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!

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:

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

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

- 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`.

Because of the sheer number of files associated with Linux porting and driver development, for MP-3 it is recommended you work out of the `/tmp` directory on the remote Engineering servers, since file access should be considerably faster there than in your CyFiles space and there should be no capacity concerns as well. Only do so temporarily (pun absolutely intended), as the `/tmp` folder does get cleared periodically. Perform a check-out and peruse the directory structure (note the back-quotes that are used to take the output of the "whoami" command in order to create a unique directory name in `/tmp`):

```
>> cd /tmp; mkdir cpre488-`whoami`; cd cpre488-`whoami`;
>> svn checkout https://source.ece.iastate.edu/svn/cpre488/MP-3
```

After sourcing the `setup.sh` file (which has been updated slightly to add a couple of useful features), open up XPS and create a design using the Base System Builder Wizard. Place the project under `MP-3/system/system.xmp`, target the Avnet ZedBoard, and keep all of the other options their default. While we wait for that project to build (don't forget to select the "Export to SDK" option), take a look at the Linux source tree provided with the repository:

```
>> cd MP-3; tar –xzvf linux.tar.gz
>> cd linux; ls
```

In this directory, there are two sub-directories: the source for the U-Boot (Universal Bootloader), and the source for the Linux kernel. Let's make the U-Boot first:

```
>> cd u-boot-xlnx
>> make zynq_zed_config; make
```

In your writeup, briefly explain how this Make process was configured to appropriately use a cross-compiler targeting the ARM architecture. Before we move on from U-Boot, there are two small steps that will facilitate things later:

```
>> cd tools; export PATH=`pwd`:$PATH
>> cd ../; cp u-boot ../../bootimage/u-boot.elf
```

The first command provides us with a link in our `$PATH` to the `mkimage` utility, which is needed during the kernel build process. The second command just copies the output u-boot executable to a directory where we are storing the intermediate files needed to prepare our SD card for Linux boot.

3) My First Kernel. Depending on the load on your system, your U-Boot build should complete at approximately the same time as your XPS project export. When starting SDK, make sure to select the `MP-3/sw` directory for your project workspace, and then switch back to your terminal window:

```
>> cd ../linux-xlnx
>> make xilinx_zynq_defconfig
>> make menuconfig
```

After running this last command, you should be presented with a "graphical" menu interface for configuring your kernel. Since we previously ran "make xilinx_zynq_defconfig", the configuration is already close to what we need. Make the following two changes:

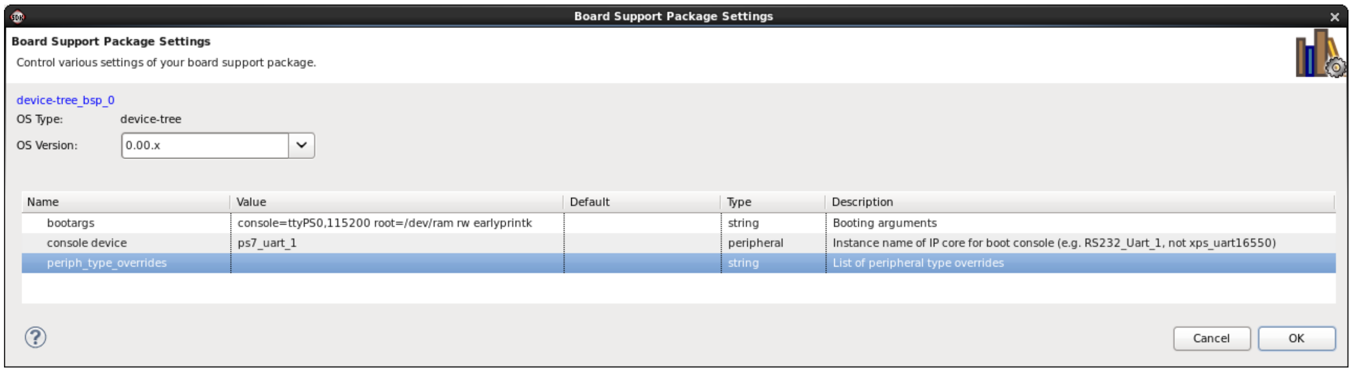
- Under "Device Drivers -> USB support", select the "USB verbose debug messages" option (pressing 'Y' selects a given option).
- Under "Device Drivers -> USB support", select the "Announce new devices" option.

Make sure to save as you exit, after which you are ready to make the kernel. Type the following command precisely, or your computer might literally explode:

```
>> make UIMAGE_LOADADDR=0x8000 uImage
```

Assuming there is no smoke emanating from your workstation, let's turn our attention back to SDK while we wait for the (lengthy) kernel build process to complete. Xilinx provides as part of their support for Linux development a custom Board Support Package (BSP) that enumerates the devices in your system in a way that the Linux kernel can load at boot. First, under "Xilinx Tools -> Repositories", add both the `MP-3/repository` directory as well as `MP-3/repository/ProcessorIPLib`.

Next, select "New -> Board Support Package", and select the "device-tree" option to generate a flat device tree. When the BSP settings window comes up, type in "`console=ttyPS0,115200 root=/dev/ram rw earlyprintk`" (without quotes) under the "bootargs" option, and select `ps7_uart_1` under the "console device" option. Before hitting "OK", confirm that your device tree configuration window looks like the following:



Board Support Package Settings

Control various settings of your board support package.

device-tree_bsp_0

OS Type: device-tree

OS Version: 0.00.x

Name	Value	Default	Type	Description
bootargs	console=ttyPS0,115200 root=/dev/ram rw earlyprintk		string	Booting arguments
console device	ps7_uart_1		peripheral	Instance name of IP core for boot console (e.g. RS232_Uart_1, not xps_uart16550)
periph_type_overrides			string	List of peripheral type overrides

In your writeup, explain what each option to the "bootargs" parameter signifies. The output of the device-tree BSP generation is a single file called `xilinx.dts`, which is conveniently located in the `libsrc/device-tree-v0_00_x/` sub-directory under the device-tree project in SDK. Open up a new terminal (as presumably the kernel is still building), and copy this file to somewhere more sensible so we don't lose track of it:

```
      >> cd MP-3;
      >> cp sw/device-tree_bsp_0/ps7_cortexa9_0/libsrc/device-
tree_v0_00_x/xilinx.dts bootimage/
```

Inspect the section in your `xilinx.dts` file pertaining to the LEDs in your system. In your writeup, provide a brief description for each sub-entry under this component. Some aspects will be obvious, and for others just make an intelligent guess. There is an online tutorial on using the device tree for the Zynq processor that you may find helpful as you go through this part: http://xillybus.com/tutorials/device-tree-zynq-1.

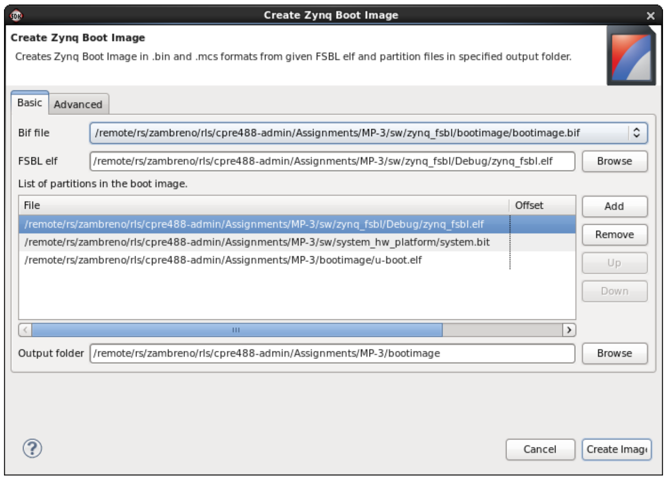
Assuming your kernel build is now (finally) ready, we want to 1) grab the output kernel binary, and 2) convert the device tree to a simple binary format:

```
      >> cp arch/arm/boot/uImage ../../bootimage/
      >> cd ../../bootimage;
      >> ../linux/linux-xlnx/scripts/dtc/dtc -I dts -O dtb -o devicetree.dtb
xilinx.dts
```

As there are so many places in this Linux build process where a small typo can lead to large problems, please work with your TA to get past any issues you do encounter early on. Note that we have simplified the process somewhat (this is not quite Linux From Scratch), as we have provided a pre-built RAMDisk image that contains the BusyBox utilities for ARM.

4) Booting Linux. As we have seen when producing `BOOT.BIN` files for demo purposes, the ZedBoard can make use of a First Stage Boot Loader (FSBL) executable to configure the programmable logic on the Zynq FPGA and launch an executable. Follow the conventional steps (available on the class wiki page) to build an FSBL design for our Linux boot process. Since the executable we are bundling is not an existing SDK software project, you will need to tweak the wiki instructions slightly:

- Call your project "zynq_fsbl", and target a new Board Support Package called "zynq_fsbl_bsp". Later in this assignment we will be using the system_bsp from MP-2, and we do not want to mix up the two library packages.



Create Zynq Boot Image

Creates Zynq Boot Image in .bin and .mcs formats from given FSBL elf and partition files in specified output folder.

Bif file: /remote/rs/zambreno/rls/cpre488-admin/Assignments/MP-3/sw/zynq_fsbl/bootimage/bootimage.bif

FSBL elf: /remote/rs/zambreno/rls/cpre488-admin/Assignments/MP-3/sw/zynq_fsbl/Debug/zynq_fsbl.elf

List of partitions in the boot image.

File	Offset
/remote/rs/zambreno/rls/cpre488-admin/Assignments/MP-3/sw/zynq_fsbl/Debug/zynq_fsbl.elf	
/remote/rs/zambreno/rls/cpre488-admin/Assignments/MP-3/sw/system_hw_platform/system.bit	
/remote/rs/zambreno/rls/cpre488-admin/Assignments/MP-3/bootimage/u-boot.elf	

Output folder: /remote/rs/zambreno/rls/cpre488-admin/Assignments/MP-3/bootimage

Your Boot Image creation screen should look something like this.

- We will be actively monitoring the boot process, so once the project is auto-generated, add the line "#define FSBL_DEBUG_INFO" at line 74 of `fsbl_debug.h`. Annoyingly, SDK does not auto-rebuild projects if only the header files have been changed, so make sure to manually rebuild your zynq_fsbl project to incorporate this change in the output executable.
- When creating the Zynq Boot Image, SDK will only be aware of two of the three binaries needed for the FSBL boot process (`zynq_fsbl.elf` and `system.bit`). Manually add the `u-boot.elf` as the third partition in the boot image – the order is FSBL, then bitfile, then U-Boot executable. We had previously placed the u-boot.elf executable in your `MP-3/bootimage` directory.
- Point the output folder to `MP-3/bootimage`.

Note that this option in SDK is a wrapper for a utility called "bootgen", and it can be run directly from the command-prompt to simplify the process of building more `BOOT.BIN` files in part 7). In any case, at this point you have everything needed to boot Linux on your ZedBoard:

1. The BusyBox RAMDisk is already located in `MP-3/sdcard`.
2. The `BOOT.BIN` file which was created in the previous step and is named `u-boot.bin`. From the bootimage directory, copy this file over to your sdcard directory.

   ```
   >> cd bootimage; cp u-boot.bin ../sdcard/BOOT.BIN
   ```

3. The uImage kernel image and devicetree blob should already be in `MP-3/bootimage`. Copy those over to your sdcard directory as well.

After copying these four files from the `MP-3/sdcard` directory 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.

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.

5) 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.

6) 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.

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 /mnt/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 XPS and modify the base system.

7) Sentry System. So far we've 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, which is not at all inconsistent and buggy!

Making sure that XPS and SDK are currently closed, copy the project settings, system, constraints, and PS7 configuration files from your final MP-2 solution to your MP-3 system:

```
>> cp MP-2/system/system.xmp MP-3/system/system.xmp
>> cp MP-2/system/system.mhs MP-3/system/system.mhs
>> cp MP-2/system/data/system.ucf MP-3/system/data/system.ucf
>> cp MP-2/system/data/ps7_system_prj.xml
   MP-3/system/data/ps7_system_prj.xml
>> xps system/system.xmp &
```

If you aren't confident in your MP-2 solution, we can post these files on Blackboard. Either way, export your design to SDK and get back to finishing part 6), if you haven't already.

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 Xilinx 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. Unfortunately, the *camera_app* infrastructure only worked correctly (and even then, not that reliably) with an older version of the standalone OS. From SDK, import the system_bsp project from MP-2.
5. You'll need to point your new FSBL application to use the imported system_bsp. Right-click on the FSBL project and select "Change Referenced BSP" to point to system_bsp.
6. Congratulations – more compile errors! These can be fixed via the suggestions on the class wiki for MP-2.

Re-generate your `BOOT.BIN`, and load up Linux. Similar to your experiences in MP-2, you might need to reboot the board a few times for the camera initialization to work correctly. 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%).

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.