DE-0 Nano SoC Driver Tutorial

Keith Lee

September 21, 2015

Introduction

his tutorial will guide you through the process of generating a Linux Kernel Module (LKM) and Application Program Interface (API) for the FPGA hardware on your DE-0. In order to access devices and peripherals in contemporary operating systems, a driver is required to to expose the hardware to the application. Memory is divided into two regions. User space memory is where, as the name implies, applications started by users are run. Kernel space memory is a region of memory addresses protected by the operating system's kernel, the primary runtime of the OS. This region includes all memory-mapped hardware interfaces, such as the AXI-connected FPGA devices on the DE-0.

In bare-wire programming, software running on a device without a supporting OS, drivers are not required because hardware addresses aren't protected from the running application. This means that the program can access everything connected to the processor or microcontroller without special permissions. Operating systems protect parts of memory to protect the correct operation of the kernel and its devices.

In order to access these devices, user programs use a dirver that reduces permitted interactions to a series of functions, methods and/or macros to a simplified C/C++ library. The two common components in a Linux driver are the LKM and API. The LKM is loaded as a special file by the OS and bridges the gap between user space and kernel space. This file has permissions, belongs to users and groups, and is visible in the file system like any other file but has special attributes that identify it as a device file. There are different types of device files with different access methods, but the easiest and most common of them is the character device. This is the device type we will focus on for this tutorial.

The API provides a convenient and, at best, human-readable collection of functions for developers. It is simply an abstraction of the driver's specialized access functions and is not strictly required to manipulate the device.

In this tutorial, we will develop driver to access the LEDs, pushbuttons and dipswitches on the FPGA side of the DE-0's physical I/Os. We will start by identifying the functionality required of the driver in order to use these devices correctly. Then, starting with a kernel module template, we will populate the required functions, macros and definitions to expose the operations identified in the previous section. In the final step, we will develop the API and a test program.

Step 0: Set up the Build Environment

Before you begin the tutorial, you must make sure the tools you need are on your PC. First of all, you will need a Linux system, preferably Ubuntu, for your build environment. If your PC is not running Linux, you can use VirtualBox to host a Linux virtual machine(VM).

Run the script called `de0-linux-buildenv-setup.sh` from within the tutorial's `files` directory by typing

```
./de0-linux-buildenv-setup.sh
```

This will install the necessary tools and libraries, download the socfpga-linux kernel source files, and build and install the ARM-Linux module libraries that match the distro on the DE0-nano SoC. It will ask twice for your superuser password. If your user account does not have superuser permissions, talk to your system administrator for assistance.

Additionally, you will need to install the SoC Embedded Design Suite from Altera's download website: `dl.altera.com/soceds`. With this installed, copy the directory "`altera/<version>/embedded/ip/altera/a` to `/usr/include/`. If you are using version EDS 15.0, `hwlib` will not contain the `socal` directory. That can be found in `.../hwlib/soc_cv_av`.

In order to flash the FPGA correctly when Linux boots, `de0_nano_soc.rbf` must be copied to the boot partition. When the SD card is inserted, in Windows, it's the only visible partition. In Linux, Two partitions are visible, but the smaller one contains the `.rbf` file. Overwrite the existing file with the one in the `/files` folder. Additionally, the `MSEL` dipswitches need to be reconfigured to allow the bootloader to flash the FPGA. These switches are located to the left of the *Cyclone V* chip and must be set to `000000`.

After running the above script, copying the `.rbf` file, setting the `MSEL` switches and installing the `/socal` directory, you should be able to complete the steps below and build a hardware driver for ARM Linux.

Step 1: Understanding the Device

For the purpose of this tutorial, we are combining three hard I/Os into a single device. For each I/O, there is an IP module in the FPGA's Quartus project connected to the hard processor system (HPS). The described hardware system is synthesized and compiled into a raw binary file (.rbf) and is placed in the boot partition of the Linux image on your DE-0's SD card. On startup, U-Boot, the bootloader, programs the FPGA using this file in much the same way as the Quartus serial programmer does over USB. The modules are mapped to a physical address in the system and the processor's memory management unit (MMU) maps physical addresses to virtual ones to facilitate the paging and caching behaviours common in most modern OSes.

The LKM is going to access the device based on its virtual address. This is its physical address plus a system-specified offset. The physical addresses of the I/O devices we want to access have been arranged in sequential order on the FPGA for simplicity and are included in the project's custom headers.

When the OS instantiates the driver, it will call an initialization function that registers the LKM with the kernel and configures any globals the driver may need. Likewise a cleanup function is called when the device is unregistered. Every time a user application tries to manipulate the driver, it opens the device file which calls a specialized function whose purpose is to maintain mutual exclusion between applications vying for the resource. When the application has completed the driver transaction, it closes the file. This triggers another function that releases the resource.

The three elements in our device have simple parallel I/O (PIO) interfaces. This means that every value can be retrieved from or written to the device controller simultaneously. Each pin's value is associated with one bit of the interface's register value. The three elements of the "`pio_ioctl`" device will require "read" operations and the LEDs will require a write operation. These ops will be accessed using an IOCTL function and will be individually identified.

Step 2: Setting up the LKM

The source files we will be editing to complete this tutorial are in the tutorial's `src` directory. In order to set up the module to perform its tasks, some elements must be added to `pio_ioctl.h`. This file has been populated with a few important details already, including some identifier constants, a few global variables and function declarations.

`MAJOR_NUM` is a required, unique, identifier the kernel uses to register the driver.
`DEVICE_NAME` is a human-readable title for the device.
`DEVICE_FILE_NAME` indicates where the character device file should exist in the file system.

The first additions the header file requires are a series of macros that generate unique numbers associated with the various I/O requests needed. Below the existing `#define` statements type:

```
#define IOCTL_SET_LED  _IOW(MAJOR_NUM, 0, char)
```

IOCTL_SET_LED will be the name of the value.
_IOW() is a macro function defined in <linux/ioctl.h> used to generate "write" request codes for device drivers.
MAJOR_NUM and the value of "0" means that the request code will be associated with the zeroth I/O request for device MAJOR_NUM.
char identifies the width of the return value.

On the next line add:

```
#define IOCTL_GET_LED _IOR(MAJOR_NUM, 1, char)
```

This means that request 1 for device MAJOR_NUM is a read operation of width char.

Next we will need read requests for the dipswitches and pushbuttons. a dipswitch read returns 4 valid bits, and the pushbuttons only 2, but they will be of width char anyway. Following the format in the statements above, add read op definitions for DIPSW and BUTTON.

The third set of elements we will add to the header file will help us identify the virtual address space the driver will access. We are going to define 2 values. The first value, PIO_REGS_BASE is the start address of our device in memory. It is the sum of the FPGA slave offset, I.E. the starting address of the memory mapped FPGA I/Os, and the base address of the first I/O element in our driver. Luckily, the "socal/hps.h" and "hps_0.h" headers contain helpful definitions to help us compute this.

```
\#define PIO_REGS_BASE (ALT_LWFPGASLVS_OFST + LED_PIO_BASE)
```

The second value, PIO_REGS_SPAN identifies the address range that encapsulates our devices. This can be defined as the value we just calculated, the top address in our array of I/Os, BUTTON_PIO_END, less the base address of our lowest pio, or LED_PIO_BASE. Add 1 and you have the span of the device's address space.

```
#define PIO_REGS_SPAN (BUTTON_PIO_END - LED_PIO_BASE + 1)
```

Step 3: Driver Behaviour

With the header complete, it is time to implement the required functions. Open up pio_ioctl.c. There are some predefined variables at the top:
Device_Open is a variable used by the device opener function to prevent multiple applications from accessing the device simultaneously.
Major is used to store the return value from the device registration process.
The functions will be reviewed in the next step.

Pointers to the three I/Os in our device will be needed. The values these pointers reference could change at any time and therefore should not be cached by the OS. For this reason, the pointer variables should be declared as volatile. Also, only a single pointer for each element should exist in memory so they should also be declared static.

So, add the following 3 lines:

```
volatile static char* led;
volatile static char* dipsw;
volatile static char* button;
```

The data structure, fops, holds the local names of the defined functions and associates them with the expected functions for correct driver behaviour. The functions required for the pio_ioctl driver are already added to fops and placeholder functions have been included in the body of the file.

The first function is device_open(). It is called when an application opens the character device file "/dev/pio_ioctl". The first thing this function needs to do is check that the resource is available. In order to do this we must confirm that Device_Open, the global variable we defined in the header file, equals 0. This is because, if it does, we are going to increment the variable so

that it is not 0.

```
if(Device_Open)
     return -EBUSY;
Device_Open++;
```

Next, the physical memory region associated with our driver must be mapped to a virtual address. For this, the `ioremap()` function is used as below.

```
virtual_base = (unsigned int) ioremap(PIO_REGS_BASE, PIO_REGS_SPAN);
```

`virtual_base` now points to the first entity in our device, the LEDs.

Next, we associate the pointers `led`, `dipsw` and `button` with the virtual address mapped to the I/O registers' location in physical memory.

```
led = virtual_base;
dipsw = virtual_base + 0x20;
button = virtual_base + 0x40;
```

The final step in opening the device is acquiring it from the kernel. This is done with the `try_module_get()` function.

```
try_module_get(THIS_MODULE);
```

The next function undoes the opener's work. First, it lets other applications know that `pio_ioctl` is free by decrementing `Device_Open`. We need to prevent dangerous memory leaks by redirecting the I/O pointers. For this we will use a sink variable. Somewhere before this function, declare `static unsigned long sink`. So assign the value of these pointers to the `sink` variable's address. Then it must release the virtual memory space we mapped above with `iounmap()` and release the device to the kernel with `module_put()`. See the complete function below.

```
static int device_release(struct inode *inode, struct file *flip)
{
    Device_Open--;
    led = (short*) &sink;
    dipsw = (short*) &sink;
    button = (char*) &sink;
    iounmap((void*)virtual_base);
    module_put(THIS_MODULE);

    return OK;
}
```

The next and most important function is `device_ioctl()`. This function performs the I/O operations that access the hardware. Beyond the pointer to the device file, this function takes two important arguments. The first is the op number. This will be one of the setter and getter codes defined in the header file. The second is the op parameter. In write operations, this is the value to be sent to the device controller. This argument should be `NULL`, or 0, for read operations.

A simple switch statement would be effective for this function but a chain of if statements could also work. We have defined four request codes in the header so we will need cases for each of these plus a default case, in the event of an erroneous request code, that returns -1.

The kernel has special functions for reading from and writing to I/O devices. These are `ioreadN()` and `iowriteN()` where `N` is the bit width you wish to access. So if we are accessing n 8-bit wide register, as is the case in `pio_ioctl`, we use `ioread8()` and `iowrite8()`. For example, to write to the LEDs, it would look like:

```
case IOCTL_SET_LED:
    iowrite8((char)ioctl_param % 0xff, led);
    return OK;
```

If we are reading the state of the LEDs, we would use:

```
case IOCTL_GET_LED:
    return ioread8(led);
```

Go ahead and fill in the rest of your switch statement for the rest of your I/O operations.

Two more functions remain. These are `init_module()` and `cleanup_module()`. They must be present and named as above. The OS uses these functions in order to correctly register the driver with the kernel. They have already been completed for you but make sure you understand how they work.

Step 4: Compiling and installing the LKM

Now that the `.c` and `.h` files have been completed, it is time to build the kernel module. Just as the program structure differs from traditional software, so does the compilation process. The Makefile that is used to compile the module is not the one in the project folder. It is actually the primary Makefile for the Linux kernel we downloaded in step 0. When we execute the command below, it will access the Makefile in the project directory in order to determine the files to be compiled together. To generate `pio_ioctl.ko`, our LKM, enter the following command from your project's `src/` directory:

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
   -C /lib/modules/3.13.0-00298-g3c77cbb9-dirty/build M=$PWD modules
```

Once you have compiled this module `pio_ioctl.ko` needs to be on the DE0's SD card. The absolute easiest way to do this is to connect it to your host Linux PC either directly or with a USB adaptor. For the purpose of this tutorial it doesn't matter which folder the file is copied to so we will place it in `/home/root`. This will make it easy to find when we telnet into the device. copying to the SD card requires superuser privileges again. You can start a file browser with superuser access in Ubuntu's GUI by entering the following command.

```
sudo nautilus
```

The new window will have root access to the SD card and will allow you to copy the `.ko` file to the folder mentioned above.

Now insert the SD card in the DE0, connect the included micro USB cable to the port next to the Ethernet and connect to it serially. For Windows users using VirtualBox, puTTY and mobaXterm will allow you to connect from your native environment, which is easier and more reliable than using VirtualBox's USB bridge. For Linux users, use the command:

```
sudo screen /dev/ttyUSB0 115200
```

Login as root and verify that our compiled module is present.

The next step is to install the module into the kernel. This is done using the `insmod` command.

```
insmod pio_ioctl.ko
```

This command runs the `init_module()` function, loading the driver into memory. Once it's running, it needs to exist as a file. The `mknod` command takes care of this:

```
mknod /dev/pio_ioctl c 122 0
```

`c` identifies the module as a character driver. `122` is the `MAJOR_NUM` we defined in the header file. `0` is the Minor. This exists so that multiple instances of the same module can be instantiated at the same time.

If you restart the DE0, you will have to re-enter the `insmod` and `mknod` commands to use the module.

1 Step 5: The API

Now that we have compiled and installed the kernel module for your driver, it is time to develop an API in order to easily access the module. The idea is to create a simple layer of abstraction for developers to use in their code. The API will be compiled as a library and included in the file structure on the DE-0's SD card.

Open `pio_ioctl_api.c` and review its contents. It includes several incomplete functions that will encapsulate the driver's functionality. `pio_open()` is a self-explanitory function. It uses a standard file access system call, `open()`, to initiate access to `pio_ioctl`. It returns a negative number if it fails to open the device, or an integer that identifies the file for the program. This number will be repeatedly used to access the driver as I/O operations are called.

```
int file_desc = open(DEVICE_FILE_NAME, 0);
```

If `file_desc` is negative, it may be helpful to print an error message to the console. Otherwise, the return value of the function should be `file_desc`.

`pio_close()` is even simpler. `file_desc()`, the function's argument is used by the `close()` function to close the device.

```
void pio_close(int file_desc)
{
    close(file_desc);
}
```

The remaining functions are also relatively self-explanitory. Each set and get function calls `ioctl()` which links to the `device_ioctl()` we completed for the LKM in step 3. This function takes at least two arguments. The first is the `file_desc` value returned from the `open()` function. The second is the request value of the I/O operation to be performed. These were defined in step 2. For read requests, these are the only arguments that `ioctl()` requires. Write requests, however, require one more: The value to be written to the device. As an example, `pio_set_led()` would contain the line:

```
int ret = ioctl(file_desc, IOCTL_SET_LED, led_state);
```

The return value for a write ioctl request is an error code, wherein negative values indicate that a problem occurred in the write process. For reads, the output is the requested data.

Armed with this knowledge, try and write the remaining get functions for the API.

Step 6: Building and using the API

The API is presented to the programmer as a library and a header file. The header, `pio_ioctl_api.h` in our case, exposes the public functions of the library to the developent environment. Compiling a shared Linux library is a two-stage process. The first step is to generate the `pio_ioctl_api.o` object file. This is the command:

```
arm-linux-gnueabihf-gcc -c -fPIC pio_ioctl_api.c -I./
     -I/lib/modules/3.13.0-00298-g3c77cbb9-dirty/include
```

The output of which will be `pio_ioctl_api.o`.

The second compile stage is to make the object shared so other binary objects can link to it. Also, there is a required naming convention among libraries, a `lib` prefix that must be applied to the output. The following command does both:

```
arm-linux-gnueabihf-gcc -shared -o libpio_ioctl_api.so pio_ioctl_api.o
```

In order to use the driver's library, it must be in the file system. The convention within Linux is to place libraries in either `/lib` or `/usr/lib` so we will put `libpio_ioctl_api.so` in `/lib`. And for header files, they tend to go in `/incluide` so that's where `pio_ioctl_api.h` will go.

Now, with the library installed and the LKM loaded – as per step 4 – the only thing left is to write a program that uses them to manipulate and monitor the physical I/Os on the DE0.