DEVELOPMENT OF UCLINUX SPI interface
On SPARTAN 3e Development board
For
Cal Poly Super Project

By

Alex Diaz

Senior Project

ELECTRICAL ENGINEERING DEPARTMENT
California Polytechnic State University
San Luis Obispo
2008
Contents
Error! Bookmark not defined.
List of Figures

Figure 1: CentOS Development Packages .......................................................... 12
Figure 2: EDK Reference Hardware Project provided by Petalinux ............................ 15
Figure 3: SPI Memory map for EDK Hardware Profile ........................................... 16
Figure 4: External Signals and interrupt signal line ................................................ 17
Figure 5: SPI External Ports .................................................................................... 18
Figure 6: Interrupt priority setup via interrupt controller ........................................ 19
Figure 7: Constraints file ......................................................................................... 21
Figure 8: Sample program using GPIO char driver .................................................. 24
Figure 9: Auto-config.h define for SPI baseaddress ................................................. 25
Figure 10: xparameters.h define for SPI baseaddress ............................................. 26
Figure 11: Boot Log errors 5 lines from the bottom ................................................. 27
Figure 12: system.mss spi - driver version ............................................................. 28
Figure 13: xspi_l.h example ................................................................................. 29
Figure 14: Xparameters.h example ....................................................................... 31
Figure 15: SPI Directly Memory access driver ....................................................... 33
Figure 16: Direct Memory Driver ........................................................................... 43
Acknowledgments

I'd like to thank Brian Estrada and Tony Wonsyld for helping me through many late nights. This project would not exist if it weren't for Dr. Jim Harris. Thank you for letting me participate in a project whose concept may be simple but whose implications are grand.
Introduction

The Super Project’s goal is to develop a low cost source of energy for underdeveloped countries with little to no power infrastructure that are well insolated. The concept is simple: In countries where waiting for big investments to build the infrastructures necessary for energy like power plants, power lines, sub stations, service centers, and safety regulations among other things, why not develop cheap, independent sources of energy for single family homes that can be delivered directly. Instead of waiting for a top down approach, why not empower the people. Our immediate goal is to provide enough low cost power to make an impact on daily life. By providing for basic needs like lighting, refrigeration, and the ability to pump water from a well using a motor, the quality of life of many people can be significantly improved.
Background

Our prototype centers on a photovoltaic panel. It is our primary means of power generation and is rated at 150W @ 30V. A DC-DC buck converter allows us to provide 12V which charges a battery and powers all the loads including DC motor, led lights, and the control system.

The control system is a very important aspect of this project. It regulates how and when the battery charges, monitors voltages, currents, and temperatures, and has margins of safety built in which will shut down the system in case of emergency. It runs on a conventional laptop computer in a Linux environment. The drawback of this control system is that the laptop can use up to 60W of power (instantaneous). Given that our solar panel can provide 150W (peak instantaneous) of power, this means that the laptop can potentially use up about 40% of our power generated.

Brian Estrada and Patrick Mariano, in their senior project, installed a distribution of uclinux called petalinux on a Spartan 3e-500 FPGA board with the idea of eventually having this small footprint, low power (3W) board provide all the controls necessary to operate our power system. Petalinux is in turn a version of Linux that is specifically developed for the use in embedded applications. It runs on a flat file system and is very small allowing it to fit in the restricted memory spaces of most embedded systems. Patrick and Brian went as far as hard coding a PWM signal output on one of the I/O pins of the FPGA board. They also were able get the dip switches and led’s working via a GPIO driver built into petalinux.
Requirements

This senior project continues the development of petalinux on the Spartan 3E development board. I will build a hardware platform in EDK using a template provided in petalinux as a base build and also add an SPI softcore (EDK IP CORE). I will then interface with the SPI hardware. There are various methods to do this, three of which I’ll address:

1) Use the included “char” driver included in petalinux which can be installed via the kernel build menu in “make menuconfig”

2) Write a driver using low level subroutines included in the SPI hardware header files

3) Write a driver to interface with the SPI core.

For this project I have attempted all three methods.

The first option allows for less code development because a driver is already written. It would involve instantiating the driver, which would be found in /dev, and using it to write and read. The second method involves taking advantage of the various routines that are defined in header file xspi_l.h which can be found in the microblaze_0/libsrc/ directory inside the EDK hardware project. In theory, simply calling the functions in an application should allow for the transfer of data through the SPI hardware.

The second method involves using subroutines provided by EDK. These subroutines are in header files that are created when “Generate Libraries and BSP’s” is run from inside EDK provided an SPI IP core has been added to the hardware profile. The header file of main concern is xspi_l.h which can be found in /$petalinux/hardware/user-hardware/Xilinx-Spartan3E500-RevD-edk91/microblaze0/include.
The third method involves writing a driver from scratch. In the CPE 329 course an SPI driver was written from scratch using Xio32 subroutines. This same method can be applied in this case. The code is the same and for the most part is modular so long as the base addresses for the SPI hardware are updated.

Once the driver is working properly, future generations of students working on the Super Project will be able to interface with SPI peripherals such as digital to analog converters as well as analog to digital converters. This will allow the ability to read data from all the system sensors which have analog output voltages through analog to digital converters that use an SPI interface. It will also allow the Buck Converter to be driven with an SPI interfaced digital to analog converter instead of a pwm signal. The SPI interface will allow the control system on the Fpga to replace the laptop and provide a significant power savings.
Environment Setup

Outline
First I’d like to specifically state what applications we will be using and their purpose.

1. Petalinux-distribution v0.30-rc1
   
   This will be installed in a Linux environment (PC) and provides compiling tools, EDK hardware designs, and the boot loaders. It will also be used to write and compile user applications and the kernel. Please note that petalinux is a distribution of uclinux and in this paper I will refer to them interchangeably[4].

2. Export
   
   This application will take the hardware design (download.bit file) created with EDK and put it in on the FPGA board.

3. EDK 9.1i
   
   This was provided by Dr. Harris.

4. ISE 9.1i
   
   This was also provided by Dr. Harris  NOTE:  Download update 9.3i from the update menu within either EDK or ISE.

5. Kermit 8.0.211
   
   Download this free program from http://www.columbia.edu/kermit/current.html. It is used to start a terminal connection to send u-boot, the kernel files, and interface with the Spartran-3e once its boots up.

6. CentOS v5
A free distribution of Linux which will run all of our petalinux software as well as EDK and ISE.

**Design Cycle**

The typical design cycle is as follows:

1. Design the hardware in EDK (reference design found in /$Petalinux/hardware/reference-designs/Xilinx-Spartan3E500-RevD-edk91 is recommended as a template)
2. Build initial Kernel
3. Write/compile/load user applications into kernel. This is done on the PC.
4. Make everything
5. Open a connection with the FPGA board using Kermit
6. Send download.bit file to FPGA via Export which loads the hardware profile as well as installs the initial boot loader, FS-BOOT, into Bram. If this is successful you will see text in your terminal window.
7. Send U-boot, the secondary boot loader, which will eventually pull the OS out of Flash Memory and boot-up the system.
8. Load kernel into flash of FPGA via Kermit transfer (takes a total of about 10 minutes).
9. Cycle power
10. Test applications.

Once the hardware is all done steps 1 and 2 and be skipped.

**CentOS**

CentOS is a free Linux distribution meant to run on a PC. Since petalinux only runs in a Linux
environment, CentOS being freely distributed, is a great candidate for our purposes. It can be obtained at 

http://www.centos.org/

After installing versions 3 and 4 I had little success with getting all of the hardware on my laptop working. Both version 3 and 4 didn’t find my wireless card. Version 3 didn’t recognize the Keyspan serial to USB adapter which is used to run a terminal connection from a USB port on the PC running CentOS to the Spartan3E serial port. Without it, unless you are using a laptop that has a serial port, you cannot proceed. You need to be able to talk to the FPGA. Version 4 and 5 actually recognized the adapter but the default drivers did not work. I couldn’t find a Linux driver from the manufacturer’s website or through a Google search so instead I used a spare USB to Serial adapter based on the Prolific chipset that I had laying around. Again, without serial communication the project cannot proceed.

I stuck with version 5 of CentOS because it worked well with my wireless Ethernet card. All installation processes are very intuitive and easy to follow. Of great importance is that when the setup process asks what kind of packages to install you click development and at a minimum install “development libraries” and “development tools”. I didn’t do this at first and had errors when trying to compile Kermit. And did I mention, it’s CentOS is free.
Petalinux
The petalinux distribution can be obtained at:

http://developer.petalogix.com/

under downloads. As of this writing the current version is 0.30-rc1 (356Mb). Once downloaded, you simply unzip it and you are ready to go. The Petalinux development environment requires Linux in order to compile user applications, the kernel, and the boot loader u-boot. This gives two options: The installation of something like VMware, a commercial software package that lets you install “virtual”
operating systems” like Linux that run I a window inside any Windows Operating system, or installing Linux directly on the hard drive via a complete reformat or a dual boot.

I tried VMware and do not recommend it. I created a “virtual machine” with CentOS installed on it and preceded with the project. Sending data to the FPGA is primarily accomplished via UART using a terminal program like Kermit. Since any changes to user applications require the recompiling of the kernel this means that the kernel will be sent to the FPGA board frequently. The kernel size is about 3MB’s. When transferring through a VMware virtualization of CENTOS it took about 30 minutes. This made the design cycle horrible! What I did instead is format my personal laptop and install CentOS directly on the hard drive. This cut down the transfer time to about 7 minutes. While not the greatest it is more bearable than 30 minute.

The reason everything is transferred via UART is because the EDK version I was using did not have a valid license for an Ethernet IP core. Transferring of the kernel files would probably significantly improve but, again, Ethernet is not required.

For further detail in the setup of the petalinux environment please review Patrick and Brian’s Senior Project[2]. A very useful guide is provided at the development site for petalinux. It is specifically tailored to the Spartan-3e-500 and can be found at the following site:


It goes over setting up the petalinux environment on a PC.
System Architecture and Design

Base Hardware

For this project I will be using the reference hardware design provided with petalinux. It is specifically built for the Spartan-3e-500 and will be the fastest way of getting thing started. It’s found in /$Petalinux/hardware/reference-designs/Xilinx-Spartan3E500-RevD-edk91. Simply copy the entire folder to $/petalinux/hardware/user-platforms. Most of the documentation for booting up petalinux use this reference hardware design.
Next simply add the SPI interface by going to IP catalog and selecting “Communication Low-Speed” and OPB SPI Interface. Under bus interface make sure you connect the SOPB to the mb_opb bus. Next in Addresses, I set it up to be 16k and start at memory location 0x41600000.
Once that’s done, click on Ports. The SPI requires, at a minimum, an SCK, MOSI, SS, and an interrupt line if you plan on using interrupts. Therefore we make the mentioned signals external since they will be going to peripherals which are external to the FPGA silicon chip.
It turns out that EDK is very picky with the signal names. I was required to name them `fpga_0_spi_(NAME OF SIGNAL)_pin`. The net names, which are available to rename when you expand “external ports” were the same as the signal names except for the ending “_pin”, which was removed. Anything other than that specific syntax threw an error during the compile so it is important that you follow this naming convention.

**Figure 4: External Signals and interrupt signal line**
If you look closely at the reference design hardware in figure 1, you will notice that it includes an interrupt controller labeled opb_intc_0. Expand opb_spi_0 and you will see a signal called IP2INTC_Irpt. Pull down the drop box and select opb_spi_0_IP2INTC_Irpt. This virtually connects the SPI interrupt line to the Interrupt controller. Now expand opb_intc_0 and click on the gray box labeled “L to H”.

**Figure 5: SPI External Ports**
Once the interrupt connection dialog box is open you will see two selection areas. On the left you have “potential interrupt connections” which show all available interrupt lines. On the right you’ll see “Connected Interrupts”. Selecting any “potential interrupt” and moving it to “connected interrupts” will allow the interrupt controller to handle that interrupt. In our case, we’d like to move opb_spi_0_IP2INTC_Irpt over to the “connected interrupts” side.

**Removing Ethernet IP Core (If license is expired)**

Before we move on, unless you have a license for the Ethernet IP you must remove it from the
hardware profile. The petalinux reference hardware project for the Spartan-3e-Revision D board includes the ETHERNET Ip core. While the use of the Ethernet Ip Core along with tftpboot would speed up development by making the transfer of all the u-boot files faster than the serial connection the license unfortunately ran out while I was testing. This meant that EDK no longer allows for the building of hardware projects. It gives an error when the license for the Ethernet IP core is not found. To solve this I had to remove the Ethernet IP core. The removal process is straight forward. Inside EDK, click on the System Assembly View; click the bus interface filter and then scroll down until you see opb Ethernet Mac. Right click it and click remove ip core and external ports.

Next we need to setup the constraints file. Click on the project tab and expand project files. Next double click on the systems.ucf file to open it up. Here we will remove all the constraints for the Ethernet driver. Simply scroll down until you come to the heading labeled Ethernet and delete all the entries. Next add the SPI signal pins to the ucf file.
We are almost done. The only thing remaining is adding the booloader FS-BOOT into BRAM. A great tutorial is provided here:

http://developer.petalogix.com/wiki/BoardGuides/Xilinx/S3E500/Tutorial

Once this is done generate libraries and BSP’s, Generate Net list, and Generate Bitstream.

Go into your hardware folder and look in the implementation folder. If the build was successful there will be a download.bit file waiting. This file is what we’ll send to the FPGA via Digilent’s Export utility. Once the hardware is built we can pretty much close EDK.

Next build the kernel. If you need a refreshes on how to accomplish this Brian and Patrick’s paper provide a very good example[2].
**Petalinux/U-boot Modifications**

**Compiling kernel without Ethernet support**

While this is all that is required for EDK, U-BOOT still has some dependencies on the Ethernet support which will give errors and stop the compiling process when the Ethernet hardware is not found. To fix this we must go to $Petalinux/software/petalinux-dist/ and type in make menuconfig.

Select Customize Kernel Settings

Exit and Save Kernel Configuration

Now select Networking

Deselect Networking support

Exit

Select Device Drivers

Select Character Devices

If we are attempting to use the built in driver then also select Xilinx OPB SPI support

Exit and save all settings

At this point if you do a “make all” you will notice that everything complies fine until you get to the U-boot portion. U-boot has Ethernet adapter dependencies which halt the compiling process since no Ethernet IP core is present. This means we need to comment out some of the code in a few source files.

Do the following:

edit /net/net.c line #93 to !CFG_CMD_NET

edit /common/cmd_net.c #31, same thing as above
edit /common/cmd_nvedit.c #49, #365, same thing as above.

This should get everything working. Basically, I negate whatever CFG_CMD_NET was defined as. It seems that if I were able to find where CFG_CMD_NET is defined, I could simply toggle its present state in one location instead of negating it at 3 different locations. Unfortunately I wasn’t able to find where this master definition occurred and instead used the slightly longer route.

Now, if using the 2.6 core type in “make all” and everything should compile. If successful you will get a few files in the /tftpboot folder as explained in the tutorial. These files will have to be transmitted to the Spartan-3e board using Kermit. The process is also covered in Brian and Patrick’s paper as well as at:

http://developer.petalogix.com/wiki/BoardGuides/Xilinx/S3E500/Tutorial

The tutorial there is specifically tailored to the Spartan3E500 board[6].

Implementation of Driver

The built in SPI “char” driver

Char drivers are easy to work with and as can be seen from the following test program I wrote for use with the GPIO char driver. This sample program takes in the DIP Switches and outputs the result to the LEDs.
Figure 8: Sample program using GPIO char driver

Something similar to this should be easy to implement for the SPI. Unfortunately, during the uclinux boot up process the OS probes for the SPI hardware and outputs “error -14”, as in figure 8, when it doesn’t find it. There is not very much documentation regarding this specific spi driver so it is difficult to pinpoint exactly what “error -14” is. One guess is that the error is output by the kernel itself. If so then error -14 for the kernel is defined as “Bad Address”. I verified the baseaddress in both auto-config.h and xparameters.h just to be sure. Both files can be found in the /$petalinux/hardware/user-
platform/Xilinx-Spartan3E500/microblaze0/include folder. The results can be seen in figure 9 and 10, and are as expected with a baseaddress of 0x416000000.

```c
#define CONFIG_XILINX_INTC_0_IRQ_ACTIVE (1)
#define CONFIG_XILINX_INTC_0_INSTANCE "opb_intc_0"
#define CONFIG_XILINX_INTC_0_HW_VER "1.00.c"

/* Definitions for SPI_0 */
#define CONFIG_XILINX_SPI_0_INSTANCE "opb_spi_0"
#define CONFIG_XILINX_SPI_0_FIFO_EXIST (1)
#define CONFIG_XILINX_SPI_0_OBP_SCK_RATIO (32)
#define CONFIG_XILINX_SPI_0_SPI_SLAVE_ONLY (0)
#define CONFIG_XILINX_SPI_0_NUM_OFFCHIP_SS_BITS (6)
#define CONFIG_XILINX_SPI_0_NUM_SS_BITS (1)
#define CONFIG_XILINX_SPI_0_NUM_BITS_SR (8)
#define CONFIG_XILINX_SPI_0_DEV_BLK_ID (4)
#define CONFIG_XILINX_SPI_0_DEV_MIR_ENABLE (0)
#define CONFIG_XILINX_SPI_0_FAMILY "spartan3e"
#define CONFIG_XILINX_SPI_0_BASEADDR (0x41600000)
#define CONFIG_XILINX_SPI_0_HIGADDR (0x41603FFF)
#define CONFIG_XILINX_SPI_0_INTERRUPT_PRESENT (1)
#define CONFIG_XILINX_SPI_0_OBP_AWIDTH (32)
#define CONFIG_XILINX_SPI_0_OBP_DWIDTH (32)
#define CONFIG_XILINX_SPI_0_IP_REG_BAR_OFFSET (0x60606060)
#define CONFIG_XILINX_SPI_0_DEPTH (16)
#define CONFIG_XILINX_SPI_0_NUM_BITS_REG (8)
```

Figure 9: Auto-config.h define for SPI baseaddress
I tried changing base addresses in the EDK memory map as well as adding and removing the Interrupt lines for the SPI and nothing seems to work. I even removed the GPIO for the dip switches and put the SPI core in its place at memory location 0x4002000 with an allocated block of memory 64k wide instead of the 16k that I’ve been using. Still, the same errors appear.
One possibility for the boot up sequence not detecting the SPI hardware could be that the driver itself was written for a very specific SPI IP Core version which happens to not be version 1.01.a.
I next attempted to use low level subroutines provided by EDK for the IP core. I imported the header file `xspi_l.h` from the `/petalinux/hardware/user-platforms/Xilinx-Spartan3E500-RevD-edk91/microblze_0/libsrc/spi_v1_01_a/src/` folder. It includes a couple #defines of specific SPI Registers.

**Low level subroutines (xspi_lh)**
Figure 13: xspi_l.h example

You must also include Linux/autoconf.h. Autoconf.h has a #define for the base address of the SPI core.

```c
#include <stdio.h>
#include <Linux/autoconf.h>
#include "xspi_l.h"

int main(char value)
{
    char val2send;
    val2send=0;
    val2send=value;
    //argv[1]=val2send;
    XSpi_mSetControlReg(CONFIG_XILINX_SPI_0_BASEADDR, XSP_CR_ENABLE_MASK | XSP_CR_MASTER_MODE_MASK | XSP_CR_CLK_PHASE_MASK);
    XSpi_mSetSlaveSelectReg(CONFIG_XILINX_SPI_0_BASEADDR, 0xFFFFFFFE);
    XSpi_mSendByte(CONFIG_XILINX_SPI_0_BASEADDR, val2send);
    XSpi_mSendByte(CONFIG_XILINX_SPI_0_BASEADDR, 0xFF);
    while((XSpi_mGetStatusReg(CONFIG_XILINX_SPI_0_BASEADDR) & XSP_SR_TX_EMPTY_MASK) == 0)
```
```c
XSpi_mSetSlaveSelectReg(CONFIG_XILINX_SPI_0_BASEADDR, 0xFFFFFFFF);
return 0;
```

Basically, the code toggles the SS line, loads 8 bits into the transmit registers, and polls the transmit empty register. This isn’t making use of the interrupts and should be easier to troubleshoot. Once the data is sent it, toggles the SS line. Unfortunately this compiles, but doesn’t work. Nothing seems to be getting clocked out.

### Direct memory access (Xio32/xio.h)

My final attempt at getting a driver to work involves manually accessing memory via the Xio_in32/out32 subroutines defined in xio.h and the xparameters.h file which can be imported from the microblaze_0/libsrc/ header file. Xio.h has a list of subroutines that can be used to access memory registers directly. Primarily used are the Xio32(register, value) and the Xio32_read(register) subroutines. The xparameters.h file contains “# defines” of all the hardware memory locations that were defined in the EDK hardware profile. This is equivalent to the approach taken in CPE 329 and is a little more detailed than the defines included in the autoconf.h file. From the data sheet of the SPI core provided by Xilinx it is evident there are various registers that need to be accessed in order to setup and write data to the SPI.

Instead of reading each register by defining its address directly I use the fact that the BaseAddress is already defined in the xparameters.h file. Then it follows that to access the appropriate registers I can simply reference the base address and add on offset, again this is the same approach used in CPE 329.
Once again, the following code compiled without errors. It sets the configuration bits for the SPI, loads 8 bits into the transmit register, enables the transmit by clearing the master inhibit bit, polls to make sure the transfer is done, and repeats the process for another 8 bits. Notice that the two bytes that are attempted to be sent are hard coded. This means that as soon as I run the application data should be output on the pins described in the ucf file. When I ran the application it executes properly but no data is seen on the oscilloscope.

Figure 14: Xparameters.h example
#include <stdio.h>
#include "/opt/pkg/petalinux-v0.30-rc1/hardware/user-platforms/Xilinx-Spartan3E500-RevD-edk91/microblaze_0/include/xio.h"
#include "/opt/pkg/petalinux-v0.30-rc1/hardware/user-platforms/Xilinx-Spartan3E500-RevD-edk91/microblaze_0/include/xparameters.h"

int vdd1=0x0F;
int vdd2=0xFF;
int busyFlag;

int main()
{
    // write initial data to master SPIDTR register (i.e. 2000)
    // but only first 8 bits (i.e. 07 for MSB) not 7D0
    XIo_Out32(XPAR_SPI_BASEADDR + 0x068, vdd1);
    // insure SPISSR register has all ones
    XIo_Out32(XPAR_SPI_BASEADDR + 0x070, 0x1);
    // write data to SPICR
    // including setting manual slave select assertion enable (bit 24),
    // enable bit (bit 30), master transfer inhibit bit (bit23)
    XIo_Out32(XPAR_SPI_BASEADDR + 0x060, 0x196);
    // write to SPISSR to manually assert SS vector
    XIo_Out32(XPAR_SPI_BASEADDR + 0x070, 0x0);
    // write data to SPICR, but clear inhibit bit to start transfer
    XIo_Out32(XPAR_SPI_BASEADDR + 0x060, 0x96);
    // poll status for completion
    busyFlag = XIo_In32(XPAR_SPI_BASEADDR + 0x64);
    busyFlag = busyFlag & 0x4;
    while(busyFlag != 0x4)
    {
        busyFlag = XIo_In32(XPAR_SPI_BASEADDR + 0x64);
        busyFlag = busyFlag & 0x4;
    }
    // set master transaction inhibit bit
    XIo_Out32(XPAR_SPI_BASEADDR + 0x060, 0x196);
    // write rest of data to master SPIDTR register (i.e. 2000)
    XIo_Out32(XPAR_SPI_BASEADDR + 0x068, vdd2);
    // write data to SPICR, but clear inhibit bit
    XIo_Out32(XPAR_SPI_BASEADDR + 0x060, 0x96);
    // poll status for completion
    busyFlag = XIo_In32(XPAR_SPI_BASEADDR + 0x64);
    busyFlag = busyFlag & 0x4;
    while(busyFlag != 0x4)
    {
        busyFlag = XIo_In32(XPAR_SPI_BASEADDR + 0x64);
        busyFlag = busyFlag & 0x4;
    }
}
// write all ones to SPIISR to deassert SS vector
XIo_Out32(XPAR_SPI_BASEADDR + 0x070, 0x1);
// disable devices
return 0;
}

Figure 15: SPI Directly Memory access driver

This is the most direct approach I’ve taken so far. In fact, it was suggested by Dr. Harris in an attempt to remove any unknowns that may have been inherent in the previous two attempts. Unfortunately, again, there is no output at any of the pins described in the constraints file.
Testing Hardware

Testing the hardware is straight forward. Basically once the user application is compiled into the uclinux image and transferred to flash via Kermit, it appears in the /bin directory upon booting up, where it can be executed. Based on the constraints various scope probes are attached to the SS line, SCK line, and MOSI Line. Usually, for SPI hardware, as soon as data is sent to the transmit register, the SCK activates, the SS line goes low, and the data gets pushed on the MOSI line. Every time I run the application I should see various data on the pins. I see nothing. Petalinux/uclinux is well documented as far as booting up is concerned but once it’s installed with its default parameters the resources online are very limited. SPI interfaces are common yet it is very difficult to find support. I feel that I am missing something simple.

Updating kernel on Spartan-3e quickly

At this point I have been testing different applications which require them to be recompiled into the kernel and resent to the Spartan 3e board. Assuming there is an already existing kernel on the board there are two choices for updating:

a) Erase all of the flash memory and reinstall everything

or

b) Erase the current kernel from flash memory and install the new kernel to flash

I will address option b, as Patrick and Brian have already addressed option a. Option b allows us to
simply update the kernel with the new updated applications. The entire environment doesn’t have to be deleted because no new hardware is being added. We are simply updating already existing applications or adding new ones that will interface with hardware that is already preset. First, using export, send the newly created download.bit file which can be found in $petalinux/hardware/user-platforms/Xilinx-Spartan3e-500-revD/implementation.

An easy way to do this is to share a folder on a windows machine. Then create a folder called “temp” in /mnt. Now simply run the following command:

```
mount -t cifs //192.168.0.115/temp /mnt/temp -o username=barryGoldwater
```

Simply replace 192.168.0.115 with your windows ip, and username with your windows username. You will get prompted for a password. Once this is done you can easily send download.bit, or any other file for that matter, to your windows machine by simply running

```
cp download.bit /mnt/temp.
```

Next run export and push download.bit onto the Spartan Board. Let the process run like normal but when it asks to hit any key to stop auto boot, hit any key. This will take you to a u-boot prompt.

Run the following commands

```
U-boot>loadb 0x24000000
type “cntrl-\-c “
```
U-boot> send /bin /tftpboot/image.ub

after about 10 minutes the file should be transferred.

Next do the following

U-boot> connect

U-Boot> protect off $(kernstart) +$(kernsize)

U-Boot> erase $(kernstart) +$(kernsize)

U-Boot> cp.b $(clobstart) $(kernstart) $(filesize)

Once the kernel is copied to flash, power cycle the board and boot into uclinux like normal. When you check your /bin file you’ll see any new applications you’ve written as well as current updated applications.
Final Assessment

After attempting to implement an SPI driver using three different approaches I have come to the conclusion that there is something very simple hindering my success. I have written some SPI drivers before, including for the Spartan 1 board, pic processors, and a rabbit processor. SPI is very well documented and the process is very simple. Unfortunately, under ucLinux I have had no success.

I believe that either:

a) The operating system itself is preventing me from accessing memory directly

OR

b) The hardware itself is not being detected.

I’ll address a first. It seems that it is quite possible that ucLinux has a unique way of accessing memory that requires some extra step to read and write to hardware. I’ve searched online and have come up with nothing.

As for b, well this could be the cause for a. If uclinux sees no hardware present then it may be impossible to access it through the OS. Using the low level sub routines I was able to store data and read it back but being that I am using said subroutines, I am not positive that the values I’m reading/writing are actually going to the spi hardware and not a buffer somewhere. Moreover, when I tried doing a similar read/write test using the direct memory approach (via xparameters.h) I only received 0’s. This could mean that indeed the hardware is not present, which doesn’t make too much sense since I built it in EDK, more than likely, I am not writing to memory correctly, or at least not in a
way uClinux likes.

**Recommendations and Improvements**

One of the drawbacks of user applications in petalinux is that each time the application is modified and compiled it must be added to the kernel and recompiled. This means that developing the driver was fairly time consuming. Even at a 7 minute developing cycle, things get a little annoying sometimes. I would recommend the purchase of an Ethernet IP Core License to speed up the transfer process. Uclinux fully support tftp transfers which, over Ethernet, are much quicker than serial communications.

Another recommendation would be to have someone work on this project that has embedded Linux design experience. Although uClinux appears to be widely used, the documentation for this open source project is very limited. The vast majority of documentation emphasizes getting the system up and running. Once the basics are achieved, the documentation dwindles quickly, and even the mailing list is not helpful.
Conclusion

This project involved attempting to write an SPI driver for the embedded operating system, uClinux, which will run on the Spartan-3e500 FPGA board. A wide variety of peripherals use SPI to communicate including Digital to Analog converters as well as Analog to Digital converters. Since all sensors used in the SuPER project provide analog output, the ability to interface with such peripherals will directly influence the replacement of the SuPER control system, which runs on a laptop, with an embedded control system that runs on the FPGA. This change in platform will give a big cost savings as well as a substantial energy saving benefit.

After trying three different approaches to implement the driver I believe accessing the xparameter registers directly is the way to proceed. It seems like a more effective way to develop the driver than trying to use recycled code. Both attempt 2 and 3 (low level subroutines, and direct access) are pretty similar in approach. They both define registers using a base address and an offset. The main difference is when using the xspi_l.h header, I can only assume that the subroutines are doing what I expect them to do.

As everything stands now, I have identified three different approaches to the same problem. The first two approaches assume that the hardware is there for me to talk to. The third approach uses no subroutines and is more than likely to yield the proper result. If I had more time I would continue my work using the xparameters register calls, and depending on the extent of that time, I might even consider finding the source code for the GPIO driver and attempting to expand on that.
Bibliography

   http://courseware.ee.calpoly.edu/~jharris/research/super_project/be_pm_sp.pdf
Appendix

Code

/*
   *Direct memory access SPI driver.
   *
   *This driver will read and write data to the SPI hardware by writing directly to control
   *registers, transmit and receive registers, and status registers
   */

#include <stdio.h>
#include "/opt/pkg/petalinux-v0.30-rc1/hardware/user-platforms/Xilinx-Spartan3E500-RevD-edk91/microblaze_0/include/xio.h"
#include "/opt/pkg/petalinux-v0.30-rc1/hardware/user-platforms/Xilinx-Spartan3E500-RevD-edk91/microblaze_0/include/xparameters.h"

int busyFlag=0;
int I=0;
int oMsb,Lsb=0;
int iMsb,iLsb=0;
long config=0;
int rx=0;

int main(int argc, char **argv)
{

If (argv[1]==""){
    Printf("Use:  stid config=word rx=byte msb=byte lsb=byte  where config/lsb/msb values are hex  i.e

        stid config=196 msb=05 lsb=ff 
        rx =0 | no receive data 
        rx =1 | receive data is returned else 0 is returned, msb and lsb can be omitted. \n"");

For(i=1;i<20;i++{
    if(argv[i]=="config"){
        config=argv[i+1];
i++;
    }elseif(argv[i]=="rx"){
        rx=argv[i+1];
i++;
    }elseif(argv[i]=="msb"){
        msb=)argv[i+1];
    }
}
i++; } 
} elseif(argv[i]=="lsb"){ 
lsb=argv[i+1]; 
i++; } 
else{ 
Break; 
}

/****************************************
*This portion takes in the config data, msb, and lsb*
*and writes it out. If receive only operation, lsb, *
*and msb are left at default values. A transmit will*
*still be performed since this is the only way to    *
*generate the sck signal.                        *
* This will be a polling operation.             *
*****************************************

// write initial data to master SPIDTR register (i.e. 2000)  
// but only first 8 bits (i.e. 07 for MSB) not 7D0
XIo_Out32(XPAR_SPI_BASEADDR + 0x068, msb);  
// insure SPISSR register has all ones
XIo_Out32(XPAR_SPI_BASEADDR + 0x070, 0x1);  
// write data to SPI CR
// including setting manual slave select assertion enable (bit 24),  
// enable bit (bit 30), master transfer inhibit bit (bit23)
XIo_Out32(XPAR_SPI_BASEADDR + 0x060, config);  
// write to SPISSR to manually assert SS vector
XIo_Out32(XPAR_SPI_BASEADDR + 0x070, 0x0);  
// write data to SPI CR, but clear inhibit bit to start transfer
XIo_Out32(XPAR_SPI_BASEADDR + 0x060, 0x96);  

//read data in
iMSB=XIo_In32(XPAR_SPI_BASEADDR + 0x61);  
// poll status for completion
busyFlag = XIo_In32(XPAR_SPI_BASEADDR + 0x64);  
busyFlag = busyFlag & 0x4;  
while(busyFlag != 0x4)  
{  
    busyFlag = XIo_In32(XPAR_SPI_BASEADDR + 0x64);  
    busyFlag = busyFlag & 0x4;  
}  
// set master tranaction inhibit bit
XIo_Out32(XPAR_SPI_BASEADDR + 0x060, config);  
// write rest of data to master SPIDTR register (i.e. 2000)
XIo_Out32(XPAR_SPI_BASEADDR + 0x068, lsb);  
// write data to SPI CR, but clear inhibit bit
XIo_Out32(XPAR_SPI_BASEADDR + 0x060, 0x96);  

//read lsb
iLsb=XIo_In32(XPAR_SPI_BASEADDR + 0x61);
// poll status for completion
busyFlag = XIo_In32(XPAR_SPI_BASEADDR + 0x64);
busyFlag = busyFlag & 0x4;
while(busyFlag != 0x4)
{
    busyFlag = XIo_In32(XPAR_SPI_BASEADDR + 0x64);
    busyFlag = busyFlag & 0x4;
    // write all ones to SPIISR to deassert SS vector
    XIo_Out32(XPAR_SPI_BASEADDR + 0x070, 0x1);
    // disable devices
}

if(rx==1){
    return iMsb.iLsb;
} else{
    Return 0;
}

Figure 16