Cypress-developed utility
- **BASH_ENV.BAT:** Cypress-developed batch file

Buttons and Lights Example Description

To properly understand how to use these tools, it is best to actually build a simple project. This application note uses the ever-popular Buttons and Lights as that example. Buttons and Lights is a very basic application so as to allow focus to remain on the tools, but at the same time presents a visual indication that the code is running. It should be noted that this is not the same Buttons and Lights example presented in the *USB Multi-Role Device Design By Example* book.

The Buttons and Lights example presented here can run on either the EZ-Host or EZ-OTG stand-alone boards shipped with the CY3663 development kit. When this code is running, pressing the S5 button will turn ON LEDs D1, and D9:D7. When button S1 is pressed, these LEDs will turn off.

The EZ-Host and EZ-OTG stand-alone boards indirectly memory map these LEDs and buttons through the U8 CPLD. Therefore, this example bit bangs a standard memory interface to RD/WR the CPLD. A complete description of these boards, including the CPLD indirect memory map, is included in the *CY3663 Hardware Users Manual* included with the CY3663 kit.

Steps for building the Buttons and Lights example are covered in later sections of this application note.

Buttons and Lights Project Files

Below is a short list of the files that make up this Buttons and Lights example.

- BAL.c
- StartupNoBIOS.s or StartupWithBIOS.s
- cy7c67200_300.h
- BAL.ld

BAL.c File Description

This file contains a simple main routine along with functions to read and write the CPLD. The main routine is just a “while(1)” loop that looks for button presses and illuminates the LEDs as appropriate.

StartupNoBIOS.s/StartupWithBIOS.s File Description

The default GNU start-up code is generally contained within crt0.s. The crt0.s file contains a lot of code overhead in functions that are rarely used for embedded applications. Therefore, we will supply our own start-up code in a.s file and instruct the compiler to use it instead of the default crt0.s file. This will save quite a bit of code space. For more information on minimizing code space by not using crt0.s, please see the *OTG-Host Boot Code Design* white paper included in the CY3663 kit.

The most important portion of our start-up file is the _start routine. This routine is intended to be run before main and is loaded at the ORG location specified in the BAL.ld file. In its simplest form, the _start code is just a jump to main. Code at this location or the start of main will generally determine how the task or main function interacts with the BIOS.

BAL.ld File Description

BAL.ld is a modified version of the default linker script. The linker script contains commands for the linker to execute and describes how the sections in the input files should map into the output file. For example, the linker script tells where the base address for code should reside.

A default linker script can be automatically generated by typing "cy16-elf-ld --verbose > BAL.ld." This command will generate a default linker script file called BAL.ld that contains everything our project needs and more. There are a few simple changes that must be made to this default script for our project (and other EZ-Host/OTG projects). These small changes are listed below.

1. Change the base address for our code to start at. This can be done by changing the ". =" statement following SECTIONS to 0x1000. The 0x1000 location ensures that we allow enough room for the GDB stub to be loaded when in debug mode.
2. Anything that is not code in the file must be commented out, because otherwise a parse error will occur when linking. This includes the GNU version information in the top of the default file and the "=======" lines at the top and bottom of the file.

Buttons and Lights Without BIOS

The StartupNoBIOS.s file never returns control of the processor to the BIOS. The BIOS first executes some initialization routines such as setting up the stack and loading the program. Then program execution begins at "_start" as defined in the StartupNoBIOS.s file. StartupNoBIOS.s contains a jump to main at "_start" where main is just a "while(1)" loop that never returns.

Buttons and Lights With BIOS

Our Buttons and Lights example does not require the BIOS to be running in the background (except initialization). However, there are still some debug benefits to having the BIOS running. When the BIOS is running, we can use some simple Cypress-developed utility programs such as qtsdump/qtudump to dump memory and qtsarena/qtuarena to give memory usage information.

If BIOS is not running in the background, the BIOS IDLE chain will not to execute. This means BIOS will never look for any SCAN vectors on the UART or USB port. It is a requirement for the BIOS to execute and look for SCAN vectors over the UART or USB in order to use qtuload/qtsload or qtuarena/qtsarena (see Binary Utilities Reference for more information on these utilities).

There are many different ways to design our Buttons and Lights application to run concurrently with BIOS running in the background. Most of the suggested ways such as adding the Buttons and Lights task into the IDLE chain are described in *USB Multi-Role Device Design by Example*. We execute the BIOS IDLE Task in a periodic Timer0 Interrupt.

To allow Timer0 to properly interrupt at a given interval and vector to the proper ISR code, StartupWithBIOS.s makes the appropriate initialization in "_start" before returning to BIOS to complete initialization. In this example, we return to BIOS instead of jumping directly to main. This is required so that BIOS will finish initialization of the UART and USB. Instead of jumping to main, our start-up code replaces the IDLER ISR vector address with main. The IDLER ISR (now main) is automatically called by BIOS upon finishing initialization.

Building the Example for Running with the Debugger (GDB)

If a program is built to run with the debugger, then the build process is different from code being built to run out of an EEPROM.

Running code with GDB will allow for breakpoints, single stepping, and other standard debug activities, in addition to running Insight, the graphical user interface for GDB. To run code with GDB, an Executable Linker Format (ELF) must be generated. This is the default format of the GCC compiler and is file-extension-independent. The only requirement is that debug symbols need to be turned on. The following command line instruction will generate a proper ELF file for use with GDB. It is assumed that the CY3663 CD has been installed and that BASH_ENV.BAT has been invoked to create a bash shell environment. In addition, change the working directory to that containing the Buttons and Lights files. Full details of how to actually run the debugger are listed in both the CY16.PDF and the *USB Multi-Role Device Design by Example* book.

It should be noted that even though StartupNoBIOS.s is used, the GDB stub that is loaded for debug will still execute the BIOS IDLE chain. This is required to allow debug communication over USB or the UART.

```
[cy]$ cy16-elf-gcc -nostartfiles -g -TBAL.ld
StartupNoBIOS.s BAL.c -o BAL
```

- The above command will generate a file called "BAL" in the ELF format. This file can then be used as an input when invoking GDB.
- The "-nostartfiles" parameter tells the compiler/linker not to use the default start-up code. Instead, we will supply StartupWithBIOS.s as the start-up code to be used.
- The "-g" parameter instructs the compiler to turn on debug symbols.
- The "-T" parameter tells the linker to use our supplied linker script instead of the default.
- The "-o" parameter is followed by the output file name.

Building the Example for Running out of the EEPROM

The build process is slightly more involved to create an image that is ready to run out of an EEPROM. All of the steps are listed below and can be entered at the command line in a bash shell, or included in a script or Makefile. Since this project is so small, a Makefile is not needed. For a Makefile

example, please see the CY3663 design examples that use Makefiles that contain similar calls as those listed below.

Below is a list of the steps required to build an EERPOM-ready image.

It should be noted that StartupNoBIOS is used as the default startup file in this example but StartupWithBIOS may be used without any additional changes.

1. Open a bash shell.

   Run BASH_ENV.bat by clicking on the shortcut created under Programs/Cypress/OTG-Host in the start menu after the CY3663 CD has been installed. Once a Bash shell is opened, a "[cy]$" prompt will be displayed.

   In addition, change the working directory to that which contains the Buttons and Lights files.

2. Assemble the start-up file into a linkable object file.

   [cy]$ cy16-elf-as StartupNoBIOS.s -o StartupNoBIOS.o.

3. Compile the .C file into a linkable object file (ELF file).

   [cy]$ cy16-elf-gcc -c -nostartfiles BAL.c -o BAL.o.

4. Link both of the object files.

   [cy]$ cy16-elf-ld -TBAL.ld StartupNoBIOS.o BAL.o -o BAL.

5. Obtain a Listing file (optional).

   [cy]$ cy16-elf-objdump -D -z -t --source BAL > BAL.lst.

6. Convert the ELF file format to a Binary file format.

   [cy]$ cy16-elf-objcopy -O binary BAL BAL.bin.

7. Add appropriate SCAN signatures to the Binary file.

   [cy]$ scanwrap BAL.bin BAL_Scan.bin 0x00001000.

8. Download the Binary file to the EEROM.

   [cy]$ qtui2c BAL_Scan.bin -f.

Below is a detailed description of each of the above command line entries.

1. BASH_ENV.bat.

- BASH_ENV.bat will set up all of the required environment variables and then invoke a bash shell.

2. cy16-elf-as -StartupNoBIOS.s -o StartupNoBIOS.o

- The "-o" parameter specifies the desired output file name.

3. cy16-elf-gcc -c -nostartfiles BAL.c -o BAL.o

- The "-c" parameter causes a compile/assemble without running the linker.
- The "-nostartfiles" parameter tells the compiler/linker not to use the default start-up code.
- The "-o" parameter specifies the desired output file name.

4. cy16-elf-ld -TBAL.ld StartupNoBIOS.o BAL.o -o BAL

- The "-T" parameter tells the linker to use our supplied linker script instead of the default.
- The "-o" parameter specifies the desired output file name.

5. cy16-elf-objdump -D -z -t --source BAL > BAL.lst.

- This step is not required, but is nice to have.
- The "-D" parameter causes the contents of all sections to be disassembled.
- The "-z" parameter causes blocks of zeros to be disassembled.
- The "-t" parameter prints the symbols table entries of the file.
- The "--source" parameter displays source code intermixed with disassembly, if possible.
- The ">" parameter directs the output to a file with name provided after the ">" parameter.

6. cy16-elf-objcopy -O binary BAL BAL.bin.

- The "-O" parameter writes an output file in the format specified after the "-O" parameter. In this case, we specify the "binary" format.

7. scanwrap BAL.bin BAL_Scan.bin 0x00001000.

- The address parameter is the base address where the program is to be loaded. This is the same address that was added into to *.ld file and needs to be the address where the _start routine in the start-up file resides.

8. qtui2c BAL_Scan.bin -f.

- Please see the *Binary Utilities Reference* document for detailed instructions on how to run this utility. Basically, the EZ-Host or EZ-OTG mezzanine card should have all DIP switches OFF and then the board should be powered with the USB cable plugged into peripheral port 2A. Next, the DIP switches should be set to select the appropriate EEPROM to download code to. An example is to set switches [6:3] ON and [2:1] OFF (0bXX111100) which selects EEPROM#4. Finally, qtui2c should be run followed by a board reset. At this point the Buttons and Lights code should be running.
- The "-f" parameter specifies that the EEPROM is 4K-64K. If the EERPOM is 256-2K, no "-f" parameter is required.

References

All documents listed below that were referenced throughout this application note can be found in the CY3663 Development Kit CD image.

1. *USB Multi-Role Device Design By Example*
2. *Binary Utilities Reference*.pdf
3. *CY3663* Hardware Users Manual.pdf
4. *Frameworks Reference Manual*.pdf
5. CY16.pdf
6. *GNUPro Toolkit Documentation* (several documents)

Conclusion

Once you have built and run the Buttons and Lights example presented in this application note, you should have a basic understanding of how to get up and running with the CY16 toolset. The next step is to start learning about the EZ-Host and EZ-OTG BIOS and Frameworks. These two items will provide a very comprehensive foundation on which to build your application code. In addition, Frameworks relies heavily on Makefiles that in turn use some of the basic commands discussed throughout this document.

Source Code

BAL.c

```
#include "cy7c67200_300.h"

typedef unsigned short uint16;

#define WRITE_REGISTER(address, value)  (*(volatile uint16 *) (address)) = ( (uint16) (value) )
#define READ_REGISTER(address)          (*(volatile uint16 *) (address))
#define INPLACE_OR(address, value)  __asm( "or [%0], %1" : : "p" ((address)), "g" ((value)) )
#define INPLACE_AND(address, value)  __asm( "and [%0], %1" : : "p" ((address)), "g" ((value)) )

void writeCPLDADX(uint16 address)
{
    INPLACE_AND(GPIO1_OUT_DATA_REG, ~0x0008);   /* A0 = 0 */
    INPLACE_AND(GPIO1_OUT_DATA_REG, ~0x0020);   /* nCS = 0 */
    INPLACE_AND(GPIO1_OUT_DATA_REG, ~0x0040);   /* nWR = 0 */

    INPLACE_OR(GPIO0_DIR_REG, 0x00FF);          /*Write Address*/
    WRITE_REGISTER(GPIO0_OUT_DATA_REG, address);

    INPLACE_OR(GPIO1_OUT_DATA_REG, 0x0040);     /* nWR = 1 */
    INPLACE_OR(GPIO1_OUT_DATA_REG, 0x0020);     /* nCS = 1 */
    INPLACE_OR(GPIO1_OUT_DATA_REG, 0x0008);     /* A0 = 1 */
}

void writeCPLDDATA(uint16 data)
{
    INPLACE_OR(GPIO1_OUT_DATA_REG, 0x0008);     /* A0 = 1 */
    INPLACE_AND(GPIO1_OUT_DATA_REG,~0x0020);    /* nCS = 0 */
    INPLACE_AND(GPIO1_OUT_DATA_REG, ~0x0040);   /* nWR = 0 */

    INPLACE_OR(GPIO0_DIR_REG, 0x00FF);          /*Write Address*/
    WRITE_REGISTER(GPIO0_OUT_DATA_REG, data);

    INPLACE_OR(GPIO1_OUT_DATA_REG, 0x0040);     /* nWR = 1 */
    INPLACE_OR(GPIO1_OUT_DATA_REG, 0x0020);     /* nCS = 1 */
    INPLACE_OR(GPIO1_OUT_DATA_REG, 0x0008);     /* A0 = 1 */
}

int readCPLDDATA()
{
    uint16 data;

    INPLACE_OR(GPIO1_OUT_DATA_REG, 0x0008);    /* A0 = 1 */
    INPLACE_AND(GPIO1_OUT_DATA_REG, ~0x0020);   /* nCS = 0 */
    INPLACE_AND(GPIO1_OUT_DATA_REG, ~0x0080);   /* nRD = 0 */

    INPLACE_AND(GPIO0_DIR_REG, 0xFF00);         /*Write Address*/
    data = (READ_REGISTER(GPIO0_IN_DATA_REG) & 0x00FF);

    INPLACE_OR(GPIO1_OUT_DATA_REG, 0x0080);     /* nWR = 1 */
    INPLACE_OR(GPIO1_OUT_DATA_REG, 0x0020);     /* nCS = 1 */

    return data;
}

void writeCPLD(int address, int data)
{
    writeCPLDADX(address);
    writeCPLDDATA(data);
}

int readCPLD(int address)
{
    writeCPLDADX(address);
    return readCPLDDATA();
}

int main()
{
    uint16 button_read = 0;

    INPLACE_AND(UART_CTL_REG, 0xFE);            /* Disable UART to free up GPIO 6 & 7 on EZ-OTG for CPLD*/

    INPLACE_OR(GPIO0_OUT_DATA_REG, 0x00FF);     /* Data Port */
    INPLACE_OR(GPIO1_OUT_DATA_REG, 0x00E8);     /* A0 = nCS = nRD = nWR = 1 */
    INPLACE_OR(GPIO0_DIR_REG, 0x00FF);
    INPLACE_OR(GPIO1_DIR_REG, 0x00FF);
```

```
 while(1)   {

    button_read = readCPLD(0x0000);             /* Read Buttons */

    if((button_read & 0x01) == 0)   {            /* If Button S1 (PB_UP) is pressed, turn LEDs OFF */
      writeCPLD(0x7, 0xFF);
      writeCPLD(0x1, 0xFF);                     /* Clear Button S1 Status */
    }

    if((button_read & 0x08) == 0)   {            /* If Button S5 (PB_DN) is pressed, turn LEDs ON */
      writeCPLD(0x7, 0x00);
      writeCPLD(0x4, 0xFF);                     /* Clear Button S5 Status */
    }

    }

}}
```

StartupNoBIOS.c

```
; File: startup.s

; Declare the main function.
.global main

.section .text

.global _start
_start:
    ; Jump to the main function.
    jmp    main

; End of file: startup.s
```

StartupWithBIOS.c

```
; File: startup.s

; Declare the main function.
.global main

.section .text

.global _start
_start:
    ; Jump the main function.
    ; jmp    main

;   Replace TMR0 ISR with tmr0_isr
    mov     [0], tmr0_isr

;   Enable TMR0 INT
    or      [0xC00e], 0x0001

;  Replace IDLER ISR with main.Now BIOS IDLE chain is only called in TMR0 ISR.
    mov     [(71*2)],main
    ret

;   int 70 enables the IDLE_INT.  This allows bios to execute periodically
;   since main never returns.  Now we can use debug USB port and UART.

tmr0_isr:               ; TMR0 ISR, now calls BIOS IDLE
    push    [0xc000]    ; Push Flags Register
    int     73          ; PUSHALL_INT
    int     70          ; IDLE_INT (will check UART and USB for Scan signitures)
    int     74          ; POPALL_INT

    mov     [0xc010], 10000     ; Load TMR0 so that it will expire every 10ms
    pop     [0xc000]            ; Pop Flags Register
    sti
    ret

; End of file: startup.s
```


Approved AN048 8/20/03 kkv