CprE 488 – Embedded Systems Design

MP-3: Embedded Linux

Assigned: Monday of Week 8
Due: Monday of Week 10
Points: 100 + bonus for target recognition capabilities

[Note: to this point in the semester we have been writing bare metal code, i.e. applications without anything more than the most basic system software ("standalone" mode in Xilinx terms). While this comes with several advantages due to its simplicity, it is fairly common for embedded systems to run an Operating System (OS). As the complexities of an OS are significant enough to merit their own course, in this class we are focusing more on practical aspects of porting an OS (Linux) to an (ARM-based) embedded system. Specifically, the goal of this Machine Problem is for your group to gain familiarity with three different aspects of embedded system design:

1. *Linux bring up – you will work through the stages of configuring, compiling, and booting for an open source Linux kernel targeting the Xilinx ZedBoard.*
2. *Linux driver development – you will adapt a template to develop a driver for a USB-powered missile launcher.*
3. *Linux system programming – you will write applications that target conventional Linux device drivers.*

1) Your Mission. It was bound to happen sooner or later. I am of course speaking of an alien invasion. Having already laid waste to the 100 most populous cities in the United States, the aliens have moved on to central Iowa. Holed up in Coover Hall, you and your friends have miraculously discovered the aliens' hidden weakness: foam-tipped darts! It's arguably no less plausible than the plot of *Independence Day* or even *Signs*.

The only thing standing between the aliens and their goal of taking over campus is you, your ZedBoard, and a USB missile launcher. Unlike simpler interfaces we have used earlier in the semester, USB devices are typically complex enough to require an Operating System to control their behavior. For that reason, you will port Linux to the ZedBoard, adapt a USB driver for your USB missile launcher, and develop an automated sentry system using the Linux code base and the camera system from MP-2. You only have two weeks of supplies remaining – so it is definitely time to get to work!

The Dream Cheeky Thunder USB Missile Launcher: humanity's last hope. Images not to scale.

2) Getting Started. As opposed to using an existing distribution that includes common binaries via a package management system, we are porting Linux directly from the main kernel source tree. This process is sometimes referred to as Open Source Linux (OSL)[1]. While this may seem like a daunting task, consider the following simplifying characteristics of our ZedBoard:

- There is a well-defined solid-state boot sequence for the system, namely when jumpers "MIO4" and "MIO5" are connected to 3V3, a small boot ROM copies code from a file named `BOOT.BIN` on the sdcard and starts executing instructions starting from address 0.
- The Processing System side of the Zynq FPGA is fairly conventional: two ARM cores with commonly-found features (e.g. timers, DMA), and interfaces (e.g. DRAM, USB, UART). Ignoring the programmable logic side of the Zynq, we can make use of the very stable ARM branch of the Linux kernel.
- The Xilinx tools provide some automation capability to specify drivers for IP cores instantiated in programmable logic. In the absence of drivers, the simple memory-mapped nature of the AXI bus allows us to access our soft-IP core peripherals via explicit pointer references to `/dev/mem`.

The files to get you started with this MP are located here:
- **MP-3.zip**: Has Documentation, starter exported hardware platform (.xsa, and bitfile) with LEDs, switches, and buttons configured, and Linux devices driver starter code.

Unzip MP-3.zip, and become familiar with the directory structure, and where information is located.

For the first part of this lab, you will be primarily following the **PetaLinux Tools Reference Guide (ug1144)** to configure, build, and boot a Linux kernel on to the Zedboard.

Petalinux development: This nicely bundles the needed embedded Linux tools for compiling a Linux kernel, and performing Linux device driver development, and application development. **Note: this toolset only runs in Linux**. We have installed these tools for easy access onto the **"Engineering Linux**" VDI machines. Of course, you can install petalinux on your own personal machines if you would like (just make sure to install 2020.1)

- **"Engineering Linux" VDI machine**:
  - Directions for installing the VDI software: https://etg.ece.iastate.edu/vdi/
  - Or you can go here: https://vdi.engineering.iastate.edu
    - This link appears to give you the option to launch the VDI on a client running locally on your machine, or you can access it via a webpage.
- **Use your X drive:** Make sure to use you X: instead of your U: for MP3. Building the Linux kernel may take up to 100 GB of space, and you have MUCH more space on you X:. I think your U: maxes at a few GB.

While the **PetaLinux Tools Reference Guide (ug1144)** is the main location for learning how to use the tools, please refer to the following quick reference notes found in the MP3 release folder: **petalinux-quick-ref.txt**

In your writeup, briefly explain how this Make process was configured to appropriately use a cross-compiler targeting the ARM architecture.

[1]Suggested motto: "Open Source Linux – for when Gentoo is too boring and simplistic"

3) Booting Linux. Once you have built and packaged your Linux Kernel using Petalinux, copy the files required to boot Linux to an SD card: `BOOT.BIN, image.ub, boot.scr`

After copying these files to the top level of your ZedBoard's SD card, connect it back to the board. Making sure that PuTTY is open, power on the board. In your writeup, provide an annotation explaining some of the boot messages that print out as Linux is booting on your ZedBoard. The output is over 200 lines long, so annotate at least 30 of the messages in your report. Note that the PS-RST button on the ZedBoard is quite useful for this assignment, as it will restart the boot process from the beginning. There is no need to use the USB JTAG cable to download bitfiles or executables.

Note: Once booted you will be prompted for a username and password: `root` for both.

Plugging in the USB missile launcher into the USB-OTG port announces that a new device has been found and recognized. Include these kernel messages in your writeup and attempt to provide some meaning to the output.

4) Controlling Simple Peripherals. While the Linux kernel we have now successfully ported to the ZedBoard does provide simple drivers for the AXI-GPIO peripherals we have connected to the buttons, LEDs, and switches in the programmable logic, there is no need to use these. We know from MP-0 that these peripherals are simple memory-mapped devices.

In most embedded Linux configurations, access to physical memory is provided using a character device file located in `/dev/mem`. While the "correct" way to access devices is to use their associated driver, by directly accessing `/dev/mem` as root we can read or write from any location in memory. Later we will use the *mmap()* system function to create a new mapping between the physical memory address space and the virtual memory space of our applications, but for now we can do so by calling the `devmem` utility:

```
zynq> devmem 0x0
0xE59F0000
```

Read up on the `devmem` utility, and confirm via experimentation that you are able to read the state of the buttons and switches, and write patterns to the LEDs. Create a simple shell script that has the following functionality:

- The script continuously polls the buttons and switches.
- When no buttons are pressed, the LEDs should reflect the current state of the switches.
- When 1 button is pressed, the LEDs should reflect the inverse of the current state of the switches.
- When more than 1 button is pressed, the LEDs should be all 1s.

Note that the sdcard is automatically mounted in the `/mnt` directory on the ZedBoard, and this is the only solid-state storage available for you to save files. So place this script (called `LEDfun.sh`) on the sdcard and submit it with your MP-3 submission .zip file.

[1]Suggested motto: "Open Source Linux – for when Gentoo is too boring and simplistic"

5) USB Driver Development. Linux differentiates between drivers compiled directly into the kernel (static kernel modules) and so-called loadable kernel modules. These loadable kernel modules have the advantage of flexible development since they do not require a full kernel rebuild and reboot. The main disadvantage over the static kernel is a slight performance penalty associated due to more fragmented kernel memory, but for our purposes this overhead is trivial.

When we plugged the USB missile launcher into our USB-OTG port in step 4), we noticed that it was "claimed" by the hid-generic driver (which is typically used for generic USB human interface devices such as joysticks and keyboards). While there are many ways we could modify the kernel to prevent this from happening, for purposes of simplicity, we will manually unbind this driver each time we reboot Linux (it will save you time and headaches if you create another quick shell script for the commands in this section):

```
zynq> echo "1-1:1.0" > /sys/bus/usb/drivers/usbhid/unbind
```

The command structure for many USB devices is complex enough that a simple memory-mapped interface is not appropriate. For this reason we will be creating a simple character device driver and dynamically loading it as a kernel module. It is recommended you (again) read through Chapters 1-3 and 13 of the *Linux Device Drivers* book before proceeding. Post on Blackboard any questions you may have as you read through these chapters. Given the limited capabilities for debugging and the lack of recovery when something goes wrong, the USB driver module development is the most challenging part of this assignment.

The Linux kernel source includes a skeleton code for a USB driver, which we have placed in the `MP-3/drivers` directory. Copy this file (`usb-skeleton.c`) to a file named `launcher_driver.c` and start editing. In your write-up, summarize the changes that you've made. Some pointers and requirements:

- You do not need to reverse engineer the missile launcher USB protocol (although that would also make for a fun assignment). The file `launcher_commands.h` includes the command codes – of which we will only be using UP, DOWN, LEFT, RIGHT, STOP, and FIRE.
- The Makefile is also already in place, which compiles the `launcher_driver` kernel module as well as an example user application that you can use to test your driver.
- Replace **every** reference to the "skel" skeleton device with an appropriate launcher-specific equivalent. For starters, line 27 defines a USB_SKEL_VENDOR_ID. Instead, use the LAUNCHER_VENDOR_ID provided in `launcher_commands.h`. Do not do a global replace, make these changes manually so you gain some understanding of how the skeleton code works.
- We don't need to use any of the "read" functionality, but it's probably more effort to properly remove those callbacks than it is to just ignore them. Same with the "write_bulk" functionality.
- Our device does not have any bulk or isochronous endpoints, and just 1 input interrupt endpoint. Consequently you can simplify the code in *launcher_probe()* (or just comment out the error handling) to setup a control URB.
- The "write" operations are packed into a control URB and sent to the device. In function *launcher_write()*, you will need to receive the user data and call *usb_control_msg()*:
  - The call to *usb_control_msg()* should replace the calls that fill and submit a bulk urb in function *launcher_write()*.
  - We are always sending 8 byte messages, with byte 0 being 0x02 (LAUNCHER_CTRL_CMD_PREFIX), byte 1 corresponding to the user input byte (the command), and the rest of the bytes being 0x0.
- Add a valid MODULE_AUTHOR and MODULE_DESCRIPTION.

[1]Suggested motto: "Open Source Linux – for when Gentoo is too boring and simplistic"

The following blog provides some details on the type of changes to the USB skeleton driver code you will need to make: http://matthias.vallentin.net/blog/2007/04/writing-a-linux-kernel-driver-for-an-unknown-usb-device/ (note – the code listed there will not directly work for our device, so do not attempt to just copy/paste).

Once you have some confidence that your driver is correct, copy the launcher_driver.ko and launcher_fire executable to your SD card, and after Linux has booted on the ZedBoard, unbind the usb-hid driver as directed above, and insert your kernel module into the running kernel:

```
zynq> insmod <location-of-your-ko-file>/launcher-driver.ko
```

If everything's working, the new character device driver should show up as `/dev/launcher0`. Before continuing, place your face 5 inches away from the missile launcher and confirm that the following command has the appropriate effect:

```
zynq> echo -en '\x10' > /dev/launcher0
```

This method of accessing the character device for our launcher, while valid, is limited to the speed of the shell environment. A lower latency method of access can be had by directly accessing the file `/dev/launcher0` using standard C system calls in our application, as is illustrated in `launcher_fire.c`. Walk through this file and in your writeup, describe how it generally works.

This code only just fires the launcher. Create a new application (`launcher-fire-buttons.c`) which uses the 5 pushbuttons to direct the launcher, with the center button acting as the "fire" button. The *mmap()* function will allow you to map the physical memory address for the buttons, so check the man page to learn how to use it correctly. Also, make sure to update your Makefile to compile this application. Part of your team can skip ahead to part 7) at this point, as we are heading back to VIVADO/Vitis and modify the base system.

6) Sentry System. So far we have been able to keep the aliens at bay due our superior engineering skillset and attentive patrols. To return to a life of normalcy, however, it would be preferable to have an automated sentry system to drive the aliens further away. Fortunately, we have the camera infrastructure from MP-2 at our disposal!

1. Open your MP2 in Vitis
2. Create a first stage boot loader project
    a. File -> New -> Application Project
    b. Select the platform you want to use (e.g., color pipeline platform)
    c. Select the "Zynq FSBL" template option, in the past we have typically selected the "Hello World" option. Note, if you get a message about not having a xilffs component you will need to add it to your platform's BSP (i.e., the platform.spr of the Platform project you want to use)

[1]Suggested motto: "Open Source Linux – for when Gentoo is too boring and simplistic"

For better or for worse, the various peripherals we used in the MP-2 assignment (color filter array, timing controllers, color resampling) do not have much in the way of driver support in the Linux kernel. For this reason there is little value in updating our device tree to represent the MP-2 system. We can however, access the framebuffer memory in Linux by adapting the *mmap()* approach from the previous section.

There are a few complicating factors when integrating the MP-2 camera system into MP-3:

1. We need to run the *camera_app* software infrastructure from MP-2, but our `BOOT.BIN` is already associated with launching the `u-boot.elf` for Linux. Make sure this sentence makes sense before continuing.
2. One workaround is to directly copy the files from *camera_app* into a new FSBL application, so that the initialization of the camera-related peripherals can occur before Linux hand-off. Do that – the appropriate place to call your *camera_app* functionality is in function *FsblHookBeforeHandoff()* in `fsbl_hooks.c`.
3. The first compile error you'll receive when doing so relates to there being multiple defined *main()* functions. Rename your *main()* in `camera_app.c` to *camera_main()*, and call that directly in *FsblHookBeforeHandoff()*. Remove any calls to your *camera_loop()* from MP-2.
4. Congratulations – more compile errors! These can be fixed via the suggestions on the class wiki for MP-3.
5. Once you have fixed all the errors, Vitis will provide you with a .elf file that you can use in the Petalinux flow. Previously when you did the petalinux package step, you just used the default for the `--fsbl` option, now you will provide the .elf file you just created.

Re-generate your `BOOT.BIN`, (i.e., Update you PetaLinux project using the appropriate .xsa, .bit, and .elf files) and load up Linux. Create a new application, called `launcher_fire_camera.c`, which uses the camera framebuffer and USB missile launcher driver to act as an automated sentry system. Your sentry should have the following features (in your writeup, describe the general algorithm you implemented):

- We have various dartboard-style targets printed out that your application should be able to detect. There are 7 variants (red, blue, green, cyan, magenta, yellow, and black) – select whichever color you believe will be easiest to detect in software.
- Color detection is not as simple as checking "if (pixel == RED)". Printing ink on paper is not an exact science, and even if it were, the targets are laminated and are somewhat shiny. Err on the side of false positives versus waiting for precise matches.
- There are three framebuffers in the system, but since we are not doing in-loop camera processing (MP-2 style), you can read any of the three buffers using *mmap()* and expect to get relatively accurate data.
- Your application needs to take this detection result (presumably an X,Y framebuffer coordinate) and direct the missile launcher to fire in that location. Don't over-engineer this part either – the darts when fired have a pretty wide margin of error, at least after they've been heavily played with by toddlers over a long weekend.
- You can assume a slow moving target at a 3 foot range.

What to submit: a .zip file containing your modified source files (your final `LEDfun.sh`, `launcher_driver.c`, `launcher_fire_buttons.c`, and `launcher_fire_camera.c`) and your writeup in PDF format containing the highlighted sections of this document. In the Blackboard submission, list each team member with a percentage of their overall effort on MP-3 (with percentages summing to 100%).

[1]Suggested motto: "Open Source Linux – for when Gentoo is too boring and simplistic"

What to demo: at least one group member must be available to demo the current state of your implementation. A full demo score requires a system that can fire the USB rocket launcher based on some camera-driven feedback, but partial credit will be given for effort. Be prepared to briefly discuss your source code.

BONUS credit. MP-3 has two bonus point criteria. The first is detection *accuracy*. How far away can your camera system detect one of the provided targets (assuming a slowly moving target)? For each foot of distance beyond the 3 foot minimum in which you can consistently detect a target, you will earn 2 bonus points. (up to 20 bonus points)

The second MP-3 bonus point criterion is based on sentry *precision*. Assuming accurate object detection at the 3 foot minimum, can you reliably hit the target with your launcher? (10 bonus points).

Each group is limited to 100 bonus points for the entire semester.

[1]Suggested motto: "Open Source Linux – for when Gentoo is too boring and simplistic"