Iowa State University
Computer and Electrical Engineering Department

CPRE 488 S1-2

PROJECT DOCUMENTATION

Jun 25, 2023

Austin Beinder
Will Galles
Jon Pixler
Gregory Ling
Douglas Zuercher

Contents

1 Project Overview **4**
1.1 Summary 4

2 Hardware BOM **5**

3 Vivado BOM **6**

4 Overview **7**

5 Vivado **7**

6 Bluetooth PMODs **7**

7 Petalinux Build Setup **9**

8 ESP32 Gamepad Control **9**

9 Camera **10**
9.1 Goal 10
9.2 Initial Camera Testing 10
9.3 Opening the Petalinux Demo project with camera support 11
9.4 Final attempt 11
9.5 Running on the Pi 12

10 Raspberry Pi **13**

11 Roomba Control **13**

12 BNO055 IMUs **13**

13 Turret Control with IMU **14**

14 Shot tracking **15**

15 Lighthouse Deck Communication via Python **16**

16 CrazyFlie Drone Control **17**

17 Future notes **17**

18 Appendix A: Petalinux Notes **18**
18.1 Create Project 18
18.2 Configuration 18
18.3 Enable ARM NEON instructions in OpenCV 20
18.4 Enable the overlayfs in fstab - DID NOT WORK 21
18.5 Build 22
18.6 Create boot.bin 22
18.7 Partition SD Card 22
18.7.1 Option 1: Using fdisk with sudo and the physical SD card attached 22
18.7.2 Option 2: Use VDI 24
18.7.3 Option 3: Using the zedboard 25
18.8 Set up for booting 27
18.9 Updating the root partition 27
18.10 Boot! 28
18.11 OpenCV Configuration 28
18.12 Other 28

1 PROJECT OVERVIEW

1.1 Summary

Team S1-2 presents: “Fire At Will: A Roomba-Tank Shooter.”

In essence, Fire At Will is an FPS game played through real hardware. The base hardware consists of a CPR E 288 Roomba with the TIVA replaced by a Zybo Z7-10 with a turret mounted to the front. The player can wear a Meta Quest VR headset or use a normal display to view a live stream of the camera mounted to the top of the turret. Due to hardware limitations streaming the video through the Zybo, a Raspberry Pi was added which controls a CrazyRadio to communicate with a CrazyFlie drone with Lighthouse Deck, a second Lighthouse Deck on the top of the Roomba, and the camera on the turret. The Pi also allows for communication between the zybo and the browser through a WebSocket connection. An ESP32 PMOD is also used to communicate between the Zybo and a wireless Xbox game controller. Two IMUs, one mounted to the Roomba and one mounted to the top of the turret give the angle of the turret with respect to the Roomba for software shot tracking.

Shooting will be mimicked through software, that way actual darts are not being used. This will protect drones from damage, remove the four-shot limit of the turrets, and remove variation due to darts. The player will receive a point each time they shoot a drone. When shot, the targeted drone will land safely on the ground.

This game is helpfully described by Figure 1, below. This will also be in our marketing material.

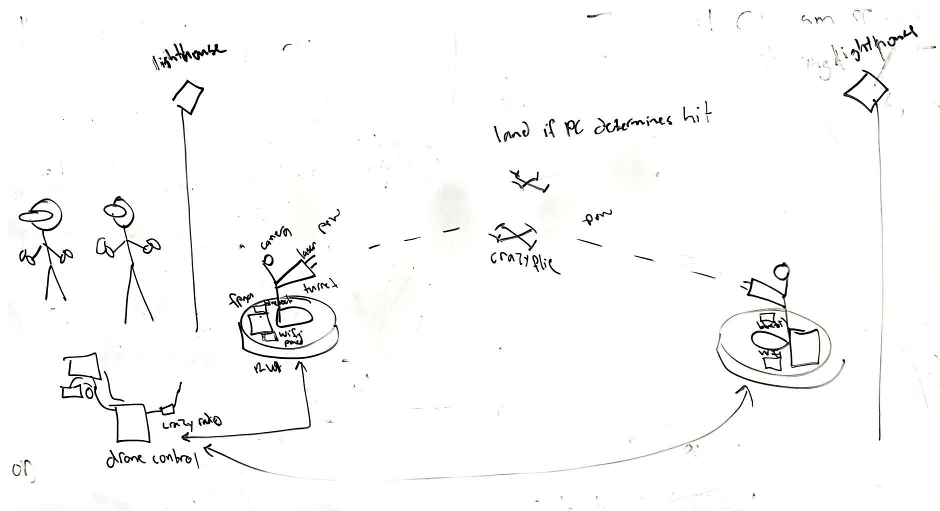

Figure 1: Visual depiction of Fire At Will: A Roomba-Tank Shooter

Figure 2: Final picture of Sparky

2 HARDWARE BOM

1 x CrazyFlie

2 x CrazyFlie Lighthouse Deck

1 x Raspberry Pi 4B

1 x OV5640 with long cable

2 x BNO055

1 x Roomba with baseboard

1 x Turret

1 x USB battery pack with USB-C and USB-A power outputs (Anker PowerCore III)

1 x CrazyFlie USB Radio

1 x Zybo Z7-10

2 x Bluetooth PMOD for wireless serial console to desktop

1 x Quest VR headset (optional)

1 x ESP32 PMOD

Assorted laser-cut acrylic mounts

3 Vivado BOM

1. Zynq Processing System
   (a) USB1 - Launcher
   (b) UART1 - Bluetooth Console
   (c) I2C0 - BNO055 IMU
   (d) I2C1 - OV5640 Camera Control (Moved to PI)
2. 4 AXI Uartlite IPs
   (a) UL1 - ESP32 Game Controller
   (b) UL2 - Cybot
   (c) UL3 - Raspberry PI - Web interface
   (d) UL5 - Raspberry PI - CrazyFlie data

4 OVERVIEW

Wiring diagram?

ZYBO -> CYBOT -> TURRET <-> PI <-> CAMERA <->

5 VIVADO

The final Vivado design for this project was fairly simple in comparison to the overall project. We added a lot of complexity in an attempt to make the camera function well, but ended up not using most of it.

In essence, three major components were used: AXI Uartlite IPs for every additional UART required, a PS I2C bus for the two IMUs to connect to, and the Zynq PS itself. Since we wanted a wireless console to Petalinux, we remapped the PS UART1 from the USB FTDI chip to a PMOD connector and wired it to a bluetooth PMOD.

Notes we learned in Vivado:

1. Don't edit the PS UARTs, Petalinux got confused when we tried to shift the console to PS UART0. I'm sure there's a way to configure it right, we just didn't find it easy enough and it was easier to remap UART1 to EMIO. Supposedly UARTLITE's can also be used as a console if the kernel and u-boot are configured correctly to do so.

2. We connected the IMUs to a PS I2C over the MIO PMOD on the same bus, tying one ADDR pin high and the other low to differentiate the two. You could use two I2C ports instead, but we had the other I2C in use attempting to make the camera work.

6 BLUETOOTH PMODS

The Bluetooth PMODs[1] were relatively easy to set up once we actually read the instructions[2].

[1] https://digilent.com/shop/pmod-bt2-bluetooth-interface/

[2] http://ww1.microchip.com/downloads/en/DeviceDoc/bluetooth_cr_UG-v1.0r.pdf

Pick two of them and start by resetting them both to factory default:

1. Power on with only JP1 shorted
2. Remove and replace JP1 three times and the LED should blink fast for a brief period
3. Remove JP1 and power cycle

Pick one to be the master and one to be the slave, it appears this is mostly arbitrary.

Short JP2 and JP3 on the master and only JP2 on the slave, then power both on. They should find each other and pair (red light will stop blinking when paired). Once paired, power them off and remove JP2, but leave JP3 on the master. Now they will automatically reconnect on power loss and will function equivalent to a uart wire at 115200 baud.

Notes we learned with the Bluetooth PMODs:

1. Using the jumpers is far easier than using the CMD interface.
2. Sending large quantities of data fast causes the red LED to blink very rapidly and communication will cease until a power cycle. Normal operation will not usually cause this. This was especially noticeable when GCC spewed hundreds of build errors from a single during development.
3. To tell if one end is communicating over UART, you can enter three dollar signs $$$ and you should get CMD as the response. You can then enter three dashes followed by return to go back to normal mode, you should see END be printed on the console. This communicates to your local PMOD only and does not test the bluetooth connection. Note that this also means that sending three dollar signs normally *will cause issues.*
4. On power-up, they appeared to connect within about three seconds. It takes longer if one PMOD still thinks it is connected as you have to wait for it to timeout and enter pairing mode.

7 Petalinux Build Setup

See Appendix A. Mostly, the build setup was very similar to any other petalinux project, there were just a lot of small configuration changes required to get it working.

8 ESP32 Gamepad Control

In the final state of the project, the roomba is controlled by an Xbox 1 game controller that communicates to an ESP32 PMOD on the Zybo. The ESP32 PMOD was relatively easy to setup. An example project implementing some basic Bluetooth functionality was first loaded, and then modifications were made to make the ESP32 connect to the team's controller.

Once connections between the ESP32 and Xbox 1 controller could be reliably made, the controller's message structure was broken down. In total, a message from the controller is 16 bytes long, with each byte containing the information shown below in Table 1.

Byte(s)	Control (Mask)
0 (LSB)	Share Button (0x01)
1	Menu (0x08), Display (x004), Xbox (0x10), RJoyPress (0x40), LJoyPress (0x20)
2	B (0x02), A (0x01), X (0x08), Y (0x10), RBumper (0x80), LBumper (0x40)
3	DPad Up (0x01), DPad Down (0x05), DPad Right (0x03), DPad Left (0x07)
5,4	RTrigger Position (0x0000 unpressed, 0xff03 pressed)
7,6	LTrigger Position (0x0000 unpressed, 0xff03 pressed)
9,8	RJoy Vertical Position (0x0000 up, 0xffff down)
11,10	RJoy Horizontal Position (0x0000 left, 0xffff right)
13,12	LJoy Vertical Position (0x0000 up, 0xffff down)
15,14	LJoy Horizontal Position (0x0000 left, 0xffff right)

Table 1: Xbox 1 Bluetooth Message Structure

The controller also sends occasional battery update events, which were conveniently already handled by the example code.

The Xbox 1 controller sends Bluetooth messages whenever an input changes. The ESP32 tracked the latest message, along with the last battery update, and sent a message over SPI to the Zybo every 100 ms. The SPI message also included a status flag indicating if the controller

was connected or not.

Notes we learned with the ESP32 gamepad controls:

1. The ESP32 PMODs do not support Bluetooth Classic, only BLE.
2. Most controllers (including all of the controllers in Coover) support Bluetooth Classic, not BLE.
3. Xbox 1 controllers with model number 1914 and 1797 support BLE. Model 1914 can be purchased at Walmart.

9 CAMERA

Oh dear, where even to begin?

9.1 Goal

Keeping our end goal in mind was hard because there were many different ways of making the camera work, but few where we could reasonably stream it out to a WiFi-enabled device. Our goal was to configure and control the camera in Petalinux, but have the data stream from the MIPI CSI-2 directly to a custom AXI-Stream SPI module to the ESP32 for broadcasting.

9.2 Initial Camera Testing

To start, we attempted to prove the camera would work for us by loading an example Vivado hardware and software project from Digilent. We struggled more that necessary getting the examples to work, in large because we did not realize many projects used Tcl scripts to setup the project.

Eventually, we settled on using the this project from Digilent. This example was particularly attractive as it also included code for configuring the PCAM on the fly, which we thought we

may use later. Because this example does not use a Tcl script to populate the Vivado project, we got the example up with little trouble.

We were very excited to *finally* have the camera running. If only we knew just how long it would take before we'd see it running, again.

9.3 Opening the Petalinux Demo project with camera support

This deserves an entire subsection on its own for all the trouble it caused. The Petalinux Demo project for Z7-20 has support for the PCAM and is mostly compatbile with the Z7-10, and the BSP was missing pieces, so we cloned the repo directly from `https://github.com/Digilent/Zybo-Z7`, tag `20/Petalinux/2020.1-1`. Ensure you update and initialize the submodules. Now, to open the hardware project is a little harder because the script to initialize the project is in a different repo at `https://github.com/Digilent/digilent-vivado-scripts`, `checkout.tcl`. It required a bit of minor tweaking, but mostly just editing paths and ensuring the vivado ip repo was actually checked out as a submodule in the example project. Some parts of the block design tcl file had to be manually adjusted to work with the Z7-10, but this project did build and appear to run. We did not pursue this further since it used a framebuffer in Petalinux, and for our purposes we just wanted to control the camera, but having the image data go through Petalinux was unnecessary and more complicated than we wished. We also ran into issues where removing the framebuffer didn't work because the Petalinux driver for the camera required both the camera and the MIPI/framebuffer to exist to start a stream. If the camera were to be streamed to the Zybo for image recognition and HDMI passthrough only such as the target acquisition project, this would be a suitable option. Also, we did not test this to see if the MIPI CSI-2 Rx Subsystem worked as expected, in every other project we tried to use the current MIPI CSI-2 Rx Subsystem IP block in, it failed to work correctly.

9.4 Final attempt

Our final attempt before transitioning to the Pi for video was to use the pre-production MIPI DPHY and CSI-2 blocks given in the petalinux example project (which worked), and configure the camera manually in Vitis. This worked to an extent. The MIPI DPHY worked great, but these blocks are marked as pre-production and evidently were stripped-down versions of

the real Digilent IPs for this camera made open-source specifically for that example project, so the CSI-2 block only supported RAW10. I manually edited the VHDL (LLP.vhd is the main culprit) to add support for the JPEG data type and make it raw 8 bits in/out instead of parsing the 10-bit RAW10 format. Note: the MIPI receives 32 bits per clock (4 bytes of JPEG data), but the AXI-Stream out of the block is 40 bits wide. I padded with zeroes in VHDL since we can't reconfigure half of the MIPI CSI-2 block, it is just too old and it worked. We were able to get JPEG data off of the camera and stream it over SPI, but at that point we compiled the code for the ESP32 and realized that it didn't have enough memory to hold a single compressed frame and gave up to use the Pi instead.

9.5 Running on the Pi

Making the camera run on the Pi was its own adventure, first step was to follow the Raspberry Pi guide for recompiling the kernel and build in support for the OV5640. Then I modifed the OV5647 DTB overlay replacing the name, I2C address, added an entry for xclk, and capitalized DOVDD, DVDD, and AVDD everywhere. After that and adding it to /boot/config.txt, the camera appeared. JPEG does not work by default because the Unicam driver does not support it, so I added support by listing MJPG and JPEG in the supported formats list (copying params from the OV5640 driver), and rebuilding the kernel module. We used ustreamer[3] to stream the video from the camera, but evidently the JPEG sent back from the camera was invalid. I manually hacked apart the hardware stream encoder in ustreamer to make it skip the invalid bytes in the JPEG stream when it forwards the stream to the browser. In the future, I would be curious where those bytes are being added and if they can be removed so the stream is continuous and correct automatically.

Notes we learned about the camera:

1. Use the old DPHY and CSI-2 blocks. Replacing them for the new MIPI CSI-2 Rx Subsystem block breaks everything every time we tried.
2. Don't be afraid to edit the CSI-2 block for different data formats, LLP.vhd is fairly easy to read and likely where you need to make edits.
3. For non-wireless streaming on-Zybo processing would work fine. Our main limitation was getting the data in the air fast enough.

[3] `https://github.com/pikvm/ustreamer`

10 Raspberry Pi

Setting up the Raspberry Pi itself was rather simple except for the camera interface. Python/Pip just worked with the CrazyFlie equipment. I personally prefer NetworkManager, so I used raspi-config to change the default nework manager and used nmcli/nmtui to connect to IASTATE. Note for netreg'ing a Pi, the Lynx browser (`apt install lynx`) is a great terminal-based browser for headless systems that is capable of logging in to netreg the Pi. `lynx http://netreg.iastate.edu` usually gets you in.

11 Roomba Control

The Roomba is controlled by the Zybo using the iRobot Open Interface spec. The software side of this is simple: every loop of the main program, the Zybo reads the most recent message from the ESP32 PMOD (if a new packet exists), converts the left joy stick data to "tank styled" drive information, and sends data over UART to the roomba.

Importantly, the ETG warned us that the Roomba interface pins are unreliable and prone to voltage spikes that can damage the Zybo. As a result, we did not directly connect the Zybo to the Roomba connector. Instead, we connected the Zybo to the ETG-provided baseboard, which has additional over-voltage protection circuitry on it.

12 BNO055 IMUs

I2C communication from linux is very similar to UART[4]. To connect to an I2C device, use `i2cdetect -y -r #` where # is the i2c number listed in `/dev` to ensure the I2C device is accessible and verify the correct bus. Once found, use the `open` syscall to open a connection to the bus, then `ioctl` to identify the I2C slave address the file descriptor should communicate with. In our case, we had two file descriptors open to the i2c bus, each with the slave address set to the corresponding IMU's address.

[4] `https://www.kernel.org/doc/html/latest/i2c/dev-interface.html`

```
// Open the I2C bus.
int fd = open("/dev/i2c-#", O_RDWR);

// Set the slave address for this file descriptor. All future reads and
// writes to this fd will communicate with this slave address.
ioctl(fd, I2C_SLAVE, addr);
```

To read a register on the BNO055, use the `write` syscall to write the address, then the `read` syscall to read back the data. The BNO055 supports continuous reads, so to read consecutive registers, you can call `read` with more than one byte of data.

Writing is similar, write the address followed by the data. This can be done in one write syscall or across several if that is easier.

Using this interface, we adapted Gregory's old BNO055 IMU code from 288 to communicate with the Cybots.

Notes we learned about the BNO055 IMUs:

1. It was unclear whether or not calibration actually helped.
2. We ended up using the IMU mode instead of the NDOF mode Gregory used with the Roombas because it appeared to skew less as the roomba made quick turns.
3. Math is hard working with angles from different sources.

13 Turret Control with IMU

The turret controls were a hybrid system utilizing the game controller along with the IMU attached to the turret. The system first gathers a desired angle from the user and then uses the real world angle to calculate the needed turret movement.

The controller utilized the right analog stick to produce the desired absolute angle. This angle was calculated by scaling the stick output that ranged from +- 500 units in both the up/down and left/right orientations. This value was then scaled to the max angles of the turret with

the yaw angle being limited to +- 135 degrees and the pitch angle being limited to -5 to +30 degrees.

Once the desired angle is determined the system then uses the turret and cybot IMUs to calculate the current angle that the turret is facing. It does this by utilizing the IMU mode of both of the IMUs. In this mode they both output their relative angles compares to when they were first powered on. The turrent angle was calculated by finding the difference between the two IMUs. In this case it was assumed that both IMUs would be aligned on startup. Finally this angle would then represent the real world angle of our turret.

With both our desired angle and real world angle for the turret we could calculate the needed turret movement to get us from the real world angle to the desired angle. This was done with a simple subtraction of the two angles and then translating the result into up/down and left/right movement commands. Finally those commands were sent to the turret USB driver to be executed.

Finally this entire process repeats each time the main processing loop repeats.

14 SHOT TRACKING

The shot tracking system was a integration of many tools to create a system that was capable to tracking multiple objects independently moving through the arena and was capable of determining if the turret was aimed at a drone. It was comprised of the lighthouse boards on both the crazyflie and on the cybot along with the cybot and turret IMUs.

The first step in the process was finding relative the turret angle. This is the same process that is used to finding the real world angle of the turrent control. We calculate the difference in angle between the cybot and turret IMUs to get our relative turret angle.

The next step in the process was using the cybot lighthouse deck to get the absolute yaw of the cybot. This was a value that was provided by the lighthouse deck directly and there was not a need to do any calculations on the zybo side. From there we could add the relative turret angle to the absolute cybot yaw to get our absolute turret yaw.

The next step in the process was to get the XYZ locations of both the crazyflie and cybot using their light house decks. This information is obtained through the Pi that is running the

lighthouse software.

Finally with all of our location and angle date captured we can then preform the calculations. The first calculation was for the pitch plane to determine if the turret was aimed at the drone in that plane. The first step was to get the XY distance from the cybot and the drone. This was done using the basic distance formula. From there that distance along with the difference of Zs could be used in an atan calculation. This would then provide and angle output that we could then compare to the pitch angle of the turret. If they match or are within our margin of error it is determined as a hit in the pitch plane.

15 LIGHTHOUSE DECK COMMUNICATION VIA PYTHON

The lighthouse deck was connected to the raspberry pi which allowed us to use python to communicate with it. The lighthouse deck is developed by bitcraze and has many open source resources to help work with it. In the pi folder in the project zip file there are lighthouse python scripts that allow us to either communicate with the lighthouse deck and determine position, fly the drone, or to test the lighthouse deck. The files in the folder are basic hover demo matlab.py, lighthousebootup.py, lighthouseCommTest.py, and lighthouseDeck-DronePathCombined.py and lighthouseDeckDronePathCombined2.py. The primary file to use will be lighthouseDeckDronePathCombined2.py and is meant to be run immediately after the drive controller executable is run.

The regular uart pins on the ligthouse deck were used in order to communicate with the deck. I believe we used uart1 on the deck. Please note that while you can use the external pads to run the bootloader script if you want, the firmware will only work with the pins regularly connected to the crazyflie.

The protocol used to communicate with the lighthouse deck over uart is to first send it into a break condition, then send a boot command which will run the bootloader, after which point it should send back sweep data and sync frames. The sweep data has to go through several layers of interpretation before being converted to angles of azimuth and elevation to each base station that is visible. When two base stations are visible by a sensor, and we know the base station location, we can draw lines and use the intersection point to find the location of the lighthouse deck. Since there are 4 sensors, we can use the angles of these sensors relative to each other and the x and y axes in order to determine yaw.

The lighthouse was connected to the /dev/ttyAMA3 uart port. The geometry used to find position relative to the base stations comes from when the lighthouse deck on the drone is calibrated. See the Crazyflie Drone Control section in order to learn how to calibrate the lighthouse deck on the drone. But when the calibration is done, it should say the xyz of each base station. These positions were hard coded on line 863 of lighthouseDeckDronePathCombined2.py and whenever we updated the calibration of the drone we would also update this line, which ensured the cybot and drone had the same sense of position relative to each other.

Data on the lighthouse position and drone position are then passed over uart /dev/ttyAMA4 to the FPGA which does more processing with the data.

16 CrazyFlie Drone Control

Crazyflies are used a lot in CPRE 488 and MicroCart Senior Design projects. The crazyflie is controlled from the same python script that controls the lighthouse deck. The script requires the use of the crazyradio to be plugged into a usb port on the pi, which allows communication with the drone. The drone then also needs to be calibrated to learn the geometry of the base stations. This geometry should then be manually copied to the geometry of the ligthouse deck code, see Lighthouse Deck section. These two youtube videos describe how to use the lighthouse deck system:

`https://www.youtube.com/watch?v=DCEHht72B08&t=2s&ab_channel=Bitcraze`
`https://www.youtube.com/watch?v=OHiReFUJXJM&ab_channel=MicroCART`

The actual drone control logic is pretty simple. It just has the drone take off, and optionally fly in a rectangle, or stay still. When the fpga determines a hit has occurred, the drone will land, wait 8 seconds, and then take off again. The drone has position and rotation logging enabled so that it is sending its position to the fpga every 10ms or as quickly as it can receive position data.

17 Future notes

Some things we'd like to have tried differently:

1. Connect a usb hub to the zybo and perform all computation there instead of offloading communication with the Lighthouse Deck and CrazyFlie to the PI. It should be possible if python were built into petalinux. One major issue that will need to be addressed is using pip to install the python libraries required as that requires an active internet connection. Two alternatives would be downloading what pip requires manually, transferring it to the Zybo, and telling pip to use the pre-downloaded wheels or reconstructing the interface to use C.

2. The major limiting factor with the ESP32 streaming video data was RAM usage. If enough compression is used and the frame were chunked as it is streamed so only a portion of a frame is in-memory on the ESP32 at a time, that may still work. Chunking the video data was just too much hassle for the last few days before demo to work through. If that and the item above did work, we could theoretically remove the Pi completely which was our original goal.

18 Appendix A: Petalinux Notes

These are more rough notes from during the Petalinux build process:

18.1 Create Project

```
petalinux-create -t project -n faw --template zynq
cd faw
(copy .bit and .xsa into parent directory)
```

18.2 Configuration

```
petalinux-config --get-hw-description ..
  - DTG
    - Machine Name - MUST BE template
  - Image Packaging Configuration
    - Root Filesystem Type = EXT4
```

```
    - Device Node = /dev/mmcblk0p2
    - No copy images to tftpboot (don't need it, saves errors later)
    - rootfs type only tar.gz
  - Auto hardware
    - Select uartlite for main console
  - Yocto Build Settings
    - Machine Name - MUST BE zynq-generic
    - TMPDIR location /tmp/faw1/
    - Workspace location /tmp/faw1/workspace - much faster checkout
      during menuconfig with devtool, must run
      petalinux-build -c kernel -x finish
      when done configuring.

petalinux-config -c kernel
  - Device Drivers
    - Character Devices
      - Serial Drivers
        - Xilinx uartlite serial port support + allow console
- Cadence PS uart + console
    - USB Support
      - USB announce new devices (optional)
    - Multimedia Support
      - I2C Encoders...
        - OV5640 support
      - V4L Platform Devices
        - Xilinx CSI2Rx subsystem
  - Kernal Hacking
    - Enable magic SysRq Key, disable over serial (optional)

petalinux-config -c u-boot
  - Device Drivers
    - Serial Drivers
      - Xilinx uartlite support

petalinux-config -c rootfs
```

```
  - Image Features
    - auto-login
  - Filesystem Packages
    - libs/opencv, libs/opencv-dev -- do we need both?
    - misc/packagegroup-core-buildessential -- GCC, etc.
    - libs/ffmpeg, libs/ffmpeg-dev
    - console/utils/screen
```

For native usb support, set bsp/device-tree/files/system-user.dtsi to:

```
/include/ "system-conf.dtsi"
/ {
  usb_phy0: usb_phy@0 {
compatible = "ulpi-phy";
#phy-cells = <0>;
reg = <0xe0002000 0x1000>;
view-port = <0x0170>;
drv-vbus;
};
};

&usb0 {
dr_mode = "host";
usb-phy = <&usb_phy0>;
};
```

18.3 Enable ARM NEON instructions in OpenCV

```
https://support.xilinx.com/s/question/0D52E00006iHiPVSA0/opencv-neon-optimization-
on-petalinux-20191?language=en_US

mkdir -p project-spec/meta-user/recipes-support/opencv
vim project-spec/meta-user/recipes-support/opencv/opencv_%.bbappend
```

```
# opencv_%.bbappend content:
EXTRA_OECMAKE_append_zynq = " -DENABLE_NEON=ON -DENABLE_VFPV3=ON"
```

18.4 Enable the overlayfs in fstab - DID NOT WORK

If this is attempted again, use AUFS, very similar overlay structure, but slightly different fstab options.

```
mkdir -p project-spec/meta-user/recipes-core/base-files/base-files
vim project-spec/meta-user/recipes-core/base-files/base-files_%.bbappend
‘‘‘
FILESEXTRAPATHS_prepend := "${THISDIR}/base-files:"

SRC_URI_append = " \
file://fstab \
"

do_install_append() {
install -d ${D}/mnt/overlay
install -d ${D}${sysconfdir}/
install -m 0644 ${WORKDIR}/fstab ${D}${sysconfdir}/
}
‘‘‘
vim project-spec/meta-user/recipes-core/base-files/base-files/fstab
‘‘‘
/dev/root    /              auto    defaults                   1  1
proc         /proc          proc    defaults                   0  0
devpts       /dev/pts       devpts  mode=0620,gid=5            0  0
tmpfs        /run           tmpfs   mode=0755,nodev,nosuid,strictatime 0  0
tmpfs        /var/volatile  tmpfs   defaults                   0  0

/dev/mmcblk0p3 /mnt/overlay ext4    defaults                   0  0
none         /              aufs    br=/mnt/overlay=rw:/=ro    0  0
```

‘ ‘ ‘

This will cause the root fs to be mounted at / read only, but the third partition will be mounted *on top of* the root fs. Any writes to the root filesystem will be stored in the overlay partition (p3), so if you need to update the rootfs, you can wipe and replace all the contents of p2 and your changes will persist. This is a very common setup to allow you to persist configuration changes when the entire root filesystem needs to be periodically updated from scratch. Now it’s safe to edit code on the device, it will be saved to the overlay partition on the sd card. You can also save it to the mmcblk0p1 partition which is FAT and readable by Windows, or on a usb stick connected via the missile launcher usb adapter.

18.5 Build

```
petalinux-build
```

18.6 Create boot.bin

```
petalinux-package --boot --force --fpga ~/Desktop/faw/system_wrapper.bit \
  --fsbl --u-boot
cp images/linux/BOOT.BIN images/linux/boot.scr images/linux/system.dtb \
  images/linux/uImage images/linux/rootfs.tar.gz ~/Desktop/faw
```

18.7 Partition SD Card

18.7.1 Option 1: Using fdisk with sudo and the physical SD card attached

```
ls /dev
```

- Insert the SD card

```
ls /dev
```

- Find what appeared when the SD card was inserted. There should be at least one entry that starts with ‘sd’ or ‘mmcblk’. Save the shortest of the names. Replace sdx with the name of this device in all future commands. Ex:

```
/dev/sda => The full raw SD card exposed as a file you can read/write to
/dev/sda1 => The raw first partition
/dev/sda2 => The raw second partition
...
```

Make sure the device you selected really is the sd card. You don’t want to run this command on your hard drive.

To edit the partition table, run fdisk on the full device:

```
sudo fdisk /dev/sdx
‘p‘ => Print the partition table, make sure it looks reasonable for the sd
       card. You don’t want to repartition your hard drive.
‘o‘ => Create a new empty DOS partition table (NOT GPT)
-- Create the boot partition (FAT32, 256MB)
‘n‘ => Create a new partition (primary, 1, default first sector,
       last sector +256M). It will say type is Linux,
‘t‘ => Change the partition type code to W95 FAT32 (Hex code = ‘b‘)
-- Create the root partition (EXT4, 2GB)
‘n‘ => (primary, 2, default, +1G)
-- Create the overlay partition (EXT4, rest of device)
‘p‘ => Should look similar to this (Device Boot names will vary)
Disk /dev/sda: 4294 MB, 4294967296 bytes, 8388608 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk label type: dos
Disk identifier: 0x09c750bc

  Device Boot      Start         End      Blocks   Id  System
/dev/sda1            2048      526335      262144    b  W95 FAT32
```

```
/dev/sda2          526336     4720639     2097152   83  Linux
/dev/sda3         4720640     8388607     1833984   83  Linux
‘w‘ => Write partition table to disk.
```

This has updated the partition table, but the filesystems have not been initialized yet. Running ‘ls /dev/‘ should now show three partitions, ‘/dev/sdx1‘, ‘/dev/sdx2‘, and ‘/dev/sdx3‘. The first needs to be formatted fat (make sure you change the ‘x‘)

```
sudo mkfs.vfat /dev/sdx1
sudo mkfs.ext4 /dev/sdx2
sudo mkfs.ext4 /dev/sdx3
```

Now, these partitions are ready to be used.

Mount sdx1 and sdx2:

```
mkdir root boot
sudo mount /dev/sdx1 boot
sudo mount /dev/sdx2 root
```

Copy BOOT.BIN, boot.scr, uImage, and system.dtb *only* into boot Untar rootfs:

```
cd root && sudo tar xzf /path/to/rootfs.tar.gz‘
```

If you need to update the rootfs, the easiest method is probably run ‘mkfs.ext4 /dev/sdx2‘ to wipe the partition and untar it again.

18.7.2 Option 2: Use VDI

Nope. Every route would require root access at some point in the process. You need a linux computer with root to make the partition table and edit it correctly.

18.7.3 Option 3: Using the zedboard

Use the sw-btn-led petalinux project from mp3 with the following packages added:

```
petalinux-config -c rootfs
 Filesystem Packages > base > e2fsprogs > e2fsprogs, resize2fs, mkfs
```

– Alternatively, download the pre-built minimal boot files 'minimal_boot.zip' and use those to boot the zedboard.

Login to the zedboard as root:root, remove the sd card you booted from, and insert the one you wish to format. If you need to reboot at any point, you will need to reinsert the original sd card. Because you are running from a ram fs, you do not need to keep the boot sd card inserted.

With the sd card to be formatted inserted into the zedboard's sd card slot, run fdisk to edit the partition table:

```
fdisk /dev/mmcblk0
'p' => Print the current partition table
'o' => Create a new empty DOS partition table (NOT GPT)
-- Create the boot partition (FAT32, 256MB)
'n' => Create a new partition (primary, 1, default first sector,
       last sector +256M). It will say type is Linux,
't' => Change the partition type code to W95 FAT32 (Hex code = 'b')
-- Create the root partition (EXT4, 2GB)
'n' => (primary, 2, default, +2G (if you have a 16GB card, you can
       up this to 4G if you want))
-- Create the overlay partition (EXT4, 3, default, rest of device)
'p' => Should look similar to this:
'''
Command (m for help): p
Disk /dev/mmcblk0: 15 GB, 15931539456 bytes, 31116288 sectors
486192 cylinders, 4 heads, 16 sectors/track
Units: sectors of 1 * 512 = 512 bytes
```

```
Device        Boot ... StartLBA    EndLBA    Sectors  Size Id Type
/dev/mmcblk0p1              16    524303     524288  256M  b Win95 FAT32
/dev/mmcblk0p2          524304   6815759    6291456 3072M 83 Linux
/dev/mmcblk0p3         6815760  31116287   24300528 11.5G 83 Linux
‘‘‘
‘w‘ => Write partition table to disk.
```

Now remove and re-insert the sd card to reload the partition table in kernel.

ls /dev/mmc* should show three partitions and the root device node.

Initialize filesystems on each of these partitions:

```
mkfs.vfat /dev/mmcblk0p1
mkfs.ext4 /dev/mmcblk0p2
mkfs.ext4 /dev/mmcblk0p3
```

Get a FAT formatted usb stick and put the BOOT.BIN, boot.scr, uImage, system.dtb, and rootfs.tar.gz on it from the images folder produced by your petalinux build earlier. Insert the usb stick in the missile launcher’s usb adapter into the zedboard.

Mount the usb stick:

```
mkdir -p /mnt/usb && mount /dev/sda1 /mnt/usb
```

Copy BOOT.BIN, boot.scr, uImage, and system.dtb *only* into boot (/mnt/sd-mmcblk0p1)

```
cp /mnt/usb/BOOT.BIN /mnt/usb/boot.scr /mnt/usb/uImage \
/mnt/usb/system.dtb /mnt/sd-mmcblk0p1
```

Untar rootfs into root (/mnt/sd-mmcblk0p2): It might take a while

```
cd /mnt/sd-mmcblk0p2 && tar xzf /mnt/usb/rootfs.tar.gz
```

If you need to update the rootfs, the easiest method is probably run ‘mkfs.ext4 /dev/mmcblk0p2‘ to wipe the filesystem and untar the new rootfs.

18.8 Set up for booting

This works on any computer, as FAT filesystems are very compatible. Copy the zImage, system.dtb, u-boot.elf, and BOOT.BIN files into the boot partition (FAT32, partition 1).

If on linux, mount the ROOT filesystem, cd to it, then run 'tar xzf /path/to/rootfs.tar.gz'. Eject the sd card and you're good to boot on the zedboard.

If not on linux, copy the rootfs.tar.gz file into the boot partition as well. If the filesystem is too large to fit, run through Option 1 again until it's big enough to fit. Eject the sd card and insert it back into the zedboard. Use the zedboard to mount the root filesystem and untar the filesystem into it. Note: A usb stick works great for this too, you can connect a usb stick using the usb adapter to the missile launcher to hold the rootfs instead of putting it in the boot partition, then you won't have as many space errors.

```
cd /tmp
mkdir root boot
mount /dev/mmcblk0p1 boot
mount /dev/mmcblk0p2 root
cd root
tar xzf /tmp/boot/...rootfs.tar.gz
```

If you run out of space at any point, perform Option 3 again, allocating more space where needed and try again.

18.9 Updating the root partition

If you need to update the root partition, follow the same steps as above for set up for booting, but make sure to delete everything in the root partition (or mkfs.ext4 on the partition again to clear it out) before extracting the new rootfs onto it.

18.10 Boot!

Now you can build against libopencv. We used gcc, screen, and vim on-device to compile for opencv, but you should be able to make an application through petalinux as well.

18.11 OpenCV Configuration

```
petalinux-config --get-hw-configuration /path/to/xsa
 -> Image Packaging Configuration -> Root Filesystem Type -> EXT4
  -> Device Node = /dev/mmcblk0p2
  -> No copy images to tftpboot (don't need it, saves errors later)
```

18.12 Other

```
https://developer.technexion.com/docs/using-gpio-from-a-linux-shell
to access MIO7:
cd /sys/class/gpio
gpiochipXXX gives you the base address, 906 + 7 = 913
cd /sys/class/gpio
echo 960 > export
cd gpio960
echo out > direction
echo 0 > value
echo 1 > value
https://support.xilinx.com/s/question/0D52E00006iHi0kSAK/
 how-to-tell-gpio-emio-pin-number-in-software?language=en_US
https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842398/
 Linux+GPIO+Driver#LinuxGPIODriver-UsingGPIOwithSysFs
emio gpios are based at 54 on the zynq = gpio[0]
i2c:
cd /sys/bus/i2c/devices/i2c-1
echo test 0x78 > new_device
cd ..
```

```
cd 1-0078

#include <fcntl.h>
#include <unistd.h>
#include <linux/i2c-dev.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>

int main() {
        int fd = open("/dev/i2c-1", O_RDWR);
        if (fd < 0) { perror("Open"); exit(1); }
        if (ioctl(fd, I2C_SLAVE, 0x3c) < 0) { perror("Open"); exit(1); }

        uint16_t val = 0x0A30;
        uint8_t res = 0;
        write(fd, &val, 2);
        read(fd, &res, 1);
        printf("%x\n", res);
        close(fd);
}
 cd /mnt && umount sd-mmcblk0p2 && mkfs.ext4 /dev/mmcblk0p2 && \
   mount /dev/mmcblk0p2 sd-mmcblk0p2; mkdir -p usb && \
   mount /dev/sda1 usb; cd sd-mmcblk0p2 && \
   tar xzf /mnt/usb/rootfs.tar.gz && cd .. && \
   umount sd-mmcblk0p2

diff -u helloworld.c newhelloworld.c > helloworld.patch

export PATH=$PATH:/tmp/gling/faw/build/tmp/work/x86_64-linux/ \
qemu-xilinx-helper-native/1.0-r1/recipe-sysroot-native/usr/bin/

width=640 && height=480 && rate=15
media-ctl -d /dev/media0 -V '"ov5640 0-003c":0 \
```

```
  [fmt:UYVY/'"$width"x"$height"'@1/'"$rate"' field:none]'
media-ctl -d /dev/media0 -V '"43c60000.mipi_csi2_rx_subsystem":1 \
  [fmt:UYVY/'"$width"x"$height"' field:none]'
v4l2-ctl -d /dev/video0 \
  --set-fmt-video=width="$width",height="$height",pixelformat='YUYV'
yavta -c14 -f YUYV -s "$width"x"$height" -F /dev/video0
```