IOWA STATE UNIVERSITY

ELECTRICAL AND COMPUTER ENGINEERING

**CprE 488**

# Machine Problem 3

**March 25, 2024**

**Group :** S1-4

Member 1: Nathan Thoms

Member 2: Taylor Johnson

Member 3: Jack Cassidy

Member 4 : John Lavigne

1) *Briefly explain how the Make process was configured to appropriately use a cross-compiler targeting the ARM architecture.*

PetaLinux is a development environment and toolchain that includes cross-compilation capabilities. The process to compile & build the linux kernel for a given hardware platform include:

- <u>BSP Generation</u>
    - Specify the board, fabric configuration, interconnections, IP cores, constraints etc.. (petalinux-config --get-hw-description <location-of-xsa-file>)
- <u>Software Image Configuration & Generation</u>
    - Configure the linux kernel and drivers, this is done through the menuconfig gui. (petalinux-config -c kernel)
    - Build the kernel and other files required by the packager. (petalinux-build -c kernel, petalinux-build)
- <u>Inclusion of Hardware Images</u> - Take compiled Linux kernel and package it into an executable. Specify the .bit for fpga fabric configuration. Specify .elf for customization of FSBL. (petalinux-package --boot --force --fpga <path-to-your-bitfile>/<bitfile-name>.bit --fsbl --u-boot)

2) *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.*

1.) U-Boot 2020.01 (Mar 24 2024 - 17:26:24 +0000)

Version of u-boot used on specified date

Board information (lines 2-7):

2.) CPU: Zynq 7z020

Identifies the FPGA's CPU

3.) Silicon: v3.1

Identifies the chip

4.) Model: Zynq Zed Development Board

Identifies the zed board

5.) DRAM: ECC disabled 512 MiB

Disables the Dynamic Random Access Memory and specifies its size as 512 Megabytes

6.) Flash: 0 Bytes

No flash memory

7.) NAND: 0 MiB

No NAND memory

```
8.) MMC:   mmc@e0100000: 0
```

MMC is for a multi media card which has address location specified.

```
9.) Loading Environment from SPI Flash... SF: Detected s25fl256s1 with page size 256
    Bytes, erase size 64 KiB, total 32 MiB
```

An attempt to load an environment from SPI flash.

```
10.)    *** Warning - bad CRC, using default environment
```

CRC stands for cyclic redundancy check. It is a type of error check on data transmission. It indicates that there were changes made to RAW data and possible corruption of the environment, so u-boot uses default environment instead.

Identifying / setting up the serial port (lines 11-13), all three, in, out, and error, are set to the same location.

```
11.)In:    serial@e0001000
```

Address location for input to a UART serial device.

```
12.)Out:   serial@e0001000
```

Address location for output of a UART serial device.

```
13.)Err:   serial@e0001000
```

Address location for error of a UART serial device.

```
14.)    Net:
```

Beginning to configure the ethernet connection (lines 15-17):

```
15.)ZYNQ GEM: e000b000, mdio bus e000b000, phyaddr 0, interface rgmii-id
```

Gives the address for the Gigabit Ethernet MAC (Media Access Control), MDIO (Management Data Input/Output) bus address, and physical address.

```
16.)    Warning: ethernet@e000b000 using MAC address from DT
17.)    eth0: ethernet@e000b000
```

Lines 16 and 17 indicate that the ethernet controller at the memory address is provided by the device tree.

```
18.)    Hit any key to stop autoboot:  0
```

Allows the user to exit or stop the auto boot process by pressing any keyboard key, though the boot process continues and does not wait for user input.

```
19.)    switch to partitions #0, OK
20.)    mmc0 is current device
```

The MMC is for a multi media card is the current device.

```
21.)    Scanning mmc 0:1...
```

Searching the MMC partition specified.

```
22.)    Found U-Boot script /boot.scr
```

Located the instructions for booting the system in the boot script file.

```
23.)    123 bytes read in 19 ms (5.9 KiB/s)
```

The size of the file read and at what speed.

```
24.)    ## Executing script at 03000000
```

Gives the starting address of the boot script.

```
25.)    11395424 bytes read in 642 ms (16.9 MiB/s)
```

Indicates the size of the data that was read / executed and at what speed.

Configuring the kernel (lines 26-41):

```
26.)    ## Loading kernel from FIT Image at 15000000 ...
```

The kernel image is loaded from a flattened image tree (FIT).

```
27.)    Using 'conf@system-top.dtb' configuration
```

Uses the device tree blob to configure the kernel.

```
28.)    Verifying Hash Integrity ... OK
```

Checks that the kernel image is appropriate for use.

```
29.)    Trying 'kernel@1' kernel subimage
```

Attempting to load a kernel subimage (linux kernel image).

```
30.)     Description:  Linux kernel
```

This gives a description of the subimage being loaded.

```
31.)    Type:          Kernel Image
```

Identifies the type of subimage being loaded.

```
32.)    Compression:  uncompressed
```

Means that the subimage is not compressed.

```
33.)    Data Start:   0x150000e8
```

Gives the start address for the kernel image.

```
34.)    Data Size:    4181976 Bytes = 4 MiB
```

Specifies the size of the kernel image about 4 Megabytes.

```
35.)    Architecture: ARM
```

Indicates that the subimage / kernel image uses an ARM architecture.

```
36.)    OS:           Linux
```

Specifies the type of operating system the kernel image uses.

```
37.)    Load Address: 0x00200000
```

This is the address for the kernel image to load to.

```
38.)    Entry Point:  0x00200000
```

This specifies the address where the processor will start execution once the kernel image has been loaded.

```
39.)    Hash algo:    sha256
```

This specifies the hash algorithm used to ensure that no corruption of the kernel image occurred during data transfer / when the image was loaded.

```
40.)    Hash value:
          a0a89aaea5b04b4795d9011477c742b0cd9962481411314cb0990ac07ae9cce6
```

This is the hash value of the kernel image that was used to verify whether the kernel image was corrupted.

```
41.)    Verifying Hash Integrity ... sha256+ OK
```

This indicates that the kernel image was not corrupted.

Configures / set-up the Random Access Memory (RAM) (lines 42-55):

```
42.)    ## Loading ramdisk from FIT Image at 15000000 ...
```

The RAMdisk is loaded from a flattened image tree.

```
43.)    Using 'conf@system-top.dtb' configuration
```

Uses the device tree blob to configure the RAMdisk.

```
44.)    Verifying Hash Integrity ... OK
```

Checks that the RAMdisk image is appropriate for use.

```
45.)    Trying 'ramdisk@1' ramdisk subimage
```

Attempting to load a ramdisk subimage (petalinux ramdisk image).

```
46.)    Description:  petalinux-image-minimal
```

This gives a description of the subimage being loaded, that it was created from petalinux.

```
47.)    Type:         RAMDisk Image
```

This identifies the type of subimage being loaded for setting up the RAM.

```
48.)    Compression:  uncompressed
```

Indicates the start address for the RAMdisk image.

```
49.)    Data Start:   0x1540321c
```

Gives the start address for the RAMdisk image.

```
50.)    Data Size:    7186895 Bytes = 6.9 MiB
```

Specifies the size of the RAMdisk image about 6.9 Megabytes.

```
51.) Architecture: ARM
```

Indicates the subimage / RAMdisk image uses an ARM architecture.

```
52.)    OS:           Linux
```

Specifies the type of operating system the RAMdisk image uses.

```
53.)    Hash algo:    sha256
```

This specifies the hash algorithm used to ensure that no corruption of the RAMdisk image occurred during data transfer / when the image was loaded.

```
54.)    Hash value:
          e106a418c0c0c64da9d285a266b6ba443762cd7be8d6d036ddc7c9c953f0f286
```

This is the hash value of the RAMdisk image that was used to verify whether the RAMdisk image was corrupted.

```
55.) Verifying Hash Integrity ... sha256+ OK
```

This indicates that the RAMdisk image was not corrupted.

3) *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.*

Kernel Messages:

```
PetaLinux 2020.1 avnet-digilent-zedboard-2020_1 /dev/ttyPS0

avnet-digilent-zedboard-2020_1 login: usb 1-1: new low-speed USB device number 4 using
ci_hdrc
usb 1-1: New USB device found, idVendor=2123, idProduct=1010, bcdDevice= 0.01
usb 1-1: New USB device strings: Mfr=1, Product=2, SerialNumber=0
usb 1-1: Product: USB Missile Launcher
usb 1-1: Manufacturer: Syntek
hid-generic 0003:2123:1010.0003: device has no listeners, quitting
```

```
Disconnect
PetaLinux 2020.1 avnet-digilent-zedboard-2020_1 /dev/ttyPS0

avnet-digilent-zedboard-2020_1 login: usb 1-1: USB disconnect, device number 3
```

The kernel messages indicate the recognition that a new USB device has been connected. The USB device is then assigned a device number, given a major and minor number, and identified as a USB Missile launcher manufactured by Syntek. The hid-generic device driver attempts to communicate with the missile launcher, but is incompatible, which causes the device to become disconnected.

4) *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.*

The changes made to the usb-skeleton.c template included:

- Replacing functions and data structures prefixed with "skel" to "launcher" (etc. skel_write() to launcher_write()).
- We copy pasted #defines from launcher-commands.h to reference during control urb configurations.
- Updated constants such as vendor and product IDs to match that of the missile launcher product to be added to the module device table.
- Added entries into the usb_launcher structure for storing the last command and interrupt endpoint.

- Remove the registering of a bulk endpoint from within launcher_probe() and add an interrupt endpoint.
- Create an interrupt urb and submit it.
- Crease a callback function to be called upon completion of interrupt urb that checks boundaries and stops launcher if necessary, then resubmits the interrupt urb.
- Updated launcher_write to create a control command for the launcher and send it to the control endpoint through the usb.h api's usb_control_msg().

5) *Walk through launcher_fire.c. and describe how it generally works.*

In launcher_fire.c, first it defines all register values for the commands in launcher_commands.h that already exist. There are two functions in the driver code, a main function and a launcher_cmd function. The main function initializes all variables that are needed and then it opens the file device, runs launch_cmd with the command of fire, delay for a bit and then runs launch_cmd again but with of stop and lastly it closes the file device. In launcher_cmd it uses the write function to attempt to send the command to the file device, when it works it will give a delay which is for launching the missile.

6) *Describe the general algorithm you implemented to use the camera framebuffer and USB missile launcher driver to act as an automated sentry system in launcher_fire_camera.c.*

The algorithm loops through the camera framebuffer and creates an x, y coordinate for each pixel – a coordinate pair for each 4 byte read. It proceeds by extracting the YCbCr information and converting it to RGB. Once in RGB representation a threshold filter is applied to the individual color components and if the pixel passes the x and y values are added to the running sum of x and y values. After looping over the framebuffer we divide the x and y sums by the number of unfiltered pixels – giving us the mean x and y pixel coordinates. We set the center of the screen to be the (0, 0) coordinate and then translate the mean x and y to be relative to the center. We check whether the absolute x or y component of the mean vector is larger and move in that direction. This is repeated until the mean vector lies within a window setup around the base coordinate. In this case the launcher prepares to fire by moving the launcher slightly up to compensate for the camera's elevated reference relative to the launcher. FIRE – aliens are turned into goop.

Team Member Contribution

| | |
|---|---|
| Nathan Thoms | 35% |
| Taylor Johnson | 25% |
| Jack Cassidy | 25% |
| John Lavigne | 15% |