



# Electrical and Computer Engineering

**ECE453**

## **Custom IP Modules and Linux Device Drivers**

## 1. Lab Overview

The Altera SoC allows a designer to define custom digital peripherals in the FPGA of the Altera SoC. Altera calls these custom designs Intellectual Property blocks (IP). Each IP is configured and interfaced through a register set that defines the behavior of the device.

The ARM core(s) of the Altera SoC interface with an IP through an Avalon interface. The Avalon interface bus provides a common interface between the IP blocks and the ARM core(s). Qsys places all IP blocks in the physical memory map of the ARM core(s). The Linux kernel will interface with the IP blocks by reading and writing the physical memory address locations where the IPs have been mapped. Each IP block will require a custom device driver to allow software to interface with these devices.

A template IP block and associated Linux driver have already been developed for you. You will modify the IP block to add support to control the 8 WS2812B LEDs on your ECE453 mezzanine board.

## 2. Custom IP Blocks

When creating a custom IP block, you can describe the behavior of the IP block in a hardware description language such as Verilog or VHDL. The normal design process is to design and simulate your IP module separately before you add it to your SoC system using Quartus. This lab supplies you with an already working Verilog file that will be used during the course of this lab. This IP block will act as a reference for the design of your own IP blocks.

When you design your own IP block for this lab, be sure to **simulate** the behavior of your register set and verify the waveforms required to write to the WS2812B LEDs in ModelSim. Building a new SoC image takes roughly 10 minutes, so testing your Verilog by running it on the SoC is a very, very inefficient way to consume your time. Running ModelSim on the other hand takes only seconds.

You **SHOULD NOT** modify any of the input or outputs for the ece453 IP block. There are quite a few steps required to get a new IP block added to a SoC project that have been completed for you. If you change the input/output, you will need to complete these steps on your own, which is error prone and time consuming.

## 2.1. IP Block Review: Avalong Interface

The following section describes the contents of /home/ece453/shared/lab4c/SoC /ghrd\_top.v

1. The ece453 module consists of the Avalon interface (shown in red) and a custom interface designed for the ECE453 module.

```
10 module ece453(  
11     // signals to connect to an Avalon clock source interface  
12     clk,  
13     reset,  
14     // signals to connect to an Avalon-MM slave interface  
15     slave_address,  
16     slave_read,  
17     slave_write,  
18     slave_readdata,  
19     slave_writedata,  
20     slave_byteenable,  
21  
22     // ECE453 Interface  
23     gpio_inputs,  
24     gpio_outputs,  
25     irq_out  
26 );
```

The Avalon interface is an Altera supplied interface that is used to communicate between the ARM core and the FPGA on the SoC. The Avalon interface provides a standard interface that takes care synchronizing data transfers between the CPU and the FPGA.

Table 1: Avalong Interface Signals gives a short description of what each signal does

Signal Name	Description
clk	Clock used for the synchronous logic of the IP block
reset	Asynchronous Active High Reset
slave_address	Lowest 4 bits of the slave address. Used to decode up to 16 registers in the IP blocks register set
slave_read	Active high signal indicating a read operation
slave_write	Active high signal indicating a write operation
slave_readdata	32-bit output bus
slave_writedata	32-bit input bus
slave_byteenable	A 4-bit input used to indicate which bytes are being read/written.

Table 1: Avalong Interface Signals

***You should not remove, add, or rename any of the Avalon interface signals!***

## 2.2. IP Block Review: ECE453 Interface

In addition to the Avalon interface, the ECE453 IP block has three other inputs. These signals were added to allow the register set you design interface with both the ARM CPU and any external hardware that you need to control using the FPGA

```
10 module ece453(  
11 // signals to connect to an Avalon clock source interface  
12 clk,  
13 reset,  
14 // signals to connect to an Avalon-MM slave interface  
15 slave_address,  
16 slave_read,  
17 slave_write,  
18 slave_readdata,  
19 slave_writedata,  
20 slave_byteenable,  
21  
22 // ECE453 Interface  
23 gpio_inputs,  
24 gpio_outputs,  
25 irq_out  
26 );
```

Signal Name	Description
gpio_in	A 32-bit vector used to provide inputs to the ECE453 IP block.
gpio_out	A 32-bit vector used to provide outputs from the ECE453 IP block.
irq_out	A 1-bit vector used to indicate an interrupt condition.

***You should not remove, add, or rename any of the ECE453 interface signals!***

### 2.3. IP Block Review: ECE453 Register Set

You can examine the Verilog code for the register set to see the specific implementation details. The following are a summary of the 16 registers in the ECE453 IP block found in `/home/ece453/shared/lab4c/SoC /ghrd_top.v`.

Register Name	Device ID
Address	0x0
Description	This register is used is a read-only register that returns a device ID
Read/Write	R
Bit(s) 31:0	0xECE45300

Register Name	Control
Address	0x1
Description	This register is used to initiate sending data to the WS2812Bs on the ECE453 mezzanine card.
Read/Write	RW
Bit(s) 31:1	Reserved. Always read 0.
Bit(s) 0	Writing a 1 starts a write to the WS2812B LEDs. All 8 LEDs are written every time this bit is set to 1. This bit will auto clear after it is set.

Register Name	Status
Address	0x2
Description	This register is used to initiate when the WS2812B FSM is busy
Read/Write	RW
Bit(s) 31:1	Reserved. Always read 0.
Bit(s) 0	Busy Bit. Set to 1 when the WS2812B IP block is actively writing data to the LEDs. Set to 0 when inactive

Register Name	Interrupt Mask Register
Address	0x3
Description	This register is used to enable interrupts to the ARM interrupt controller. An interrupt is only enabled if the corresponding bit is set to a 1.
Read/Write	RW
Bit(s) 31	WS2812B Write Done
Bit(s) 30:0	Any change in value on GPIO_IN[30:0] will result in an interrupt.

Register Name	Interrupt Register
Address	0x4
Description	Current state of active Interrupts. Interrupts are cleared by writing a 1 to an active interrupt.
Read/Write	RW
Bit(s) 31	WS2812B Write Done is active.
Bit(s) 30:0	GPIO_IN[30:0] interrupt is active

Register Name	GPIO In
Address	0x5
Description	Holds the current value of the input pins to the ECE453 IP block. This register is read-only
Read/Write	R
Bit(s) 31:0	Data

Register Name	GPIO Out
Address	0x6
Description	Writes to this register place the corresponding value to gpio_outs
Read/Write	RW
Bit(s) 31:0	Data

Register Name	Unused
Address	0x7
Description	This is an un-used register. You can write a value to this register, but this value has no effect.
Read/Write	RW
Bit(s) 31:0	Data

Register Name	WS2812B 0-7
Address	0x8-0x15
Description	Holds a 24-bit color code for one of the WS2812B LEDs.
Bit(s) 31:24	Reserved. Always read 0.
Bit(s) 23-0	24-bit color code for WS2812B in the nth position.

## 2.4. Implementing WS2812B IP Block

Using this information, you will write a Verilog block that will generate the timing waveforms required for the the WS2812B LEDs. When bit 0 of the control register is set to a 1, your Verilog module will use the data in registers WS2812B 7 through 0 to update the 8 WS2812B registers on the interface board. While your module is updated the WS2812Bs, you will set the busy bit to a 1. After all 8 LEDs have been updated, set the busy bit to 0.

The connections for this block of Verilog have already been made for you. You can find the WS2812B datasheet on the course homepage.

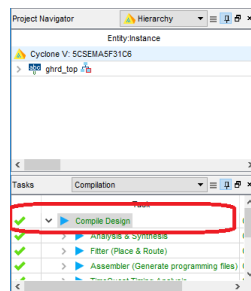
# 3. Testing Your Module on the DE1-SoC

## 3.1. Generating the DE1-SoC Programming Image

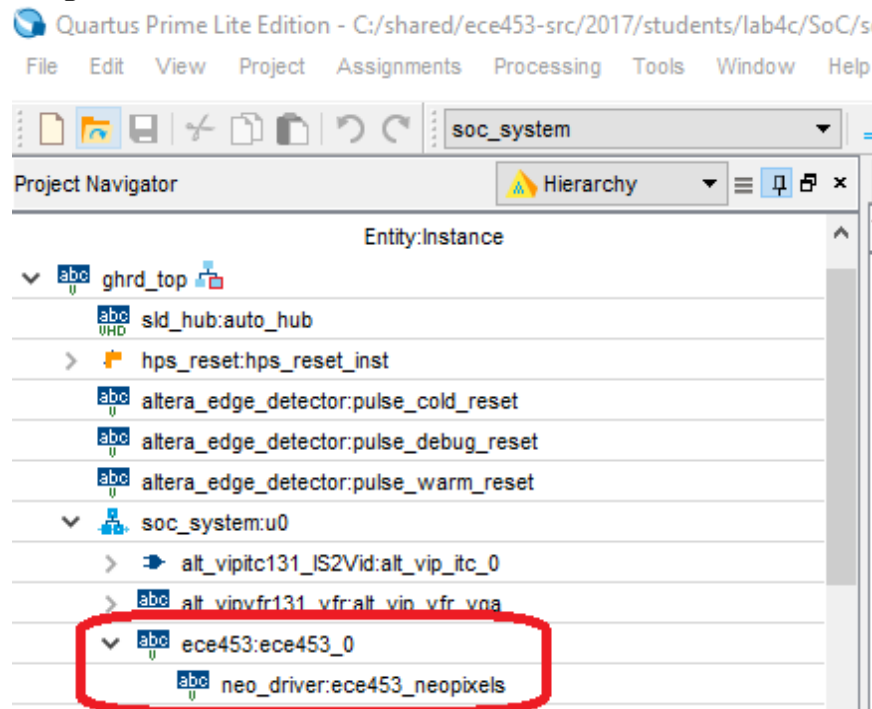
Be sure to simulate and verify your waveforms in ModelSim BEFORE you attempt to build and test your design on the DE1-SoC. When you have done this, proceed with building the programming image as described in Lab1c.

All of the steps in this section will take place in the Windows host.

3. Copy your neo\_driver module into `C:\shared\lab4c\SoC\ip\ece453\ece453.v`
4. Open Qsys You can find Qsys under “Tools” on the top menu. When the Qsys window opens, select soc\_system.qsys from the Open dialog.
5. Select the “Generate HDL” button near the lower right hand corner of the screen and then select “Generate”. This process takes about a minute and will finish with some warnings, but should not have any errors.
6. Select the “Finish” button to close Qsys. Quartus is going to issue a warning message to you after Qsys closes. You can hit OK and ignore this message.
7. Once this is done, you can double click on the “Compile Design” entry in the Tasks pane on the left. It will take roughly 10 minutes to build your system.



**Note:** If you are trying to modify a Verilog file in Quartus Project where you have already generated the files using Qsys, you will need to navigate to the ece453 module using the Project navigator.



Qsys makes a copy your initial Verilog file from `C:\shared\lab4c\SoC\ip\ece453\ece453.v` and places it into `C:\shared\lab4c\SoC\soc_system\synthesis\submodules`. Any changes you make to your Verilog file needs to happen in the `soc_system\synthesis\submodules` directory! If you clean the project, be sure to save off your Verilog files from the submodules directory or else they will be deleted.



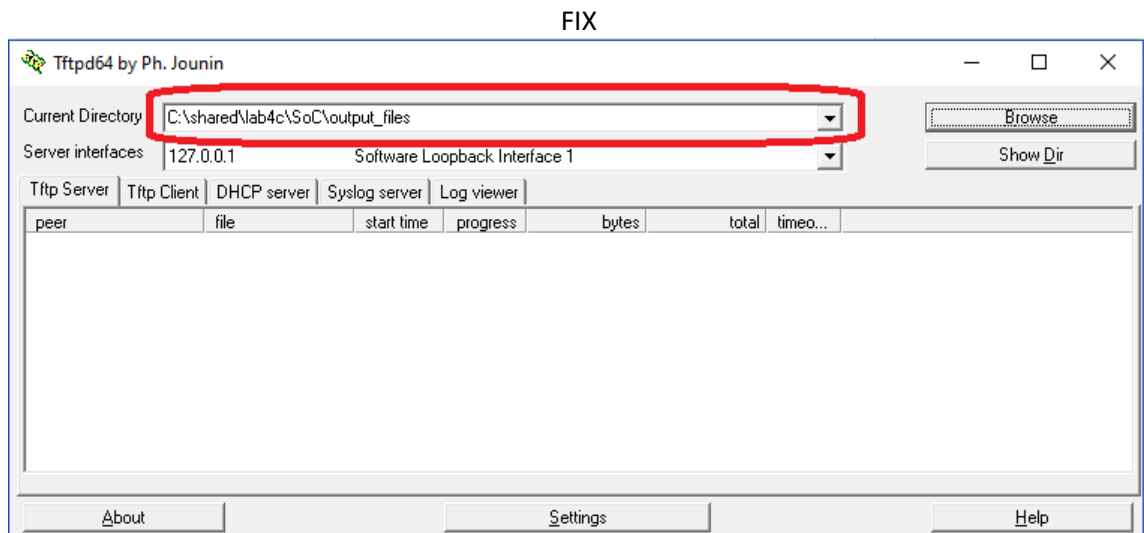
- When your Quartus build completes, you now need to convert the default programming file type (.sof) to a raw binary format (.rbf). This is done using File→Convert Programming Files.

In the window that pops up,

- Select a programming file type as Raw Binary File
- Select mode 1-bit Passive.
- Select a file name of output\_files/soc\_system.rbf
- Select Add File and navigate to output\_files/soc\_system.sof
- Click on Properties
- Select Compression
- Generate the File

### 3.2. Booting to Linux

- Open the TFTP64 server shortcut on your Windows PC. When the TFTP64 application opens, click on the “Browse” button and navigate to `c:\shared\lab4c\SoC\ece453\output_files`.



- Plug the microSD card into the DE1-SoC and power the board on.
- From the serial terminal, you should see that u-boot pulls the programmable logic programming file (soc\_system.rbf) from the TFTP server and then boots to Linux.

## 8. Writing a Device Driver

In order for the Linux kernel to be able to utilize an IP block, a Linux developer needs to write a kernel driver that provides access to the hardware. The kernel driver publishes an interface that allows user applications to send/receive/configure the hardware. You will examine a Kernel Loadable Module (KLM) that will allow you to dynamically load and unload a device driver that will communicate with your custom IP block in the programmable fabric of the SoC. The advantage of having a kernel loadable module is that you will not have to re-compile the entire Linux kernel every time you make a change to your driver.

### 8.1. Associating a Driver with Hardware

The Linux kernel can determine the appropriate driver to use for a given piece of hardware by examining the device tree. Each device in the device tree contains a string with a list of compatible drivers. The Linux kernel then matches device drivers to devices as long as both the device tree and device driver have the same compatible string listed.

You can see this below when we examine both the device tree and the kernel driver that was written for the ECE453 device block.

The image shows two screenshots of a code editor (GVIM) side-by-side. The top screenshot displays the `ece453_driver.c` file. A red box highlights the `static const struct of_device_id ece453_of_match[]` array, which contains a single entry with the compatible string `"uwmadison,ece453-1.0"`. The bottom screenshot displays the `soc_system.dts` file. A red box highlights the `ece453_0` device node, which also lists the compatible string `"uwmadison,ece453-1.0"` along with other hardware details like registers and interrupts. The matching compatible strings in both files demonstrate how the Linux kernel associates the driver with the hardware.

```
ece453_driver.c (/mnt/hgfs/shar.../2017/Staff/kernel/ece453) - GVIM
686 err_irq:
687     free_irq(irq, pdev);
688 err_release_region:
689     release_mem_region(res->start, remap_size);
690     return -1;
691 }
692
693
694
695 /* device match table to match with device node in device tree */
696 static const struct of_device_id ece453_of_match[] = {
697     { .compatible = "uwmadison,ece453-1.0" },
698     {}
699 };
700
701 /* platform driver structure for mytimer driver */
702 static struct platform_driver ece453_platform_driver = {
703     .driver = {
704         .name = DRIVER_NAME,
705         .owner = THIS_MODULE,
706         .of_match_table = ece453_of_match,
707         .probe = ece453_probe,
708         .remove = ece453_remove,
709     },
710 };
711
712 :set number 708,10 95%

soc_system.dts (/mnt/hgfs/share.../Staff/kernel/device_tree) - GVIM1
clocks = <clk_stream &clk 0>;
clock-names = "clock reset", "clock master";
max-width = <1024>; /* MAX IMAGE WIDTH type NUMBER */
max-height = <768>; /* MAX IMAGE HEIGHT type NUMBER */
bits-per-color = <8>; /* BITS PER PIXEL PER COLOR PLANE type NUMBER */
colors-per-beat = <4>; /* NUMBER OF CHANNELS IN PARALLEL type NUMBER */
beats-per-pixel = <1>; /* NUMBER OF CHANNELS IN SEQUENCE type NUMBER */
mem-word-width = <128>; /* MEM PORT WIDTH type NUMBER */
}; //end vip@0x100000100 (alt_vip_vfr_vga)

ece453_0: del_soc_ece453@0x100000000 {
    compatible = "uwmadison,ece453-1.0";
    reg = <0x00000001 0x00000000 0x00000040>;
    interrupt-parent = <0 40 4>;
    interrupts = <0 40 4>;
    clocks = <clk 0>;
}; //end del_soc_ece453@0x100000000 (ece453_0)

sysid_qsys: sysid@0x100010000 {
    compatible = "altr,sysid-16.0", "altr,sysid-1.0";
    reg = <0x00000001 0x00010000 0x00000008>;
    clocks = <clk 0>;
    id = <2899645186>; /* embeddedsw.dts.params.id type NUMBER */
};
135,22-26 11%
```

The device tree (`soc_system.dts`) is compiled into binary object (`soc_system.dtb`) which is loaded into memory by U-Boot. When the Linux OS begins to execute, it will examine device tree binary in memory to determine the location of hardware resources in the memory map and to associate hardware resources with the compatible kernel device driver.

## 8.2. Initializing the Driver

Once a device driver has been identified, the Linux kernel uses the `platform_driver` struct to determine how to initialize the associated hardware. The `platform_driver` struct lists information about the driver such as the compatible string and what functions should be called to initialize the hardware.

```
701 /* platform driver structure for mytimer driver */
702 static struct platform_driver ece453_platform_driver = {
703     .driver = {
704         .name = DRIVER_NAME,
705         .owner = THIS_MODULE,
706         .of_match_table = ece453_of_match},
707     .probe = ece453_probe,
708     .remove = ece453_remove,
709     .shutdown = ece453_shutdown
710 };
---
```

The probe function is used to initialize the hardware and allocate any kernel level resources that the device driver requires to operate correctly. The probe function in the provided driver does the following:

- Extracts the memory address of the IP block and maps it to a virtual address in the kernel (The variable `base_addr`).
- Determines the interrupt associated with the IP block registers a handler for the interrupts
- Creates entries in the SYSFS that provide an interface for user applications to interact with hardware

The remove function is used when the KLM is removed from the kernel. This normally disables the hardware.

## 8.3. Sysfs

The ECE453 kernel driver creates entries in the sys filesystem. The sys filesystem provides a simple interface that allows for user space applications to configure and interact with hardware. There are other commonly used methods for interacting with hardware, namely the `/dev` filesystem, but the `/sys` filesystem is the most straight forward and easiest to use.

The provided kernel driver creates a file for each register found in the IP block. This will allow the user to read and write the contents of each register. Each file that the driver will create needs to have an associated `kobj_attribute` data structure defined.

The snippet of code below shows the attribute for the control register in the ECE453 IP block.

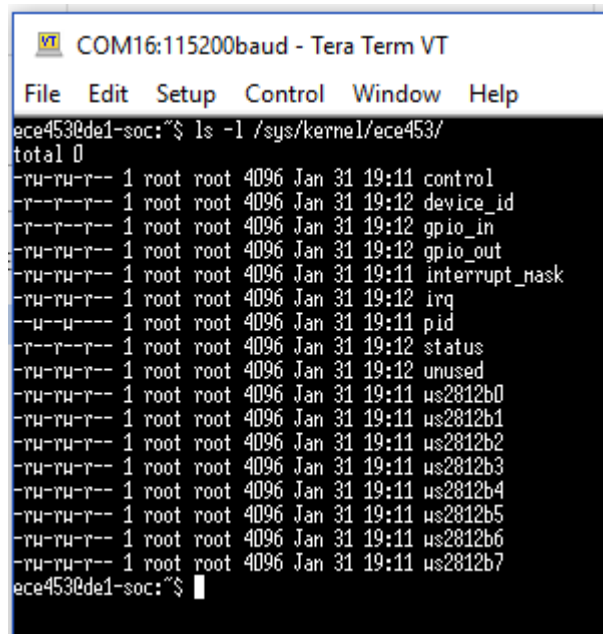
```
---
509 static struct kobj_attribute control_attribute =
510     __ATTR(control, 0664, ece453_read_control, ece453_write_control);
---
```

Here is a summary of what each field in the attribute does.

- The first parameter will be used to set the file name of the file see by the user. This parameter will tell the kernel to create a file named `control_status`. All of our sysfs files will be found in `/sys/kernel/ece453/` directory .
- 0664 sets the file permission of the file. The first 6 sets the file as read/write for the root user. The second 6 sets the file as read/write for the group that owns the file. The 4 sets the file a read only for anyone that is not the root user or in the root group.

This can be observed by looking at the resulting file permissions.

```
ls -l /sys/kernel/ece453
```



```

COM16:115200baud - Tera Term VT
File Edit Setup Control Window Help
ece453@de1-soc:~$ ls -l /sys/kernel/ece453/
total 0
-rw-rw-r-- 1 root root 4096 Jan 31 19:11 control
-r--r--r-- 1 root root 4096 Jan 31 19:12 device_id
-r--r--r-- 1 root root 4096 Jan 31 19:12 gpio_in
-rw-rw-r-- 1 root root 4096 Jan 31 19:12 gpio_out
-rw-rw-r-- 1 root root 4096 Jan 31 19:11 interrupt_mask
-rw-rw-r-- 1 root root 4096 Jan 31 19:12 irq
--u--u--- 1 root root 4096 Jan 31 19:11 pid
-r--r--r-- 1 root root 4096 Jan 31 19:12 status
-rw-rw-r-- 1 root root 4096 Jan 31 19:12 unused
-rw-rw-r-- 1 root root 4096 Jan 31 19:11 ws2812b0
-rw-rw-r-- 1 root root 4096 Jan 31 19:11 ws2812b1
-rw-rw-r-- 1 root root 4096 Jan 31 19:11 ws2812b2
-rw-rw-r-- 1 root root 4096 Jan 31 19:11 ws2812b3
-rw-rw-r-- 1 root root 4096 Jan 31 19:11 ws2812b4
-rw-rw-r-- 1 root root 4096 Jan 31 19:11 ws2812b5
-rw-rw-r-- 1 root root 4096 Jan 31 19:11 ws2812b6
-rw-rw-r-- 1 root root 4096 Jan 31 19:11 ws2812b7
ece453@de1-soc:~$

```

- The third argument is the function pointer for the read command. When a user attempts to read the file, the associated function is executed. If the file does not support reads, use NULL as the function pointer.
- The fourth argument is the function pointer for the write command. When a user attempts to write the file, the associated function is executed. If the file does not support writes, use NULL as the function pointer.

These snippets of code from the kernel driver show the read and write functions for the control and status register.

```

278 /*****
279 /*****
280 static ssize_t ece453_read_ws2812b_7 (
281     struct kobject *kobj,
282     struct kobj_attribute *attr,
283     char *buf
284 )
285 {
286     int val = ioread32(base_addr + ECE453_WS2812B_7_OFFSET);
287     return sprintf(buf, "%08x\n", val);
288 }
289

369 /*****
370 /*****
371 static ssize_t ece453_write_ws2812b_0 (
372     struct kobject *kobj,
373     struct kobj_attribute *attr,
374     const char *buf,
375     size_t count
376 )
377 {
378     int var;
379     sscanf(buf, "%xu", &var);
380     iowrite32( var, base_addr + ECE453_WS2812B_0_OFFSET);
381     return count;
382 }

```

Once all of the individual file attributes have been set, we need to keep a structure of all the attributes that the probe function will use to create each of the files.

```

554 static struct attribute *attrs[] = {
555     &device_id_attribute.attr,
556     &control_attribute.attr,
557     &status_attribute.attr,
558     &im_attribute.attr,
559     &irq_attribute.attr,
560     &gpio_in_attribute.attr,
561     &gpio_out_attribute.attr,
562     &unused_attribute.attr,
563     &ws2812b0_attribute.attr,
564     &ws2812b1_attribute.attr,
565     &ws2812b2_attribute.attr,
566     &ws2812b3_attribute.attr,
567     &ws2812b4_attribute.attr,
568     &ws2812b5_attribute.attr,
569     &ws2812b6_attribute.attr,
570     &ws2812b7_attribute.attr,
571     &pid_attribute.attr,
572     NULL /* need to NULL terminate the list of attributes */
573 };
574
575 /*

```

#### 8.4. Interrupts

The probe function reads the interrupt information from the device tree and registers a handler for the interrupt. The handler name is the second argument in the `request_irq` function (`ece453_irq_handler` in our case).

```
654  /* Register the interrupt */
655  irq = platform_get_irq(pdev, 0);
656  if (irq >= 0) {
657      ret = request_irq(irq, ece453_irq_handler, 0, pdev->name, pdev);
658      if (ret != 0) {
659          goto err_irq;
660      }
661  }
```

If we look at the contents of the handler, we see that the handler clears the hardware interrupt and MUST return a `IRQ_HANDLED` return value.

```
55  /******
56  /* Handles any interrupts that are generated by the IP module
57  /******
58  static irqreturn_t ece453_irq_handler(int irq, void *dev_id)
59  {
60      struct siginfo info;
61      struct task_struct *t;
62      int ret;
63      int active_irqs = 0;
64
65      // Read in the currently active interrupts
66      active_irqs = ioread32(base_addr + ECE453_IRQ_OFFSET);
67
68      // Acknowledge all the active interrupts
69      iowrite32( active_irqs , base_addr + ECE453_IRQ_OFFSET);
70
71      /* prepare the signal */
72      memset(&info, 0, sizeof(struct siginfo));
73      info.si_signo = SIG_TEST;
74      info.si_code = SI_QUEUE;
75
76      // Send the list of active IRQs back to the user.
77      info.si_int = active_irqs;
78
79      t = pid_task(find_pid_ns(pid, &init_pid_ns), PIDTYPE_PID);
80      if(t == NULL){
81          pr_err("no such pid\n");
82      }
83      else
84      {
85          ret = send_sig_info(SIG_TEST, &info, t);    //send the signal
86          if (ret < 0) {
87              pr_err("error sending signal\n");
88          }
89      }
90
91      return IRQ_HANDLED;
92 }
```

What you also should have noticed is all the comments about signals. What's all this business with signals? In the Linux operating system, when we want to alert another process that something has happened, we can send a signal from one process to the other. In our case, the KLM will signal the user application that something has happened and act as sort of a software interrupt.

So how does the Linux kernel know who to signal? There are two required pieces of information to make this happen. The first is the signal number. We're going to randomly use 44. The second piece of information we need is the user applications PID. Each task in the Linux kernel has a PID, or Program ID number, associated with it. We are going to require our user applications to register their PID with our driver so we know who to send the signal to. From a driver perspective, this requires us to add a PID entry to the /sys filesystem that the user application will set.

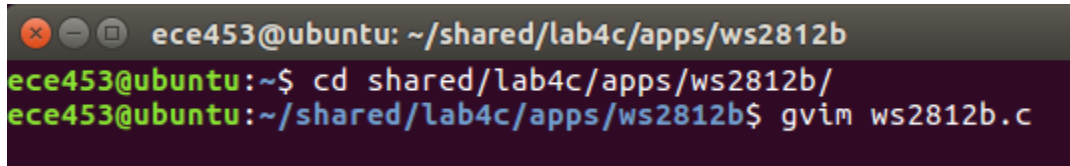
```
489 /*****  
490 /*****  
491 static ssize_t ece453_write_pid(struct kobject *kobj, struct kobj_attribute *attr,  
492                               const char *buf, size_t count)  
493 {  
494     sscanf(buf, "%d", &pid);  
495  
496     dbg("pid %d\n", pid);  
497  
498     return count;  
499 }
```

NOTE: Most device drivers do not send interrupt information back to the user application. The normal method of using the /dev filesystem is somewhat more complicated than what most projects require, so it will not be covered in as part of this class. Signals from the kernel to a user application are somewhat of a kludge, but this method is most commonly what students are looking to do with interrupts.

## 9. Writing an User Application

The user application is pretty straight forward. The user application will need to use the sys filesystem entries for the ECE453 IP block to configure the hardware and also to register a routine that is run when a signal is sent from the kernel.

```
cd ~/shared/lab4c/apps/ws2812b/  
gvim ws2812b.c
```



```
ece453@ubuntu: ~/shared/lab4c/apps/ws2812b  
ece453@ubuntu:~$ cd shared/lab4c/apps/ws2812b/  
ece453@ubuntu:~/shared/lab4c/apps/ws2812b$ gvim ws2812b.c
```

We can start by looking at registering the handler for the signal. The code below shows how the user application registers a function called `receiveData` to run when the kernel sends a signal to signal number 44 (`SIG_TEST`).

```
155 //*****  
156 //*****  
157 int main(int argc, char **argv)  
158 {  
159     struct sigaction led_sig;  
160     struct sigaction ctrl_c_sig;  
161  
162     // Set up handler for information set from the kernel driver  
163     led_sig.sa_sigaction = receiveData;  
164     led_sig.sa_flags = SA_SIGINFO;  
165     sigaction(SIG_TEST, &led_sig, NULL);  
166
```

The remainder of the application consists of helper functions that allow the application to configure the registers that are part of the IP block. These functions are very similar to the functions you wrote in a previous lab.

This application sets the color of the 8 WS2812B leds to green and then prints out a message that tells the user to press `KEY[0]`. After you press `KEY[0]`, the application will set the WS2812B LEDs to red and then exit. You DO NOT need to modify this file in any way.



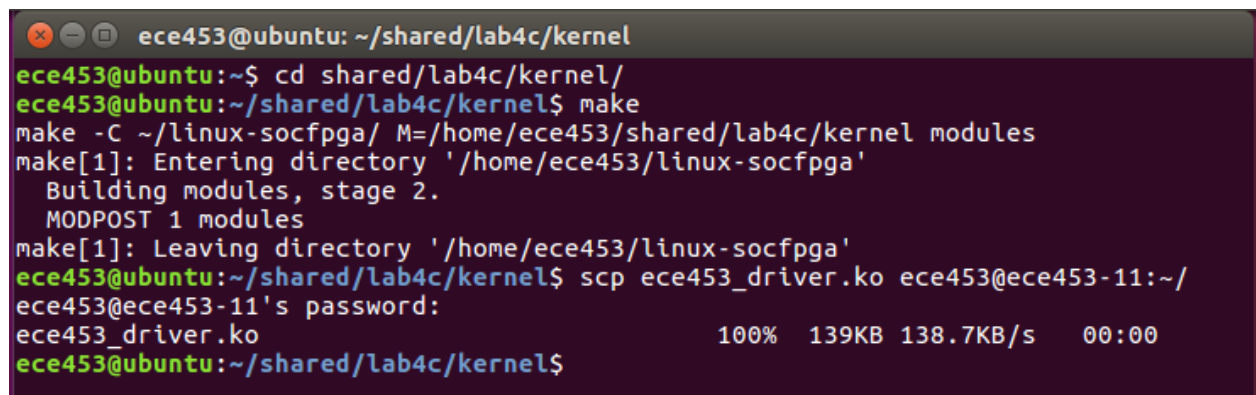
## 10. Testing the Application and Kernel Module

1. Copy your completed ece453.c, ili9341.c, and at42qt2120.c files from your lab3c directory into your lab4c directory

```
cp ~shared/lab3c/apps/ece453.c ~shared/lab4c/apps/ece453.c
cp ~shared/lab3c/apps/ili9341.c ~shared/lab4c/apps/ili9341.c
cp ~shared/lab3c/apps/at42qt2120.c ~shared/lab4c/apps/at42qt2120.c
```

2. Compile the kernel module from the Lab 4C software distribution and transfer it to the DE1-SoC board.

```
cd ~shared/lab4c/kernel/
make
scp ece453_driver.ko ece453@ece453-xx:~/
```

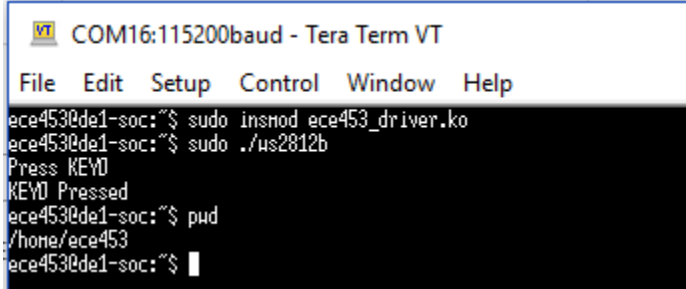


```
ece453@ubuntu: ~/shared/lab4c/kernel
ece453@ubuntu:~$ cd shared/lab4c/kernel/
ece453@ubuntu:~/shared/lab4c/kernel$ make
make -C ~/linux-socfpga/ M=/home/ece453/shared/lab4c/kernel modules
make[1]: Entering directory '/home/ece453/linux-socfpga'
Building modules, stage 2.
MODPOST 1 modules
make[1]: Leaving directory '/home/ece453/linux-socfpga'
ece453@ubuntu:~/shared/lab4c/kernel$ scp ece453_driver.ko ece453@ece453-11:~/
ece453@ece453-11's password:
ece453_driver.ko                                100% 139KB 138.7KB/s 00:00
ece453@ubuntu:~/shared/lab4c/kernel$
```

3. Compile the user application from the Lab 4C software distribution and transfer it to the DE1-SoC

```
ece453@ubuntu: ~/shared/lab4c/apps/ws2812b
ece453@ubuntu:~$ cd ~/shared/lab4c/apps/ws2812b/
ece453@ubuntu:~/shared/lab4c/apps/ws2812b$ make
arm-linux-gnueabihf-gcc -c -o ../obj/ili9341_fonts.o ../drivers/ili9341_fonts.c -I../include
arm-linux-gnueabihf-gcc -c -o ../obj/lcd.o ../drivers/lcd.c -I../include
arm-linux-gnueabihf-gcc -c -o ../obj/ece453.o ../drivers/ece453.c -I../include
arm-linux-gnueabihf-gcc -c -o ../obj/ili9341.o ../drivers/ili9341.c -I../include
arm-linux-gnueabihf-gcc -c -o ../obj/fonts.o ../drivers/fonts.c -I../include
arm-linux-gnueabihf-gcc -c -o ../obj/at42qt2120.o ../drivers/at42qt2120.c -I../include
arm-linux-gnueabihf-gcc -c -o ../obj/ws2812b.o ws2812b.c -I../include
arm-linux-gnueabihf-gcc -o ws2812b ../obj/ili9341_fonts.o ../obj/lcd.o ../obj/ece453.o ../obj/ili9341.o ../obj/fonts.o ../obj/at42qt2120.o ../obj/ws2812b.o -I../include
ece453@ubuntu:~/shared/lab4c/apps/ws2812b$ scp ws2812b ece453@ece453-11:~/
ece453@ece453-11's password:
ws2812b 100% 30KB 29.7KB/s 00:00
ece453@ubuntu:~/shared/lab4c/apps/ws2812b$
```

4. Load the kernel module on the DE1-SoC and then start the application.



```
COM16:115200baud - Tera Term VT
File Edit Setup Control Window Help
ece453@de1-soc:~$ sudo insmod ece453_driver.ko
ece453@de1-soc:~$ sudo ./ws2812b
Press KEY0
KEY0 Pressed
ece453@de1-soc:~$ pwd
/home/ece453
ece453@de1-soc:~$
```

You should observe that the WS2812B LEDs turn green. The application will then wait for you to press KEY0. Once you do, the LEDs should all turn red and the application will exit.

## 11. Writing Your Own Application

1. Make a copy of the of your WS2812B application directory

```
cp ~shared/lab4c/apps/ws2812b/ ~shared/lab4c/apps/ws2812b_student/
```

2. Modify the application so that both KEY[0] and KEY[1] generate interrupts when pressed.
3. Each time KEY[0] is pressed, increment the index and display that color on the WS2812B LEDs:

index	Color
0	0x000000
1	0x0000FF
2	0x00FF00
3	0xFF0000
4	0x00FFFF
5	0xFF00FF
6	0xFFFF00
7	0x202020

Repeat starting at 0

4. Each time KEY[1] is pressed, decrement the index and display that color.