# Lane Following and Sign Reading Robot

Nathan Kelley
Andrew Dejonge

**Abstract:**

This project seeks to research and explore embedded petalinux utilizing ROS in the programmable software of a hybrid system dev board. The devboard used is the ZYBO Z720 with a ZYNQ processor and an FPGA. The FPGA is utilized to increase the base resolution of the camera to 720p60 and is used for the image processor. The Zynq is utilized for the PID control. These combination of PS and PL create a hybrid system.

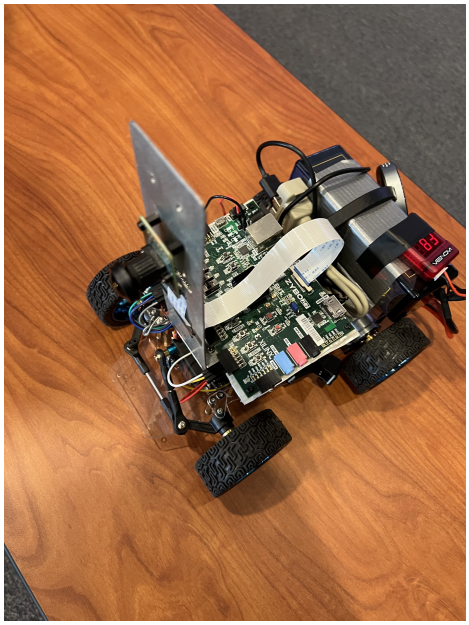
**This system seeks to allow a car to navigate a small course and stay in between two white lines.** This system uses a video feed, calculates the centroid of the each line of the lane, then averages the centroids to find the center of the lane. It then finds the centroid error between the center of the screen and the centroid of the lane, and uses PID control to attempt to correct the error. This system also uses wireless HDMI to output a video feed of what the autonomous vehicle is seeing at the current moment.

This system successfully met the goal of staying in between both white lane lines over a small course and was consistent in doing it in the controlled lighting conditions as color thresholding was utilized. The system had a response time of 41.6 ms to new stimuli and therefore is roughly 5 times faster at responding than a tesla is from detection to correction (2019 - 200ms). This system was also able to run at 1.7 miles per hour along the track.

**Video Demo Available Here : https://www.youtube.com/shorts/f_cf0l6KRUU**

**Table of Contents:**

**Physical System:**



The Osoyoo Servo Steering Robot Smart Car Was used as a chassis for this build. It has a max speed (with our weight load) of 3.8 miles per hour. We ran it at 1.7mph.

The chassis was fitted with a custom L-shape sheet metal frame to elevate the Digilent Pcam 5C.

A wood block was attached to provide a mounting location and sufficient clearance for the 3-cell Li-po battery. A space on the top of the vehicle was cleared for mounting the Zybo board.

The power for the car and its subsystems is supplied by a POVWAY 3S Lipo Battery (11.1V, 50C).A XL4015 DC/DC buck converter is used to supply 6V to the MG996R servo motor.

An MP2307 DC/DC buck converter is used to provide 5V to the Zybo Board and H-Bridge, and used as a high voltage reference for the level shifting circuit. A series of 5v fuses and a voltage monitor was used for safety to make sure we didn't overvolt our system, and to make us aware when the Lithium battery was getting low, to prevent…explosions with the battery.

The Digilent Pcam 5C was the camera of choice that received the video feed. It uses a MIPI connector plugged into the board.

Level shifters were employed to provide a logic level step-up from the 3.3V level of the Zybo to 5V of the servo and drive motors.

This car model utilizes an Ackermann steering column to turn, which is hooked up to a servo motor. MG996R is the servo model number.

JGA25-310 12V DC motors were used as the drive motors. They were supplied power from the li battery.The Osoyoo Model-X Motor Driver Module was used to drive the motors.

A Zybo Z7-20 development board houses a microcontroller as well as FPGA. FPGA stores our hardware in this system, and the Zynq Processing System microcontroller houses our ROS.

A Diagie Wireless HDMI Transmitter was used to wirelessly stream the video output.

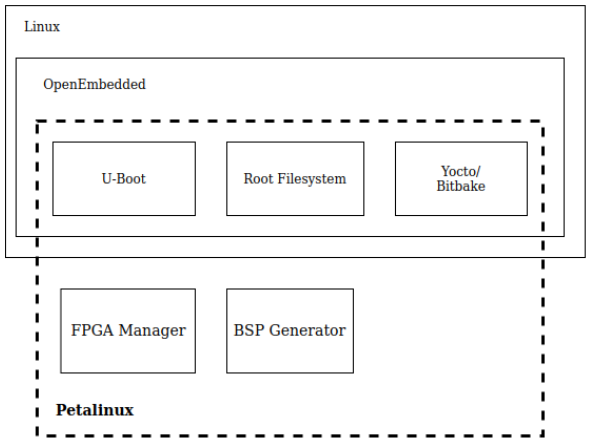
**ROS Node:**

The ROS code for this project was kept fairly simple, as much of the focus of this project was getting the ROS node to compile and run in embedded linux. Because of this, only basic packages were imported to reduce the amount of dependencies needed, and the callback function was kept short so that it can be executed at the lower frequencies of an ARM-based microcontroller. The main callback function was based on a 60Hz timer to coincide with the rate of calculations from the PL. The reason for a timer-based callback function instead of a subscriber based callback is due to limitations of communication between PS and PL requiring the use of shared memory to exchange information between the two.

Although this means that some of the benefits of using ROS are lost, namely the ability to monitor ROS topics for debugging and system overview purposes, there are still plenty of reasons why ROS is a good fit for a project like this. The main intent behind utilizing ROS instead of simply running baremetal C++ code is the simulation functionalities. There wasn't time to implement more advanced control, but the next step for this project would be to simulate the response from the PL to the ROS node with a "stimulation" node, as well as simulating the environment for the robot using Gazebo. This solves one of the big problems with development for this robot design, where any changes needed to be uploaded to the development board, followed by taking the robot to the physical test track and observing how it behaves. In order to expand upon the simple control that was implemented, it would be greatly beneficial to be able to simulate the robot traversing its environment instead of the lengthy testing process.

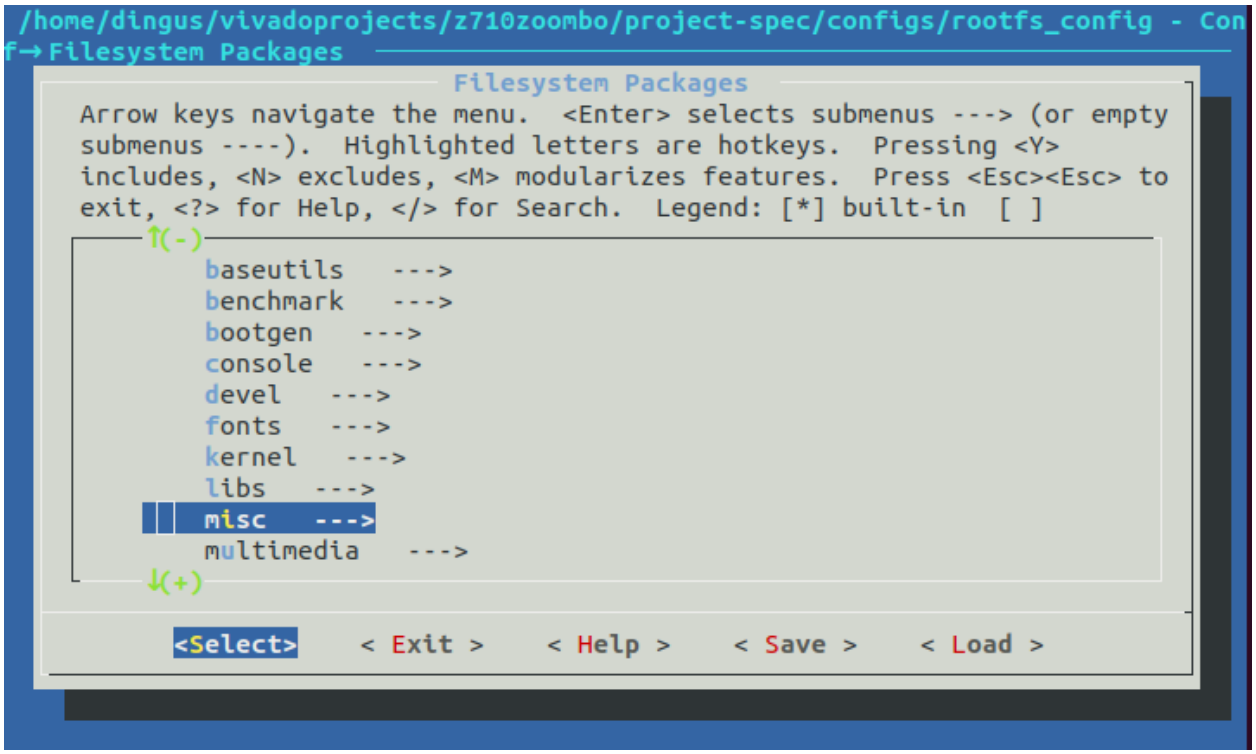
**Petalinux:**

Petalinux is a set of tools for compiling customized build of embedded linux specifically for the heterogeneous processors created by Xilinx, and is based upon the set of OpenEmbedded tools shown in the diagram below. This allows for creation of a boot media (in our case, an SD card) that contains both a customized embedded linux image as well as a hardware design specification for the FPGA on the processor. In addition to the FPGA specific functionalities of Petalinux, the tool also attempts to streamline the process of building by automating OpenEmbedded build commands and providing a GUI for configuration of the build.




Petalinux toolset hierarchy

The OpenEmbedded tools that Petalinux is based upon are Yocto, BitBake, and U-Boot. These are standard OpenEmbedded tools that handle the compilation of code for adding to embedded linux, as well as providing a base set of “layers” to the system that provide functionalities like git, python modules, and compilers for C/C++. Typically, Yocto and BitBake are entirely command line tools, and individual “recipes” (the packages that provide the functionalities for a layer) are compiled one at a time to create the linux image. With Petalinux, recipes are chosen from a GUI, then all compiled with one command. In practice, this doesn’t improve the build process very much, since the individual layers and recipes are placed into a sprawling directory of menus and submenus, with layers that offer similar functionalities placed into completely different locations. For example, the python and python3 layers are separated, with one inside of the “devel” menu and the other inside of the “misc” menu. A different example is the git layer, which can be found in the “utils” submenu of the “console” menu, instead of the “net”, “network”, or “utils” menus. Both of these problems wouldn’t be too much of an issue if there were any documentation of provided layers or some type of search functionality.




Subset of the top level layers for Petalinux

The last of the OpenEmbedded tools that make up the base of Petalinux functionality is U-Boot, which configures the startup sequence for the embedded linux install. The only difference in functionality between Petalinux’s use of U-Boot and a typical OpenEmbedded use of U-Boot is with the hardware description, or Board Support Package (BSP) that is provided to the tool. The BSP provided by Petalinux simply adds the hardware description for the FPGA design in addition to the description of the processor used (which can be either Zynq-7000, Zynq-Ultrascale, Microblaze, or Versal).

**Building with Petalinux**

Default configurations of Petalinux are simple to build, since there are only a few steps in the configuration process, however customizing the build by adding layers or increasing complexity require more steps and changes to the project's file structure. The first step is to create the directory in which Petalinux will configure and build the project. This is done as shown below, and the processor type is also specified at the same time.

```
dingus@dingus-ThinkPad-P53:~/vivadoprojects$ petalinux-create -t project -n z710
zoombo --template zynq
```

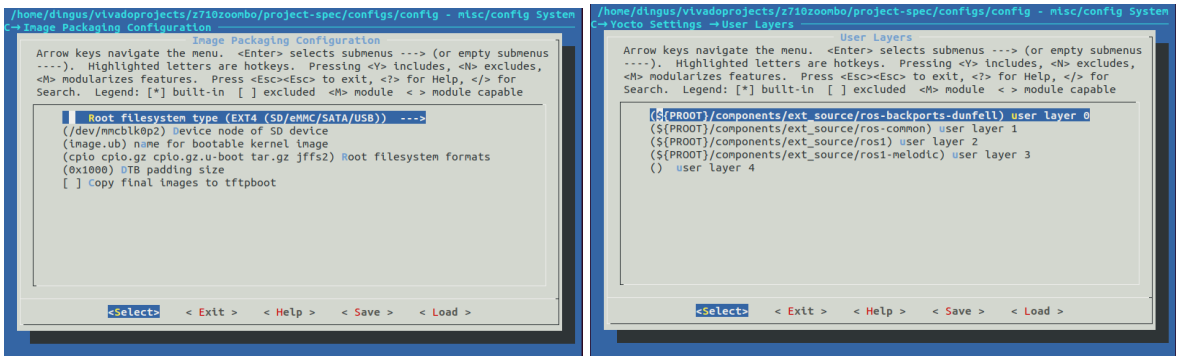
Petalinux command for initializing project directory

Everything for configuring and building will be done inside this directory. For initial configuration, the first thing needed is the FPGA design, which can be exported from Vivado as an .xsa file. The command for initial config and providing the path to this file is shown below:

```
dingus@dingus-ThinkPad-P53:~/vivadoprojects/z710zoombo$ petalinux-config --get-hw-description=/home/dingus/zybo_petali
nux/zynq_petalinux_wrapper.xsa
```

Petalinux command for setting up initial description of hardware

When this command is issued, Petalinux will show a GUI for additional configurations. Since an SD card is being used to boot from in our case, the filesystem type needs to be changed to EXT4, and TFTP boot can be disabled. Also done at this time is adding the user layers. Since the OpenEmbedded ROS layer, meta-ros, is not included as a provided Yocto layer, this is where the directory for these layers will be specified. It is important to keep in mind the dependencies for each of the meta-ros layers when adding, as any dependencies will need to be added first. There are no real standards for where these layers should be placed in the project directory, but one common practice is to place them in the "components" directory alongside the Yocto-provided layers, so that is the directory that was chosen for this project. One challenge that was presented when executing this step of the configuration process was that the newest version of Petalinux, 2021, does not allow adding user layers at this step, as any changes to the project structure cause the U-Boot and kernel configuration to fail, meaning that an entire build of linux must first be done without any additions, followed by reconfiguring with the user layers and rebuilding. In addition to this strange behavior, the 2021 version of Petalinux is based on the "Gatesgarth" release of Yocto, but fails to follow the syntax changes introduced in this release. For this reason, an older version of Petalinux tools needed to be used, or every recipe file would need to be rewritten in the older syntax. It appears that versions of Petalinux prior to 2019 no longer function due to deprecated layers being used, so the 2020 version was settled on as a functioning version of the tools that didn't require extensive rewrites of the chosen layers.



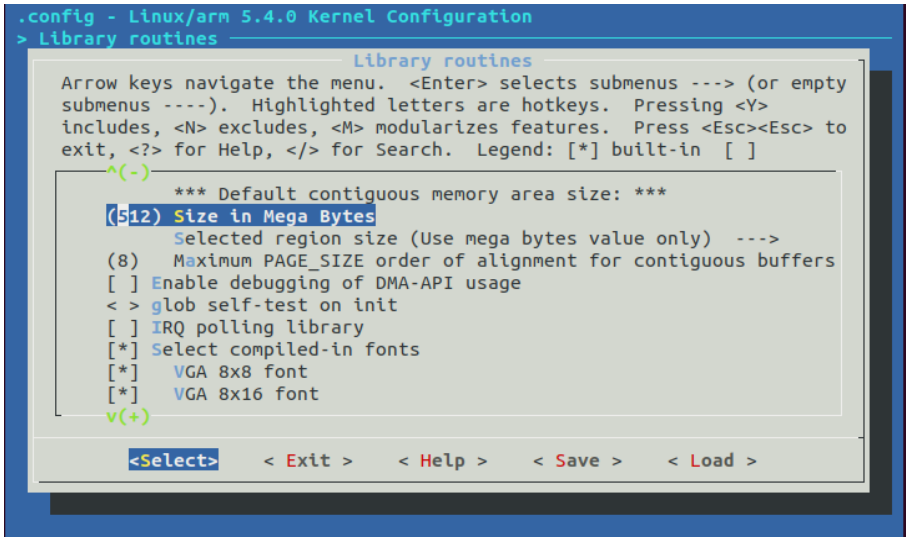

Initial Petalinux config and adding user layers

The next step in the configuration of the linux build is to set up U-Boot and the kernel. This is the most simple step in the configuration process, as there isn't much that needs changing for either of these.

```
dingus@dingus-ThinkPad-P53:~/vivadoprojects/z710zoombo$ petalinux-config -c u-boot
dingus@dingus-ThinkPad-P53:~/vivadoprojects/z710zoombo$ petalinux-config -c kernel
```

Petalinux commands for configuring U-Boot and the kernel

Only the kernel configuration in this step needs to have any changes. Because our image will have more features than the standard Petalinux image, the RAM size needs to be increased, in our case to 512MB (the size of the onboard RAM). Without this step, the install fails to boot, saying that there is insufficient space to unpack the initial startup processes. This error still happened when too many recipes were added, which caused many problems in the debugging for this project since the ROS packages were very resource intensive. Some preselected recipes could be removed to free up space, but there is no documentation on which recipes are even included, let alone which could be removed.

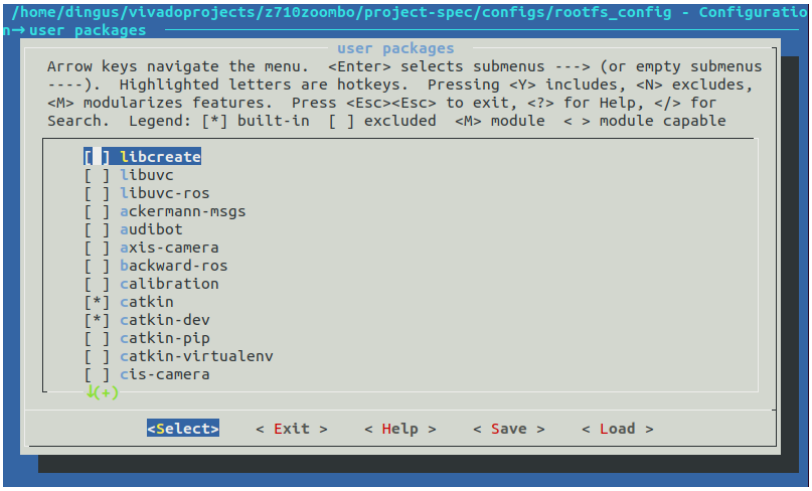



GUI configuration to increase memory allocation

The final step for configuring the project is to choose the desired BitBake recipes. Only absolutely necessary recipes were added to keep the install manageable and bootable. Since the ROS layers needed for this project are not a part of the provided Yocto layers, these needed to be added as user recipes. These can be added to the GUI by editing the user-rootfsconfig file, following the syntax shown below. A large majority of the packages that make up the ROS layer are found inside the generated-recipes folder of the ros1-melodic layer, so these recipes were manually copied into the config file. This excludes some key ROS packages however, such as roslaunch, so it was still necessary to search through all the layers to find essential recipes.

```
#Note: Mention Each package in individual line
#These packages will get added into rootfs menu entry

CONFIG_gpio-demo
CONFIG_peekpoke

CONFIG_ackermann-msgs
CONFIG_audibot
CONFIG_axis-camera
CONFIG_backward-ros
CONFIG_calibration
CONFIG_catkin
CONFIG_catkin-dev
CONFIG_catkin-pip
CONFIG_catkin-virtualenv
CONFIG_cis-camera
CONFIG_cmake
CONFIG_cmake-modules
CONFIG_common-msgs
CONFIG_control-msgs
CONFIG_control-toolbox
CONFIG_cv-camera
CONFIG_ddynamic-reconfigure
CONFIG_diagnostics
CONFIG_dynamic-reconfigure
CONFIG_driver-common
CONFIG_distance-map
CONFIG_dynamic-robot-state-publisher
CONFIG_cpp-common
CONFIG_gencpp
```

Example of adding recipes to user_rootfsconfig




User recipes as seen in the “user packages” menu

The only thing left at this point is to actually build the image, which is done with “petalinux-build”. The first build for a Petalinux project typically took about 2 hours on a 9th generation core i7 processor with 6 cores and 12 threads. After the initial build, all the generated recipes are cached in the project directory, so subsequent builds only take about 5 minutes per added recipe. The downside of having the build cached is that each Petalinux project takes over 30GB of space on the machine. This also has the downside of sometimes caching recipe builds that are non-functional, which cause a project to be unable to boot and require the creation of a fresh project (and another 2 hour initial build). During the process of debugging this project, this situation occurred about 20 times.

```
dingus@dingus-ThinkPad-P53:~/vivadoprojects/z710zoombo$ petalinux-build
INFO: Sourcing build tools
[INFO] Building project
[INFO] Sourcing build environment
[INFO] Generating workspace directory
INFO: bitbake petalinux-image-minimal
Parsing recipes: 100% |#########################################| Time: 0:01:40
Parsing of 4925 .bb files complete (0 cached, 4925 parsed). 6254 targets, 760 skipped, 1 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
WARNING: /home/dingus/vivadoprojects/z710zoombo/components/yocto/layers/meta-xilinx/meta-xilinx-bsp/recipes-ke
rnel/linux/linux-xlnx_2020.2.bb:do_compile is tainted from a forced run
WARNING: /home/dingus/vivadoprojects/z710zoombo/components/yocto/layers/meta-xilinx/meta-xilinx-bsp/recipes-bs
p/u-boot/u-boot-xlnx_2020.2.bb:do_compile is tainted from a forced run
Initialising tasks: 100% |#####################################| Time: 0:00:05
Checking sstate mirror object availability: 100% |###############| Time: 0:00:00
Sstate summary: Wanted 208 Found 192 Missed 16 Current 1294 (92% match, 98% complete)
WARNING: The gcc-cross-arm:do_configure sig is computed to be e6e99031a98ef12059ccbcbdedcdff684b845086e287d53b
10f8c31f8c7aeda1, but the sig is locked to cdeabaab964d2045029d9dad721209d09bf189192b2eee4c0ccdb4473c9942ba in
 SIGGEN_LOCKEDSIGS_t-x86-64-arm
The gcc-cross-arm:do_install sig is computed to be 1470b1d659572c4d2d634ddcdd3f764302c6fe0355e5373f978d0b6ed1e
0761a, but the sig is locked to 7dc0808ac0bcd58eb26072a3aa90e647100c9828b5fd731884193e0292b38812 in SIGGEN_LOC
KEDSIGS_t-x86-64-arm
The libgcc-initial:do_configure sig is computed to be 5ff3f79cf5fb36d39c8f68961ee4e6ef9efbe695e052e5142453fffc
aa800420, but the sig is locked to 98a6e8856579186f45d9211b037d9ecd85a7bcc21b48bc17828c9e59db43548b in SIGGEN_
LOCKEDSIGS_t-cortexa9t2hf-neon
NOTE: Executing Tasks
NOTE: Setscene tasks completed
NOTE: Tasks Summary: Attempted 5087 tasks of which 5073 didn't need to be rerun and all succeeded.

Summary: There were 3 WARNING messages shown.
INFO: copy to TFTP-boot directory is not enabled !!
[INFO] Successfully built project
```

Output from a successful Petalinux build

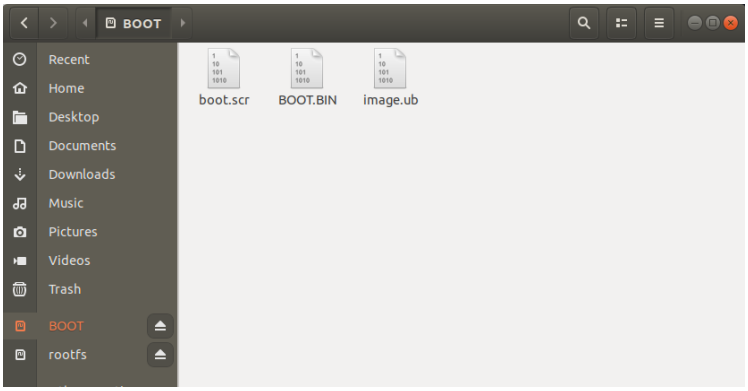
To boot the image, the “petalinux-package” command is used to create the boot files and compressed root filesystem. The boot files are simply placed in the first of two partitions on the SD card: a 1GB FAT32 partition. The rest of the SD card is formatted as EXT4 for the root filesystem, and the archive generated by the packaging command is extracted to this partition.

```
dingus@dingus-ThinkPad-P53:~/vivadoprojects/z710zoombo$ petalinux-package --boot --force --fsbl images/linux/z
ynq_fsbl.elf --fpga images/linux/system.bit --u-boot
INFO: Sourcing build tools
INFO: File in BOOT BIN: "/home/dingus/vivadoprojects/z710zoombo/images/linux/zynq_fsbl.elf"
INFO: File in BOOT BIN: "/home/dingus/vivadoprojects/z710zoombo/images/linux/system.bit"
INFO: File in BOOT BIN: "/home/dingus/vivadoprojects/z710zoombo/images/linux/u-boot.elf"
INFO: File in BOOT BIN: "/home/dingus/vivadoprojects/z710zoombo/images/linux/system.dtb"
INFO: Generating Zynq binary package BOOT.BIN...


****** Xilinx Bootgen v2020.2
  **** Build date : Nov 15 2020-06:11:24
    ** Copyright 1986-2020 Xilinx, Inc. All Rights Reserved.


[INFO]   : Bootimage generated successfully

INFO: Binary is ready.
```

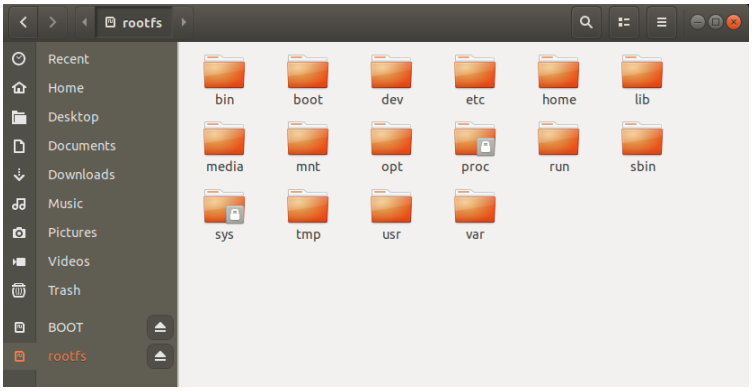
Output of the Petalinux packaging command

```
dingus@dingus-ThinkPad-P53:~/vivadoprojects/z710zoombo/images/linux$ tar -xzvf rootfs.tar.gz -C /media/dingus/rootfs/
```

Command to extract root filesystem to SD card




Contents of the FAT32 boot SD card partition




Extracted contents of the root filesystem SD card partition

Booting the image is very simple. Once the programming jumper on the Zybo board is switched from JTAG to SD, the SD card can be inserted into the board, causing the board to initialize the embedded linux build automatically on boot. In order to interface with the linux install, a serial monitoring program, such as Putty, can be opened to monitor the USB interface

connecting and powering the board. This essentially allows for using the embedded linux install in the same way as any other linux command line interface.

**Adding ROS to the Build:**

The previous steps created a working ROS install on the embedded linux image, but does not put the code for the ROS project on the board. To accomplish this, there are a couple different methods. The first attempted method was to add all the necessary ROS tools and C++ compilers necessary to create a ROS workspace on the board and compile ROS code with catkin_make. With this build, the code can be pulled from git, then built and run directly on the board. The upsides of this method are that the linux image only needs to be successfully built once, and whenever code changes it can simply be grabbed from the git page and rebuilt. One major downside, however, was that the board needed to be connected via ethernet to grab code, which poses a problem on networks like those at Oakland University that require navigating to a login screen to access the internet. This method was ultimately abandoned, as the necessary build tools and ROS modules added too much bloat to the system, causing it to fail to boot.

The chosen method, then, was to compile the ROS code directly into the embedded linux build. While this requires recompiling every time code is changed, the fact that all the built recipes are cached means that rebuilding with only one changed BitBake recipe only takes about a minute. The major benefit to this method is that the C++ build modules and ROS build tools aren't necessary, freeing up space for essential ROS functions. Additionally, this makes debugging dependencies for the project easier, since the project build will fail when building the linux image, instead of after booting into the embedded linux install, creating a ROS workspace, pulling code from git, and running catkin_make on the board.

The BitBake recipes for ROS recipes located in the ros1-melodic layer were referenced to create the recipe for the project. Based on this template, there were three major changes that needed to take place to personalize the recipe. The first, and most obvious, was to update the link to the project repository. The only peculiarity here is that the commit number needs to be given for BitBake to grab the code. This means that anytime the code in the repository is updated, the recipe must also be updated with the new commit number. The remaining two changes are both based upon the package.xml for the ROS project. Using the build, execute, and test dependencies defined in the package.xml, the corresponding variables in the recipe can be filled out. Finally, the checksum for the line in the package.xml file that describes the license for the project must be calculated to verify the correct code is being compiled. It was initially confusing trying to figure out how this is calculated, but one failed build with the recipe gave an error that also included the correctly calculated checksum value. Instead of creating an entire user layer for the one recipe, an additional folder was added to the ros1-melodic layer where the recipe could be placed. This avoided the need to create configuration files for a new layer, and ensures that the recipe is compiled along with other recipes that share similar dependencies.

```
inherit ros_distro_melodic
inherit ros_superflore_generated

DESCRIPTION = "ZyboZoombo, control a robotic vehicle using ros and communicating with FPGA"
SECTION = "devel"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://package.xml;;beginline=7;endline=7;md5=58e54c03ca7f821dd3967e2a2cd1596e"

ROS_CN = "zybo-zoombo"
ROS_BPN = "zybo-zoombo"

ROS_BUILD_DEPENDS = " \
    roscpp \
"

ROS_BUILDTOOL_DEPENDS = " \
    catkin-native \
"

ROS_EXPORT_DEPENDS = " \
    roscpp \
"

ROS_BUILDTOOL_EXPORT_DEPENDS = ""

ROS_EXEC_DEPENDS = " \
    roscpp \
"

ROS_TEST_DEPENDS = " \
    roslaunch \
"

DEPENDS = "${ROS_BUILD_DEPENDS} ${ROS_BUILDTOOL_DEPENDS}"

DEPENDS += "${ROS_EXPORT_DEPENDS} ${ROS_BUILDTOOL_EXPORT_DEPENDS}"

RDEPENDS_${PN} += "${ROS_EXEC_DEPENDS}"

ROS_BRANCH ?= "branch=main"

SRC_URI = "git://github.com/adejonge/ZyboZoombo.git;${ROS_BRANCH};protocol=https"

SRCREV = "ebd40241c8fff7031ec2f28b52862cbb5b99fa89"

S = "${WORKDIR}/git"

ROS_BUILD_TYPE = "catkin"

inherit ros_${ROS_BUILD_TYPE}
```

BitBake recipe for ROS project source code

**PID Control Theory:**

PID control is an effective way to control heading error. We use this to control the ackermann steering of our vehicle.

Our system operates on the horizontal count pixel value of the calculated centroid from hardware. The difference between the pixel and the central pixel of 675(calculated empirically because our camera is slightly off centered.

So lets say the horizontal count pixel is 625. That means there is an error of -50. This means that the bot should turn left, because the pixel is to the left. A px count of 700 for example should make it turn right instead.

PID control is actually quite easy for us because we already tie our reports to the image frame. Therefore our DeltaT is already set. Our DeltaT is simply (1/FREQinHz)*1000. We multiply by 1000 because we want to give some granulation, and it makes our pixel difference not massive. Essentially because we're working on the order of 1280 (horizontal), we're making it roughly the same order.

For the present. Simply its the subtraction error. So **Error - TARGET = P;**

For the Integral (Future), its the sum of the previous integral error and the error*DeltaT

**So I=I+ P*deltaT**

For the Derivative Part, its the difference of the previous error to the current error over deltaT.

**SO D= (P - D) /deltaT.**

In general, More P means we correct more or less based on the difference. Its sort of the scaling for the PID controller.

More I means that we increase the reaction time of the controller, but we are prone to oscillating because we also increase the amount we turn in response as well.

More D means that we smooth out the oscillations, but we slightly lower response time. D is also the most computationally intensive because it uses division and requires a previous iterative value.

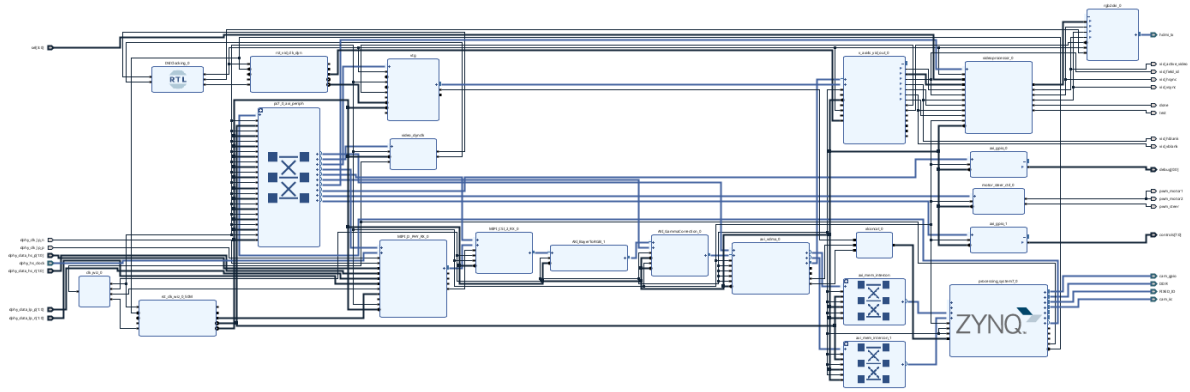
**Empirically we tested and got the following PID control to stay within the lines reasonably:**

**P = 0.3**

**I = 0.4**

**D = 0.1**

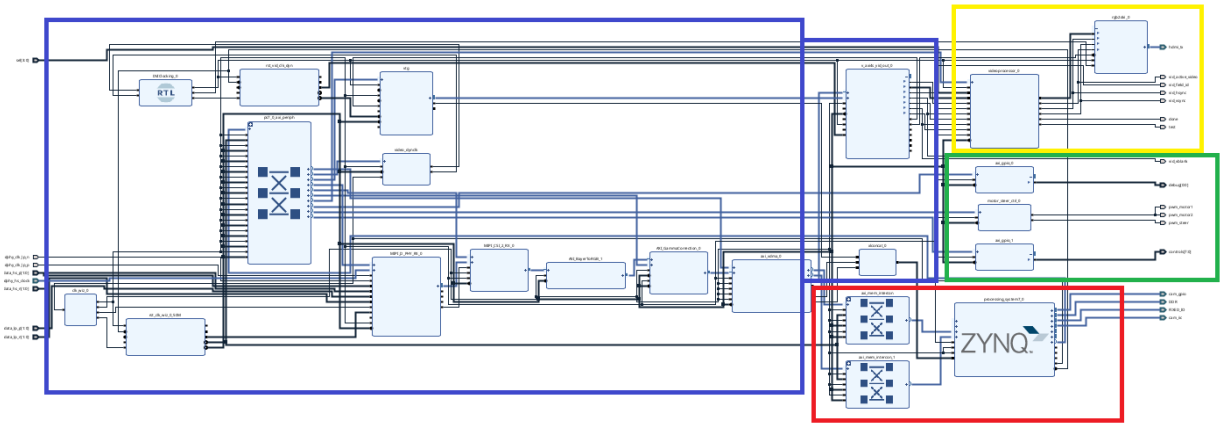
**Hardware Definition:**



The Hardware (Known as the Programmable Logic) for this project is quite expansive and complicated, and is outside the scope of the class. However, a brief hardware description is in order.

The hardware is comprised of 4 major parts.

- HDMI Controller
- PWM Controller
- Video Operator/ Centroid Finder
- Programmable software chip.



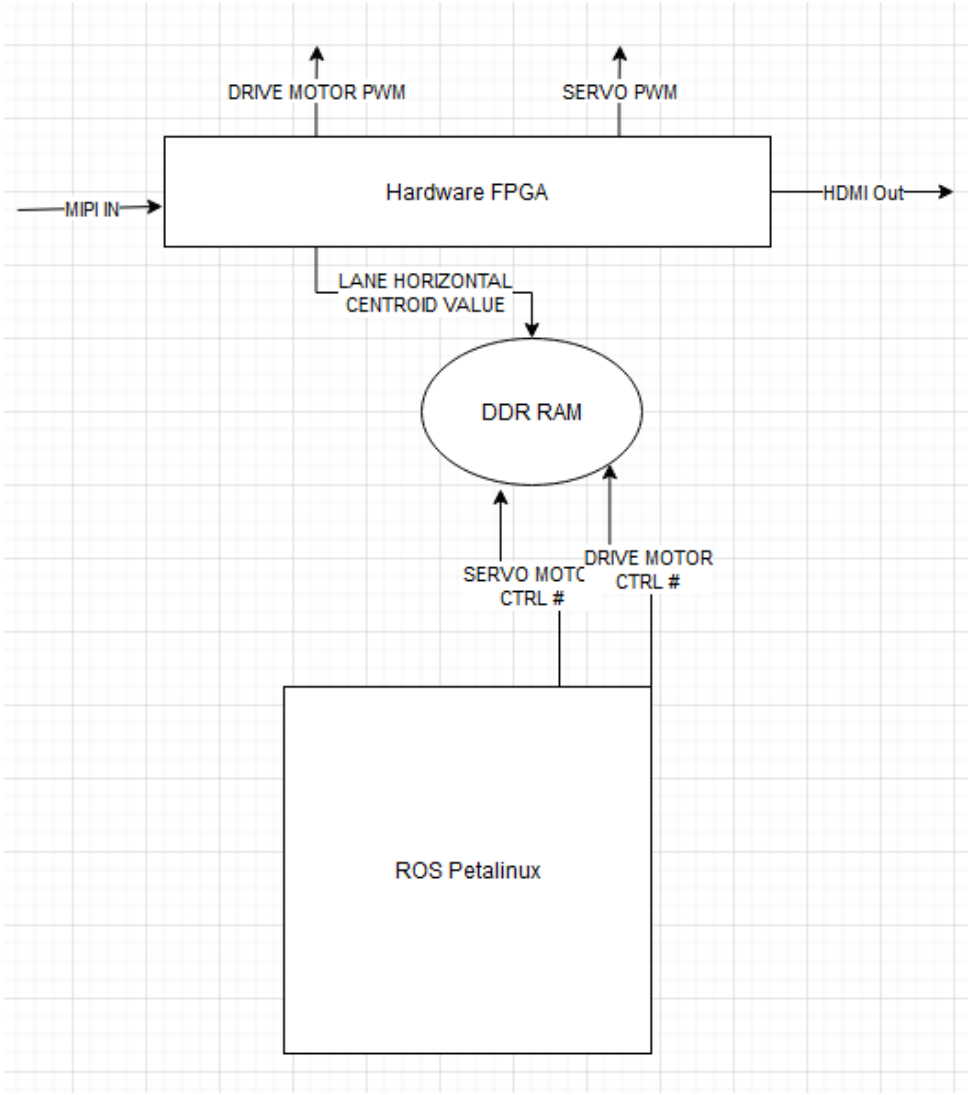
**BLUE: HDMI Controller**
**YELLOW: Video Operator/Centroid Finder**
**GREEN: PWM Controller**
**RED: Programmable Software Chip.**

The HDMI controller (BLUE) takes a MIPI PCAM 5 interface, and takes the rasters of the image, does some video editing work, and then outputs its as an HDMI Signal to the HDMI port. Using wireless HDMI this is how the image appears on the screen.

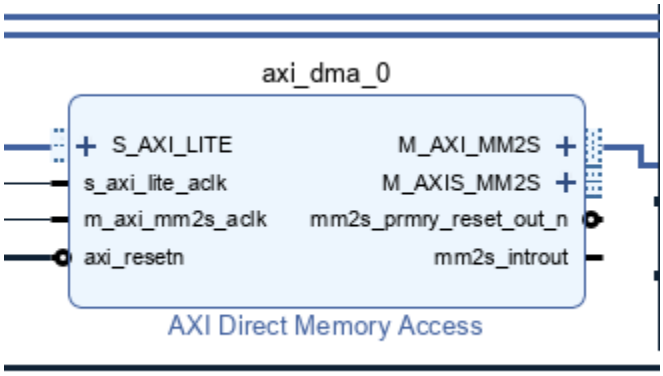
The Video Operator/ Centroid Finder does a few image operations on the rasters, and then does an aggregate sum average of a thresholded value to find 2 centroid locations, one for each lane line. These are what is sent to the ROS petalinux build.

The PWM controller takes a control variable between the number 0 and 200, and converts it into a workable PWM signal to send to the motors via a PMOD port on the board.

The Zynq processor is the processing system that we run our ROS petalinux on. It also contains a cache memory that we utilize to communicate between hardware and ROS.




If we black box this hardware system, we can see that there's a MIPI input videostream, a HDMI output video stream, 2 PWM control signal outputs from the FPGA, and it writes 2 centroid values in DDR ram. Our ROS system in software needs to simply write the 2 motor controls to the shared DDR ram, and read the 2 centroid values from the DDR ram for output. We do this by using fixed memory. We read and write from set memory locations using DMA.

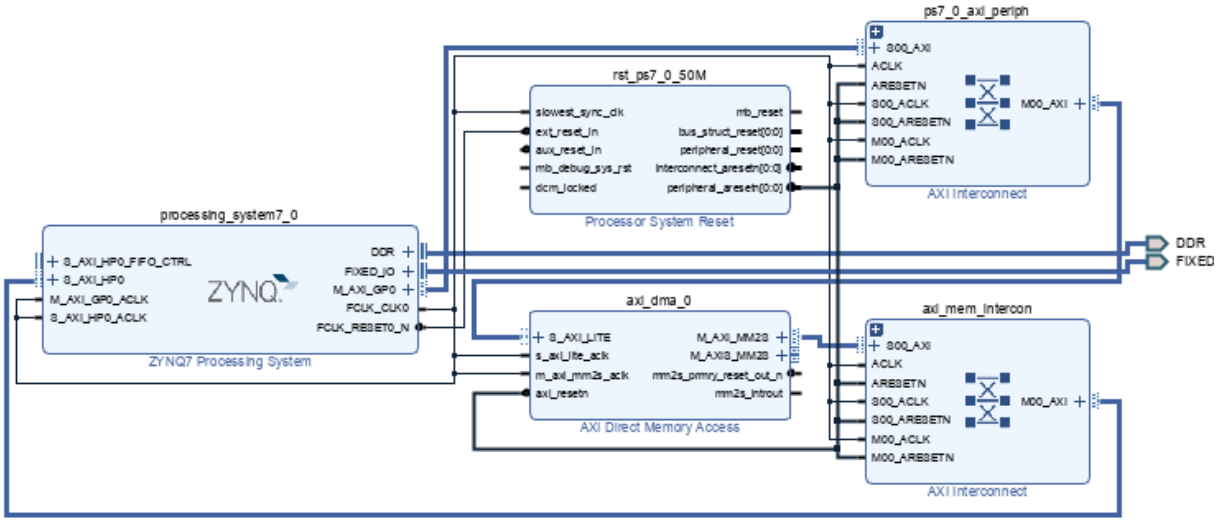



Now that we have a set location in memory that we need to read our commands from, we need to use a hardware component to continuously poll that location in memory so that we can use it in the hardware, and eventually the PWM controller.

This component is called an AXI DMA, which means an AXI (protocol) data memory access controller. This is a component provided by xilinx that pulls from system memory from the DRAM, and normally its used to pull from memory to software (PS). however, the axi dma has a register that it uses to pipeline its data, so if you need memory in the hardware, we simply just need to hook up w/e we need to the MM2S line. MM2S stands for memory map to stream, stream meaning stream back to processor, and memory map, meaning from the mapper.

The mapper is a built in hardware component that organizes and completes requests from the Programmable Logic and Programmable Software. The problem with this method, and the method chosen, is that it works upon requests, not commands. The memory mapper is on its own schedule, so it will respond with memory when it wants to (when its able to). This was a problem we'll discuss later with sending PWM values through this method.

We technically don't need a full DMA, because we're reading from memory and utilizing it in the hardware, not the software of the chip, but its easier to implement using a block diagram with a DMA because for reads its already written for us, instead of having to make an RTL component.




The DMA requires a little bit of hardware to go along with it. Processor system reset syncs the axi_dma clock with the processor system (the ROS one), and also syncs all the components with it. The interconnect is required to convert from the HP0 port (memory port), and axi hardware. There's a buffer in there that prevents loss of data, as well as the ability to split streams if necessary. This component is used heavily in the HDMI for hardware, which is outside the scope of this class.

2 of these interconnects are required. One from the command interface, and once for the memory interface. The command interface is how we control the axi_dma, we would normally get this from the Zynq processor, however, for this application, we can simply hardcode values.

The main values we write to the data memory access controller is:

**Data_addr = 0x0000F000 : the location of the memory for Servo_Sig;**

**Data_addr2 = 0x0000F004 : the location of the memory for Drive_Sig;**

These can be multiplexed and stored into registers for easy access to these numbers.

For writes we technically don't need an entire DMA to do it, because we're writing from hardware, not from software. We simply need an axi_full interface to interface with HP1. To write the 2 centroid values, our max value for each centroid is 1280 (because we're in 720p) which requires 11 bits. Therefore we can fit both numbers into a 32 bit word. These are the constraints we need to send the Axi_full request. We utilize the axi_full template provided by xilinx.

**Addr - 0x0000F008: the location of the centroid values**

**Mode - 0 : Fixed Mode**

**Burst size - 128 ;**

**Length - 1;**

These parameters are saying the following. Read exactly 1 32 bit word from memory 0x0000F008. Then keep the pointer at that location, and read it again on the next axi clock available, 128 times.

Once this is executed 128 times, it will look for the next state that these signals are in.

However, we have hardcoded them, so it will instead keep looking at this one memory location indefinitely.

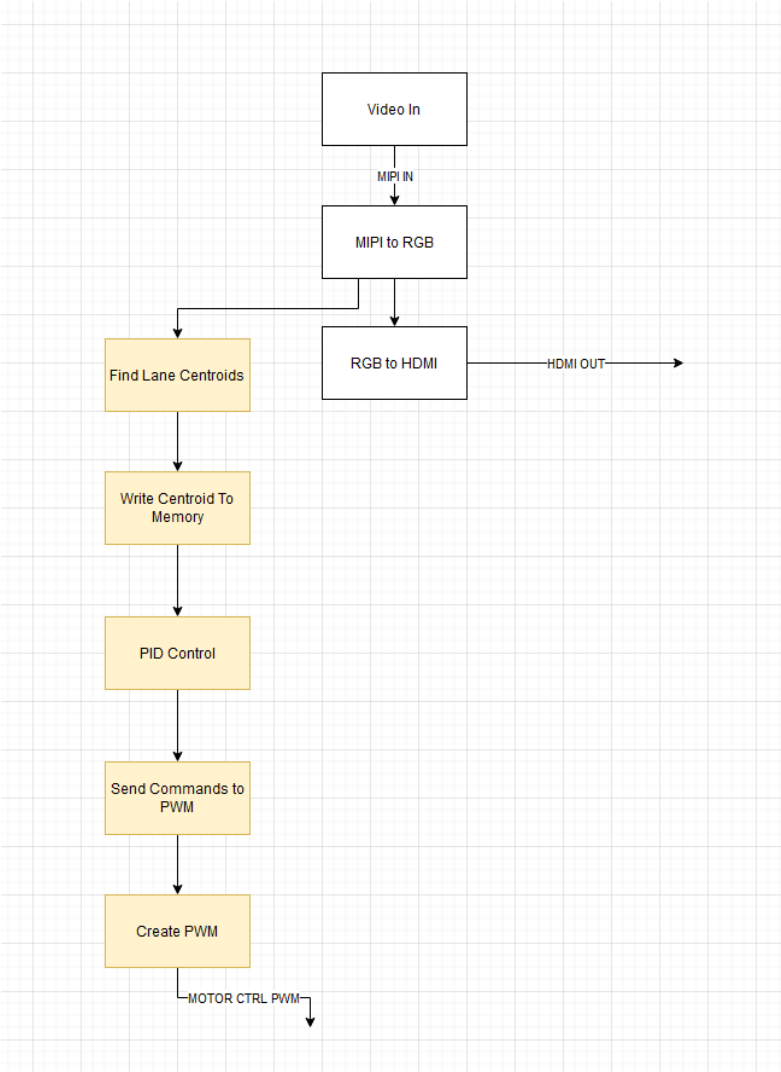
**System Limits:**

It is very easy to find our system limits because our system reports to the output PWM controller every frame. The critical path is near timezero after that because it is all hardware, so the timing can be estimated on each frame. Because we are using 720p60, this means our response time is roughly 16.7 ms… per frame, however. We use the lowest 3 points, to create a 3 frame buffer, to enhance consistency, so it is pipelined. Because it is pipelined, it reports every frame but has a delay of three frames. Thus our response time is roughly 41.6 ms. If we were to utilize the 1080p30 functionality of our camera, our response time would be roughly 99.9 ms.
Fastest Response Time (30Hz) = 99.9ms;
Fastest Response Time (60Hz) = 41.6ms;

To put this in perspective, the generalized driver total reaction time that we found, used as a baseline for autonomous vehicles is roughly a second and a half from initial view to reaction. AVS requirements require a response time of 0.5 seconds. Most Tesla autonomous vehicles have standard response times of 200 - 300 ms depending on stimulus. The research level of the new safety system proposed technology of the age has gotten down to roughly 17 - 19 ms. We surmise that these software use a form of pure asic design to make it so the response time is much faster than before, along with using higher slices with a delay to affect the Integral part of the PID control.
Driver Total Reaction Time = 1.5s
AVS Safety System response requirement = 500ms
AVS Safety System Avg response time = 200 - 300ms
AVS new Safety System tech = 19ms;

**Conclusion & Challenges:**

We always knew the system would be fairly light on the ROS side. The original idea was to take the RGB stream in, do some image processing, find the centroids, do the PID control, and create the PWM signals to send to the motors all in the programmable software.




The actual system simply reads from memory the found centroid, does the PID control, and then sends back a command value to the PWM controllers in hardware. There were 3 main reasons why we had to export the Centroid finding (image processing) and the PWM signaling to Hardware instead of software.

The first major reason was dependencies. Because we had to make a new petalinux build for every dependency that each library had, it really took alot of the time up just to get the basic dependencies completed. We dabbled with some OpenCV stuff, as well as additional

timing libraries, but the dependencies were difficult to fully implement and after spending 100+ hours getting base ROS and the standard libraries in the petalinux build, we really needed to work on getting a working system up and running.

OPENCV is fairly lightweight, but there's one major issue. We're using 720p60 video and we really wanted to keep that timing and resolution. We're using an embedded processor that frankly isn't very great. It has a max clock speed of 600MHz. Roughly, and that simply isn't fast enough to do all the image processing required. We'd have to lower the framerate to 30 and most likely have to use 240p video to meet timing.

The PWM generation was exported to hardware for a different reason. Our solution for BRAM reading was using the memory mapper of the processor for reading back to hardware. Unfortunately I was choosing my words carefully when i said "AXI REQUEST". It's not a command, it's a request. The memory mapper on the chip is a fickle beast, and while most times it's fairly good at doing things when you want it to. It's not guaranteed, and that was a problem. It sometimes would just decide not to update values when prompted, and for something as timing sensitive as a PWM controller, unfortunately, it just doesn't fit the bill, I'm sure there's another solution. We did some research into TIMING sensitive DMAs, but none of us have experience with them to make sure timings are met. Therefore we decided just to remove the PWM controller back to hardware and send command numbers.

**The final solution was a hybrid system autonomous vehicle that would consistently stay within the lines, and we're happy with the way the project turned out. We created a system with a 41.6 ms response time, well below what we thought we could. This is just the beginning of what can be done using a hybrid system. The fact you can do high resolution image processing with ease gives large potential and we are just now starting to see it in industry with FPGA assisted memory readers for cameras for automotive safety systems.**

**References:**


**Running Petalinux on Zynq SoC From Scratch - Zybo Board**
https://nuclearrambo.com/wordpress/running-petalinux-on-zynq-soc-from-scratch-zybo-board/

**ROS 2 in Kria kv260 with Petalinux 2021.2**
https://www.hackster.io/jlamperez10/ros-2-in-kria-kv260-with-petalinux-2021-2-92ec3a

**Meta-ros OpenEmbedded Build Instructions**
https://github.com/ros/meta-ros/wiki/OpenEmbedded-Build-Instructions

**Adding GCC to PetaLinux builds (compiling code on FPGA PS)**
https://www.centennialsoftwaresolutions.com/post/adding-gcc-to-petalinux-builds-compiling-code-on-fpga

**How to boot petalinux image with >1.5 GB root file system on both petalinux-qemu and sd card on zcu104 board**
https://support.xilinx.com/s/question/0D52E00006hpXRvSAM/how-to-boot-petalinux-image-with-15-gb-root-file-system-on-both-petalinuxqemu-and-sd-card-on-zcu104-board?language=en_US

**PetaLinux Tools Documentation**
https://docs.xilinx.com/v/u/2020.1-English/ug1144-petalinux-tools-reference-guide

**Create a bitbake recipe for a ROS package and cross-compile it using the meta-ros project**
https://github.com/vmayoral/diving-meta-ros