SOFTWARE EXPLANATION

The software for our project takes in an image and performs several operations included in OpenCV on it, ultimately calculating the duty cycle required by our motors to turn the vehicle in the expected direction. First, the image is transformed into the HSV color space, and the color blue is isolated. Our design relies on lane lines being the color blue. Next, Canny edge detection is used to find the edges of where the blue lane lines are. The top part of the image is cropped away, as our vehicle only needs to look at the lane lines close to it. Next, the Hough transform is used to detect line segments. Each line segment is iterated through and its slope and y-intercept are found. Segments are categorized as being of the left lane or right lane depending on their location in the image. The slopes and intercepts of each of the segments in the left and right lanes are averaged, leading to an averaged line for both sides of the lane. There may be some situations in which only one lane line is detected. In those cases, the vehicle will turn sharply to avoid that line.

Now that lines representing the lane lines have been found, the turn angle must be found. A line is created that starts at the bottom middle part of the image, and ends at the midpoint between the two lane lines in the middle part of the image. Trigonometric functions are then used to calculate the angle this drawn line would make with the horizontal. The motors for our vehicle are able to do angles of 45 to 135 degrees, where 90 degrees is going straight. Experimentation with the motors found that a 6% duty cycle causes the motor to turn fully to the right, while a 12% duty cycle causes it to turn fully to the left. Another function was created to convert the desired turn angle into the duty cycle percentage needed to achieve that turn angle. Assuming that the PWM module consisted of a counter of up to 1024, the program then calculated the value that the module would need to count up to in order to achieve the duty cycle desired.

HARDWARE EXPLANATION

The motor control and steering control modules were designed using the Xilinx Vivado Design Suite. The IPs required two new AXI4 peripherals to be created to act as drivers for the SoCar's hardware, specifically the motor and steering servo. The modules are built off a modified version of the custom IP block that was previously made, having register-controlled switches and PWM logic integrated using VHDL, the main difference being that the new module now outputs to a pin. One instance of the IP would be used to control the steering servo and a second instance controls the speed of the car motor.

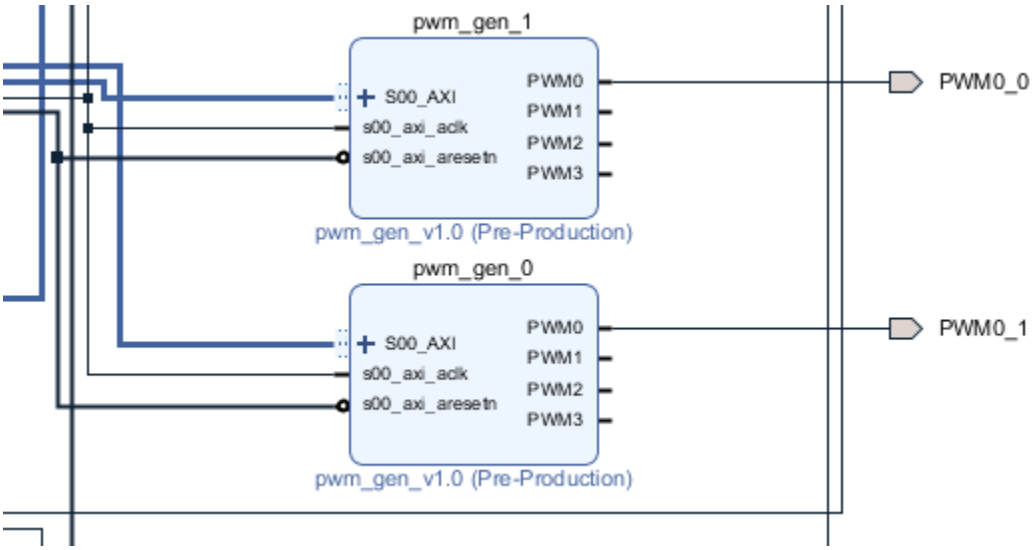


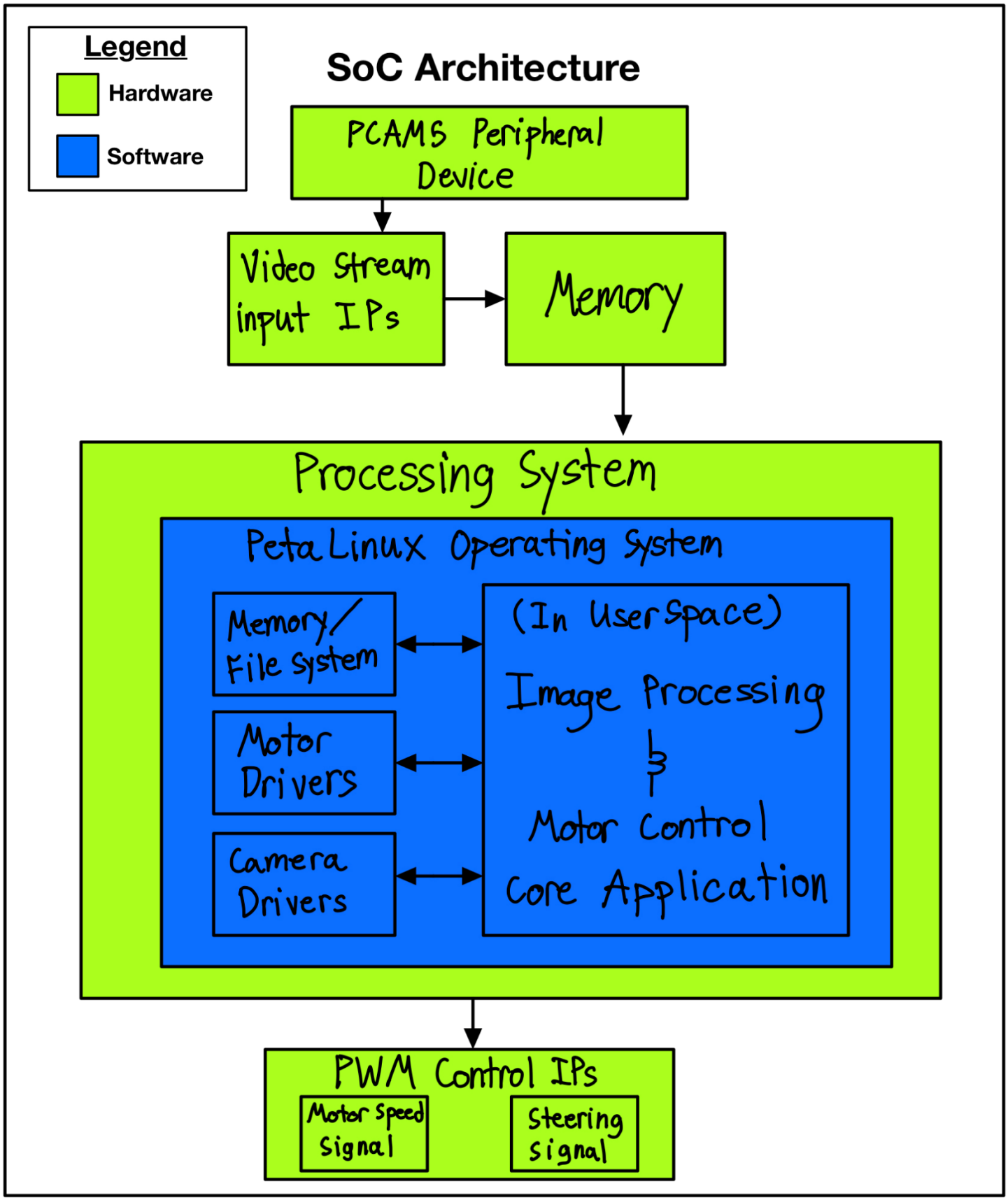
Figure 1: PWM Generator IP Instances for motor and steering control

Each module is connected to the Zynq Processing System via an AXI Interconnect and consists of a customizable PWM generator implemented using the Xilinx Software Development Kit. The count value input parameter would be used to set a duty cycle for the PWM generator. These different duty cycle square waves, operating at a frequency of 60 Hz, would control the movement functionalities of the car.

The PCam5 was connected to the Zynq Processing System via the AXI4-Lite Bus. The raw pixel data could then be processed by the MIPI D-PHY IP core, which is designed for transmission and reception of video or pixel data for camera and display interfaces, as well as the MIPI CSI-2 Receiver IP. The camera module itself is configured as an AXI GPIO block.

The PCam5 IPs and other components can be seen in the attached "SoCarBlockDesign.pdf" file.

SOC ARCHITECTURE



PROGRAM CODE

socarmain.cpp (used for gathering images from camera and vision processing)

```
#include <opencv2/opencv.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <iostream>
#include <cmath>
#include <pthread.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <poll.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/ioctl.h>

#include "charvideo_ioctl.h"


using namespace std;
using namespace cv;

#define STEERING_DRIVER "/proc/pwmgen"
#define SPEED_DRIVER "/proc/pwmgen2"
#define CAMERA_DRIVER "/dev/video"
//#define IMAGE_NAME "cameraout.ppm"
#define IMAGE_NAME "cameraout.jpg"
#define PWM_RANGE 1048575
#define BUFF_SIZE 20

void *setMotorSpeed(void *in);

/*
      C++ Code based on the Python code available at
      https://towardsdatascience.com/deeppicar-part-4-lane-following-via-opencv-
737dd9e47c96

*/

Mat edgeDetect(Mat image);
Mat croppedImage(Mat image);
vector<Vec4i> detectSegments(Mat image);
vector<Vec4i> avgSlopeIntercept(Mat image, vector<Vec4i> segs);
Vec4i makePoints(Mat image, float slope, float intercept);
int findDuty(int angle);

int main() {
```

```
	pthread_t speedThread;
	//pthread_create(&speedThread, NULL, &setMotorSpeed, NULL);

	while (1) {

		int fd;
		fd = open(CAMERA_DRIVER, O_RDONLY);
		if (fd < 0) {
			printf("Can't open %s\n", CAMERA_DRIVER);
			return -1;
		}

		//Print the VDMA's status to kernel log
		ioctl(fd, CHARVIDEO_IOCSTATUS);

		//Get the image sizes from the video driver
		int h, w, l;
		h = ioctl(fd, CHARVIDEO_IOCQHEIGHT);
		w = ioctl(fd, CHARVIDEO_IOCQWIDTH);
		l = ioctl(fd, CHARVIDEO_IOCQPIXELLEN);

		unsigned char buf[h * w * l];
		read(fd, buf, w * h * l);
		close(fd);

		char filename[100];
		sprintf(filename, "%s", IMAGE_NAME);
		FILE *outimg = fopen(filename, "wt");

		//if the pixel length is only 1 byte, then the image is grayscale (ppm
format 5)
		if (l == 1)
			fprintf(outimg, "P5\n%d %d\n%d\n", w, h, 255);
		else
			fprintf(outimg, "P6\n%d %d\n%d\n", w, h, 255);

		printf("Opened %s\n", filename);

		//The images are stored in the VDMAs in the BGR format so it must be
		//changed to RGB for human understandable images

		if (l != 1) { //BGR to RGB
			for (int i = 0; i < w * h * l; i += 3) {
				uint8_t aux = buf[i + 2];
				buf[i + 2] = buf[i];
				buf[i] = aux;
			}
		}

		fwrite(buf, 1, w * h * l, outimg);
```

```
	fclose(outimg);


	//Load the image
	Mat img = imread(IMAGE_NAME);
	//Mat img = imread("resizedOrig.jpeg");
	//Mat img = imread("oneLane.jpg");
	//Mat img = imread("road-image.jpg");

	Mat img_hsv;

	//Isolate edges
	Mat edges = edgeDetect(img);

	//In lane navigation, only bottom half of image is needed
	//Only the lanes close to the car matter
	//Can crop out the top half
	Mat cropped = croppedImage(edges);
	//imshow("Cropped", cropped);

	//Now need to detect line segments
	//Uses Hough transform
	vector<Vec4i> segs = detectSegments(cropped);

	//Find the average slope/intercept of the left and right lanes
	vector<Vec4i> avgLines = avgSlopeIntercept(img, segs);

	//Draw lines onto img
	for (int i = 0; i < avgLines.size(); i++) {
		int x1 = avgLines[i][0];
		int y1 = avgLines[i][1];
		int x2 = avgLines[i][2];
		int y2 = avgLines[i][3];
		Point p1(x1, y1);
		Point p2(x2, y2);

		line(img, p1, p2, Scalar(0, 255, 0), 2, LINE_8);

	}
	//imshow("Out", img);

	//Steering
	//If two lines found, can compute direction by averaging endpoints
	//Dir line will be from bottom middle, to that midpoint
	int xOffset;
	int yOffset;
	if (avgLines.size() == 2) {
		int Lx1 = avgLines[0][2];
		int Rx1 = avgLines[1][2];
```

```
		int mid = img.cols / 2;
		xOffset = (Lx1 + Rx1) / 2 - mid;
		yOffset = img.rows / 2;
	}

	//If only one lane found, need to turn hard to other side
	//Can do by making dir line same slope as seen line
	else if (avgLines.size() == 1) {
		int Fx1 = avgLines[0][0];
		int Fx2 = avgLines[0][2];
		xOffset = Fx2 - Fx1;
		yOffset = img.rows / 2;
	}

	//Offsets are calculated, now need to find angle
	double angToMid = atan((double)xOffset / (double)yOffset);
	//Con to degrees
	int angDegree = (int)(angToMid * 180.0 / CV_PI);

	int steeringAngle = angDegree + 90;


	if (steeringAngle < 45) {
		steeringAngle = 45;
	}
	else if (steeringAngle > 135) {
		steeringAngle = 135;
	}

	//Find Duty Cycle Percentage
	//12% is fully left
	//6% is fully right
	int dutyCount = findDuty(steeringAngle);


	//Display steering line
	double steerRad = (double)steeringAngle / 180.0 * CV_PI;
	int sx1 = img.cols / 2;
	int sy1 = img.rows;
	int sx2 = (int)(sx1 - img.rows / 2 / tan(steerRad));
	int sy2 = img.rows / 2;
	Point ps1(sx1, sy1);
	Point ps2(sx2, sy2);

	line(img, ps1, ps2, Scalar(255, 0, 0), 2, LINE_8);
	imwrite("out.jpg", img);

	char outString[BUFF_SIZE];
	sprintf(outString, "%d", dutyCount);
```

```
            cout << steeringAngle << "\n";

            FILE *steeringDriver = nullptr;//fopen(STEERING_DRIVER, "w");
            if (steeringDriver) {
                  fputs(outString, steeringDriver);
                  fseek(steeringDriver, 0, SEEK_SET);
                  fclose(steeringDriver);
            }
      }

      return 0;
}

void *setMotorSpeed(void *in) {
      char inBuff[BUFF_SIZE];

      while(1) {
            fgets(inBuff, BUFF_SIZE, stdin);

            FILE *speedDriver = fopen(SPEED_DRIVER, "w");
            if (speedDriver) {
                  fputs(inBuff, speedDriver);
                  fseek(speedDriver, 0, SEEK_SET);
            }

            fclose(speedDriver);
      }
}

int findDuty(int angle) {
      int adjusted = 90 - angle;
      float adjuster = (2.384/90.0) * adjusted;
      float dutyCycle = 7.152 + adjuster;
      float perc = dutyCycle / 100.00;
      int count = (int)(perc * PWM_RANGE);

      return count;
}

vector<Vec4i> avgSlopeIntercept(Mat image, vector<Vec4i> segs) {
      //Goal is to combine the segments into left and right lane
      //If slope < 0, left lane
      //Slope > 0, right lane
      int height = image.rows;
      int width = image.cols;

      float boundary = 0.33;
      //Left should be on left 2/3
      int leftBoundary = width * (1 - boundary);
      //Right should be on right 2/3 of image
```

```
	int rightBoundary = width * boundary;

	vector<float> leftSlopes;
	vector<float> rightSlopes;
	vector<float> leftInts;
	vector<float> rightInts;

	//Iterate through each segment in segs
	for (auto it = begin(segs); it != end(segs); ++it){
		int x1 = it->val[0];
		int y1 = it->val[1];
		int x2 = it->val[2];
		int y2 = it->val[3];

		if (x1 == x2) {
			//Vertical line segment
			continue;
		}
		//Not vertical
		//Find slope/intercept
		int rise = y2 - y1;
		int run = x2 - x1;
		float slope = (float)rise / (float)run;
		float intercept = y1 - (slope * x1);

		if (slope < 0) {
			//left lane
			if (x1 < leftBoundary && x2 < leftBoundary) {
				leftSlopes.push_back(slope);
				leftInts.push_back(intercept);
			}
		}
		else {
			//right lane
			if (x1 > rightBoundary && x2 > rightBoundary) {
				rightSlopes.push_back(slope);
				rightInts.push_back(intercept);
			}
		}
	}

	//Get avgs of the left and right segs
	float avgLSlope;
	float avgRSlope;
	float avgLInt;
	float avgRInt;
	float temp = 0;
	for (int i = 0; i < leftSlopes.size(); i++) {
		temp = temp + leftSlopes[i];
	}
```

```
	avgLSlope = temp / leftSlopes.size();
	temp = 0;

	for (int i = 0; i < rightSlopes.size(); i++) {
		temp = temp + rightSlopes[i];
	}
	avgRSlope = temp / rightSlopes.size();
	temp = 0;

	for (int i = 0; i < leftInts.size(); i++) {
		temp = temp + leftInts[i];
	}
	avgLInt = temp / leftInts.size();

	temp = 0;
	for (int i = 0; i < rightInts.size(); i++) {
		temp = temp + rightInts[i];
	}
	avgRInt = temp / rightInts.size();
	temp = 0;

	//Averages have been found
	//Checks to make sure it sees at least one seg first
	vector<Vec4i> lines;

	if (leftSlopes.size() > 0) {
		Vec4i left = makePoints(image, avgLSlope, avgLInt);
		lines.push_back(left);
	}

	if (rightSlopes.size() > 0) {
		Vec4i right = makePoints(image, avgRSlope, avgRInt);
		lines.push_back(right);
	}

	return lines;
}

Vec4i makePoints(Mat image, float slope, float intercept) {
	Vec4i line;
	int height = image.rows;
	int width = image.cols;

	int y1 = height;
	line[1] = y1;
	int y2 = (int)(y1*0.5);
	line[3] = y2;

	int x1 = (int)((y1 - intercept) / slope);
	if (x1 > 2 * width) {
```

```
            x1 = 2 * width;
      }
      if (x1 < -1 * width) {
            x1 = -1 * width;
      }

      line[0] = x1;

      int x2 = (int)((y2 - intercept) / slope);
      if (x2 > 2 * width) {
            x2 = 2 * width;
      }
      if (x2 < -1 * width) {
            x2 = -1 * width;
      }
      line[2] = x2;

      return line;
}

Mat edgeDetect(Mat image) {
      //Image converted to HSV space
      Mat img_hsv;
      cvtColor(image, img_hsv, COLOR_BGR2HSV);
      //Lifting out the blue
      //Blue is in range 60-150 in OpenCV (Others may use 120-300 range)
      Mat mask;
      inRange(img_hsv, Scalar(60, 40, 40), Scalar(150, 255, 255), mask);
      //mask is a black and white image, with white where there was blue
      //imshow("Mask", mask);
      //Edge detection
      Mat edges;
      Canny(mask, edges, 200, 400);
      //imshow("Edges", edges);

      return edges;
}

Mat croppedImage(Mat image) {
      //Get size of image
      int rows = image.rows;
      int cols = image.cols;
      Point p1(0, 0);
      Point p2(cols, rows / 2);
      rectangle(image, p1, p2, Scalar(0, 0, 0), -1, LINE_8);

      return image;
}

vector<Vec4i> detectSegments(Mat image) {
```

```
      int rho = 1;      //Distance precision in num pixels

      //Angular precision
      float angle = CV_PI / 180;    //1 degree in radians

      //Threshold for something to be considered a segment
      int numVotes = 10;

      //Min num pixels for it to be a line
      int minLineLength = 8;

      //maximum pixels that segments can be separated by and still be the same segment
      int maxLineGap = 4;

      vector<Vec4i> segs;
      HoughLinesP(image, segs, rho, angle, numVotes, minLineLength, maxLineGap = 4);

      return segs;
}
```

pwmgen.c (PWM driver for Petalinux kernel space)

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/uaccess.h> /* Needed for copy_from_user */
#include <asm/io.h> /* Needed for IO Read/Write Functions */
#include <linux/proc_fs.h> /* Needed for Proc File System Functions */
#include <linux/seq_file.h> /* Needed for Sequence File Operations */
#include <linux/platform_device.h> /* Needed for Platform Driver Functions */
#include<linux/slab.h>
#include <linux/of.h>


/* Define Driver Name */
#define DRIVER_NAME "pwmgen"


unsigned long *base_addr; /* Vitual Base Address */
struct resource *res; /* Device Resource Structure */
unsigned long remap_size; /* Device Memory Size */


/* Write operation for /proc/pwmgen
* --------------------------------
* When user cat a string to /proc/pwmgen file, the string will be stored in
* const char __user *buf. This function will copy the string from user
* space into kernel space, and change it to an unsigned long value.
* It will then write the value to the register of pwmgen controller,
```

```
* and turn on the corresponding LEDs eventually.
*/


static ssize_t proc_pwmgen_write(struct file *file, const char __user * buf,
 size_t count, loff_t * ppos)
 {
 char pwmgen_phrase[16];
 u32 pwmgen_value;


 if (count < 11) {
 if (copy_from_user(pwmgen_phrase, buf, count))
 return -EFAULT;


 pwmgen_phrase[count] = '\0';
 }


 pwmgen_value = simple_strtoul(pwmgen_phrase, NULL, 0);
 wmb();
 iowrite32(pwmgen_value, base_addr);
 return count;
 }


 /* Callback function when opening file /proc/pwmgen
 * ----------------------------------------------------
 * Read the register value of pwmgen controller, print the value to
 * the sequence file struct seq_file *p. In file open operation for /proc/pwmgen
 * this callback function will be called first to fill up the seq_file,
 * and seq_read function will print whatever in seq_file to the terminal.
 */

 static int proc_pwmgen_show(struct seq_file *p, void *v)
 {
 u32 pwmgen_value;
 pwmgen_value = ioread32(base_addr);
 seq_printf(p, "0x%x", pwmgen_value);
 return 0;
 }




/* Open function for /proc/pwmgen
 * ---------------------------------
 * When user want to read /proc/pwmgen (i.e. cat /proc/pwmgen), the open function
 * will be called first. In the open function, a seq_file will be prepared and the
```

```
* status of pwmgen will be filled into the seq_file by proc_pwmgen_show function.
*/
static int proc_pwmgen_open(struct inode *inode, struct file *file)
{
unsigned int size = 16;
char *buf;
struct seq_file *m;
int res;


buf = (char *)kmalloc(size * sizeof(char), GFP_KERNEL);
if (!buf)
return -ENOMEM;


res = single_open(file, proc_pwmgen_show, NULL);


if (!res) {
m = file->private_data;
m->buf = buf;
m->size = size;
} else {
kfree(buf);
}

return res;
}

/* File Operations for /proc/pwmgen */
static const struct file_operations proc_pwmgen_operations = {
.open = proc_pwmgen_open,
.read = seq_read,
.write = proc_pwmgen_write,
.llseek = seq_lseek,
.release = single_release
};


/* Shutdown function for pwmgen
* ---------------------------------
* Before pwmgen shutdown, turn-off all the leds
*/
static void pwmgen_shutdown(struct platform_device *pdev)
{
iowrite32(0, base_addr);
}
```

```
/* Remove function for pwmgen
* --------------------------------
* When pwmgen module is removed, turn off all the leds first,
* release virtual address and the memory region requested.

*/
static int pwmgen_remove(struct platform_device *pdev)
{
pwmgen_shutdown(pdev);


/* Remove /proc/pwmgen entry */
remove_proc_entry(DRIVER_NAME, NULL);


/* Release mapped virtual address */
iounmap(base_addr);


/* Release the region */
release_mem_region(res->start, remap_size);


return 0;
}


/* Device Probe function for pwmgen

* ----------------------------------
* Get the resource structure from the information in device tree.
* request the memory regioon needed for the controller, and map it into
* kernel virtual memory space. Create an entry under /proc file system
* and register file operations for that entry.
*/
static int pwmgen_probe(struct platform_device *pdev)

{
struct proc_dir_entry *pwmgen_proc_entry;
int ret = 0;


res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
if (!res) {
dev_err(&pdev->dev, "No memory resource\n");
return -ENODEV;
}
```

```
remap_size = res->end - res->start + 1;
if (!request_mem_region(res->start, remap_size, pdev->name)) {
dev_err(&pdev->dev, "Cannot request IO\n");
return -ENXIO;
}


base_addr = ioremap(res->start, remap_size);
if (base_addr == NULL) {
dev_err(&pdev->dev, "Couldn't ioremap memory at 0x%08lx\n",
(unsigned long)res->start);
ret = -ENOMEM;
goto err_release_region;
}




pwmgen_proc_entry = proc_create(DRIVER_NAME, 0, NULL,
&proc_pwmgen_operations);
if (pwmgen_proc_entry == NULL) {
dev_err(&pdev->dev, "Couldn't create proc entry\n");
ret = -ENOMEM;
goto err_create_proc_entry;
}


printk(KERN_INFO DRIVER_NAME " probed at VA 0x%08lx\n",
(unsigned long) base_addr);


return 0;


err_create_proc_entry:
iounmap(base_addr);
err_release_region:
release_mem_region(res->start, remap_size);


return ret;
}


/* device match table to match with device node in device tree */
static const struct of_device_id pwmgen_of_match[] = {
{.compatible = "xlnx,pwm-gen-1.0"},
{},

};
```

```
MODULE_DEVICE_TABLE(of, pwmgen_of_match);


/* platform driver structure for pwmgen driver */
static struct platform_driver pwmgen_driver = {
.driver = {
.name = DRIVER_NAME,
.owner = THIS_MODULE,
.of_match_table = pwmgen_of_match},
.probe = pwmgen_probe,
.remove = pwmgen_remove,
.shutdown = pwmgen_shutdown
};


/* Register pwmgen platform driver */
module_platform_driver(pwmgen_driver);


/* Module Infomations */
MODULE_AUTHOR("Digilent, Inc.");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION(DRIVER_NAME ": pwmgen driver (Simple Version)");
MODULE_ALIAS(DRIVER_NAME);
```

Build Scripts

buildpetalinux.sh (used to automate PetaLInux build process)

```
PETA_ROOT="../petalinux"
BITSTREAM="../hw_dep/system_wrapper.bit"
HDF_PATH="../hw_dep"

cd $PETA_ROOT
petalinux-config --get-hw-description=${HDF_PATH}
#petalinux-config -c rootfs
petalinux-build
petalinux-package --boot --fpga $BITSTREAM --u-boot --force
```

copyfstosd.sh (used to automate transferring new PetaLinux builds to sd card)

```
PETA_ROOT="../petalinux"
BOOT_PART="/media/ramsey/BOOT"
ROOT_PART="/media/ramsey/ROOTFS"
BUILD_INC="../build_include"
```

```
cd $PETA_ROOT
sudo rm ${BOOT_PART}/BOOT.BIN
sudo rm ${BOOT_PART}/image.ub
sudo cp images/linux/BOOT.BIN $BOOT_PART
sudo cp images/linux/image.ub $BOOT_PART
umount /mnt
sudo mount images/linux/rootfs.ext4 /mnt -o loop
sudo rm -r ${ROOT_PART}/*
sudo cp -r /mnt/* $ROOT_PART
sudo cp -r ${BUILD_INC}/* ${ROOT_PART}/home
sudo umount /mnt
```

userprogrambuild.sh (used to cross compile programs for the ARM processor)

```
FILE_NAME="socarmain"
H_FILE_NAME="charvideo_ioctl.h"

arm-linux-gnueabihf-g++ ../src/${FILE_NAME}.cpp ../src/${H_FILE_NAME} -std=c++11 -o
../build_include/${FILE_NAME} -I../lib/opencv/build/install/include/opencv4 -
L../lib/opencv/build/install/lib -lopencv_calib3d -lopencv_imgproc -lopencv_core -
lopencv_ml -lopencv_features2d -lopencv_objdetect -lopencv_flann -lopencv_video -
lopencv_highgui -lopencv_imgcodecs -lpthread
#arm-linux-gnueabihf-g++ ../src/${FILE_NAME}.cpp -std=c++11 -o $FILE_NAME $(pkg-config -
-cflags --libs opencv)
```

settup.sh (used to set up new PetaLinux environment)

```
mkdir /usr/local
cp -r opencv/* /usr/local
echo "/usr/local/lib" >> /etc/ld.so.conf
ldconfig
chmod +x socarmain
```

PROBLEMS ENCOUNTERED

PCam5 Integration

The PCam5 was unable to work on our board as the Linux driver that we wrote/adapted was unable to link to the hardware on the Zybo board. This was quite a confusing issue as we were able to accomplish the same for the PWM module. Apart from this, we were able to fully use the PCam5 in a bare metal implementation which verified that our hardware was not the issue. We made sure to check the compatible name (what the device tree uses to label and identify hardware to link it to drivers) and it was consistent between the hardware and drivers, which should ensure that it would work. This means that the hardware (delivering the images from the camera) as well as the software (processing the images) were both functional, but the linking between the two was at fault.

If this issue were to be resolved, we fully believe the entire project could be up and running. With an image stream from the PCam5 to the main program, we would have correct motion control of the car, which would complete our project.

TESTING

PCam5 Testing

Since the PCam5 is created by Digilent, they provided the IPs and hardware design to get the camera working. There were some changes needed to be made to get the design to be compatible with our board and Vivado version, but these changes were implemented smoothly. By using the SDK, we could run a program to capture a video stream from the camera and output it through HDMI to a monitor. The video of this demo has been included along with this submission. This testing assured us that the PCam5 would be compatible with our design, and so we built upon the provided hardware design to add the other hardware IPs we needed.

Motor and Steering Testing

In order to gain access to the motor and steering servo of the car, we had to disconnect the transmitter from both components. Since the transmitter was used to send out the signals from the remote control, we first probed these signals in action using a DAD. Doing so allowed us to see the 60 Hz frequency the remote was operating at, and also the limits of the duty cycles that would control the motor and servo. Afterwards, we tested the operation of these two components without the remote control by creating square wave signals from the DAD. This allowed us to compose a guide that explains the specific duty cycle values needed to turn the car left or right by certain degrees, and what duty cycle corresponded to a certain speed. This guide was then used to create the steering portion of the main program, which determines the output signal values needed to control the car’s movement.

Image Processing Program Testing

The operation of this program is already described in the Software Explanation section. The testing was simply done by passing in images that show a lane created with two blue pieces of tape. One image had a straight path, and the other had a curve to indicate turning. The result of the program would show a line in the direction of the lanes, the direction that the car would need to follow in order to stay between the lines. Slight corrections were made to the program until the direction line was in the correct direction we would have expected the car to follow.

Main PetaLinux Program Testing

In order to interface with the hardware on the Zybo board, we had to write/adapt drivers to communicate with the hardware. This was done in C to create modules that were sideloaded into the kernel for communicating between the hardware and Userspace applications. In addition, we

needed to cross compile the C++ application and the OpenCV libraries. The testing process for both of these consisted of adding changes to the kernel modules and user space applications, compiling the PetaLinux instance, putting the compiled user applications and kernel modules on the SD Card (as well as the PetaLinux image), and then running it on the Zybo board. As writing and testing the software was an iterative process, this had to be repeated multiple times.

MEMBER CONTRIBUTIONS

Nick Sileo –

- Contributed to planning for how this project would be started and handled, and what online sources we should use as guides.
- Translated an image processing program from Python to C++.
- Setup the image processing C++ program to take in an image and process it correctly to output a steering direction in degrees.
- The C++ program also calculated the necessary duty cycle to achieve this angle.
- Wrote the software explanation section of the documentation.

David Carvajal –

- Contributed to planning for how this project would be started and handled, and what online sources we should use as guides.
- Purchased the car and all additional components that have so far been required to operate the final project. *(the purchasing of these items was entirely voluntary so that David would be able to work on the hardware from home, so it should not be taken as a contribution over the work of other members)*
- 3D printed the parts needed to attach the additional components to the car.
- Tested the PCam5 (works), then started the main Vivado Project that will be used for the hardware design, and added the PCam5 modules to it, which operate correctly.
- Probed the motors and servos of the car to determine the PWM frequency and duty cycle to control those components. Created a guide from this that shows the exact duty cycles that would correspond to steering angles, which was used by Nick to implement the last steering conversion step of the C++ program.
- Assisted Ramsey with the PWM IP integration to the Vivado Project.
- Setup the documentation and created the SOC Architecture diagram. Also added to the testing and hardware explanation sections.

Michael Kim-

- Contributed to planning for how this project would be started and handled, and what online sources we should use as guides.
- Modified the PWM IPs from the previous assignments to output to pins rather than LEDs, allowing to output the signals needed to control the car.
- Added to the hardware explanation section of the documentation.

Ramsey Hamed-

- Created portion of socarmain.cpp that handled handshaking between the camera drivers, vision processing code, and PWM drivers.
- Created and managed the PetaLinux project that used the Vivado hardware design, C++ application, and drivers to control the car hardware.
- Adapted PWM and VDMA drivers for interfacing with PWM hardware and camera module.
- Figured out how to cross-compile the OpenCV libraries as well as well as the C++ application for PetaLinux and the ARM processor so that the C++ program can run on the Zybo board.
- Wrote multiple bash scripts for automating the development process (building PetaLinux, copying filesystem to SD card, cross-compiling user-space applications, setting up PetaLinux environment).
- Recorded the demo videos to show the different subsystems functioning properly.
- Documented code as well as problems encountered.

Paul Samuel Kommula-

- Attempted to integrate the PCam5 drivers to the image processing program in PetaLinux but errors impeded this.

ADDITIONAL NOTES

Motor Speed

Off/Base = 9 %

Max Speed = 12%

Period = 16.6 ms

freq = 60 Hz

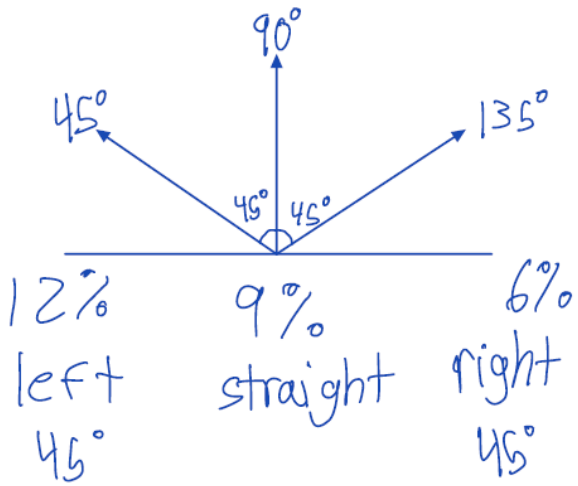
9.6% is a slow speed, good to start

Direction

Min = 6 %

Max = 12 %

Base = 9 %



Notes on how the motor speed and steering work.