LAFTR Raspberry Pi Zero Development

Author Scott Hunter
Summer 2017

Contents

Operating System

There were several OS options researched for installation on the Raspberry Pi0. The full version of Raspbian Jessie gives a graphical interface along with a fully functioning OS, and is what was used in development thus far. There is a lite version which supposedly draws about 20 mA less at idle. It has only has a CLI. After the code development stage, a RISC OS (reduced instruction set computing) or thin server should be installed for use. A version of RISC OS by Acorn is used on the beagleboard, and another version is available for Raspberry Pi; the one with only a terminal is called RISC OS Pico. Another option is to use a stripped-down version of linux, found here: <http://tech.munts.com/>. This OS is designed for microcontrollers, and can be used as a thin server. This option could be especially useful on the Raspberry Pi 0 w, so that the Pi could be SSH'ed into before flight or for debugging.

Libraries

Libraries that need to be included have the FIFO interface class of the Arduino Code work in Raspberry Pi:

<stdio.h>
<stdlib.h>
<Adafruit_GPS.h>
<string>
<wiringPi.h>

The only differences are <string> and <wiringPi.h>. The string library is used to mirror string operations native to the Arduino language. <wiringPi.h> was chosen because it follows the same syntax and has similar keywords to Arduino code. To see changes made, see Appendix D.

Even with the similarities, some functions need to be adjusted. For example, the <SD.h> and <SPI.h> libraries need not be included, since the SD card is on board the Raspberry Pi. This inherently requires some changes to the program, which have not all been made. Any libraries required to interface through UART to the GPS will also need to be included, possibly <unistd.h>, <fcntl.h>, and <termios.h>.

For future development, the Arduino Code is built out of C++ code. A quick internet search did not yield a straight answer, but it might be possible that the original Arduino program could be converted to C++ code and then adjusted to run on the Raspberry Pi. A library to look into when pursuing this option is <Arduino.h> .

There are other GPIO libraries for Raspberry Pi which could be used. WiringPi was chosen because of its similarity to Arduino. The library <bcm2385.h> could be used instead, it is supposedly a bit faster.

GPIO

Tests were done to determine read and write limits of the Raspberry Pi 0 w through the GPIO pins.

Write Speed

The write speed test was done by creating an infinite loop which would toggle a single GPIO pin from low to high with no wait time between. The output was recorded by a Tektronix MDO3024 oscilloscope. The recorded waveform is shown below, in Figure 1.

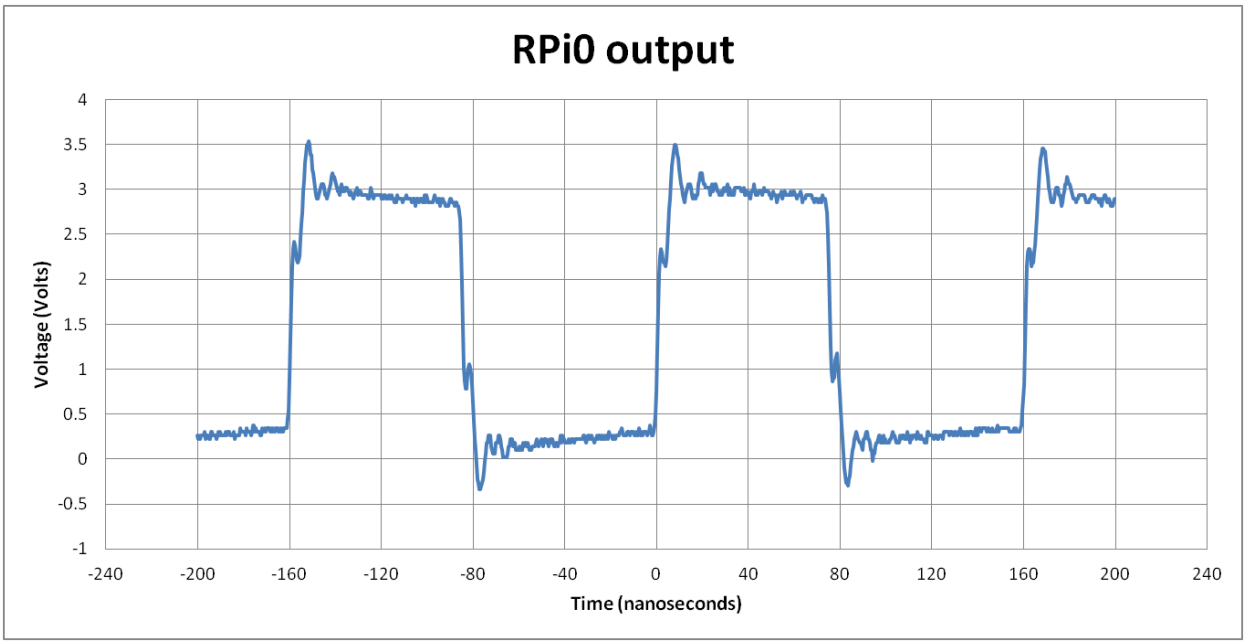

Figure 1 - RPi0 Output Test

The frequency of the square wave generated by the Raspberry Pi is about 6.2 MHz. Since the square wave was generated by one GPIO pin toggling twice per period, the frequency can be doubled to find the write speed of the Raspberry Pi0, about 12.4 MHz. Though the Raspberry Pi will primarily function as a reading device in the LAFTR payload, the write speed provides an upper limit to read speed.

Read Speed

The read speed test was done by setting a pin to read mode and applying a pull down resistor. This was connected to a Tektronix MDO3024 which output a square wave. The Raspberry Pi then streamed whatever it read straight to a text file.

A long time was spent on get the program to stop reading and close the file at a keystroke. Threading was attempted, using the library <ncurses.h>, allowing use of the command 'getch' in the Linux OS environment. Either of these could probably be made to work, but so no more

unnecessary time was spent, the program was finally made to just have the Raspberry Pi read for a set number of data points and then close.

This program was run for square waves of increasing frequency, until the recorded data seemed to start skipping pulses. The fastest square wave this particular method could accurately read was about 5 kHz, which would correspond to the Raspberry Pi reading about 10 kbps. The data stream shown below in Figures 2 and 3, shows the recording of the 5 kHz square wave, along with the recording of a 6 kHz wave, which started to lose data integrity. This can be seen by the unequal pattern of the data points.

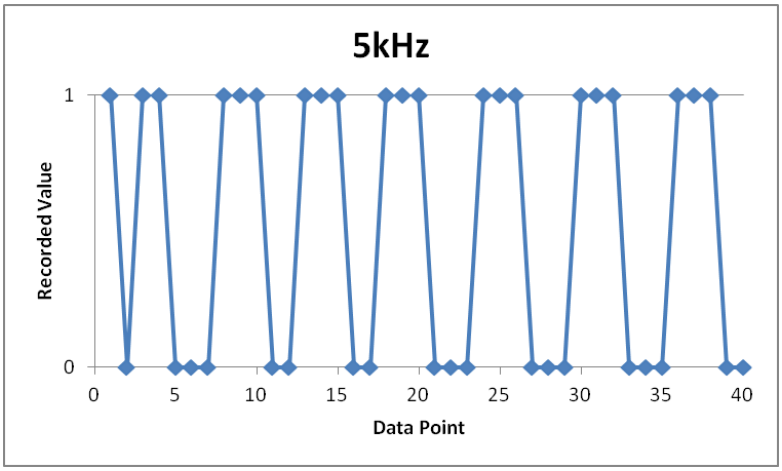

Figure 2

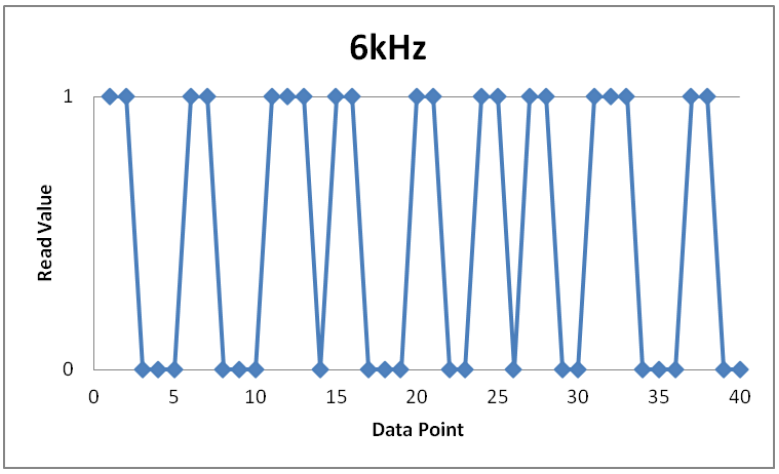

Figure 3

This may not have been the most efficient method of reading, and so the number may be lower than the true maximum read speed, but it does serve as a ballpark estimate. A more efficient

method may be to populate an array for a set number of points, and then save to the SD card. The major problem with this method is synchronizing the Raspberry Pi with the oscilloscope. If this could be done, then this would be an effective test program. Instead, the FIFO functionality described below proved more effective, since the timing of the data streamed to the Raspberry Pi was controlled by the Pi itself.

Data Logger Development

The two functions handled by the Arduino Due which were converted to be used on the Raspberry Pi are discussed below.

FIFO Functionality

This function was originally developed to simply prove that the Raspberry Pi could read from the FPGA. After that, it was used as a speed test, but keeping the design for the read test. As a result, some of the functionality which would be in the full program, like checking the MoreData line, is not part of the current program. This would need to be added back in to have a true speed test of the Raspberry Pi which could be compared to the Arduino Due. When the program (v 1.1.2) was run, reading 1048 words from the FPGA, the Raspberry Pi took about 21 ms to read the data and save it to an SD card. The times from each trial are contained below in Figures 4 and 5, below. The Arduino Due ran the same test and took about 800 ms. This means the Raspberry Pi is about 40 times faster, and is a viable option for the payload, reading at a speed of about 2.5 MHz.

Read Only	
Trial	**Time (ms)**
1	8.945
2	15.514
3	8.890
4	15.596
5	8.812
6	15.812
7	8.934
8	15.836
9	8.878
10	8.889
11	8.875
12	8.977
13	8.786
average	**10.980**

Figure 4 - FIFO Test read only to buffer. 13 tests were run because of the strange oscillating of values between 8 and 15 ms. This pattern broke on the tenth trial, so more were run to determine if it was an outlier. It could be a function of time, a result of how memory is deallocated by the Raspberry Pi OS.

Read & Save to SD	
Trial	**Time (ms)**
1	18.287
2	21.780
3	19.632
4	18.382
5	26.405
6	20.573
7	23.186
8	23.098
9	21.704
10	22.073
average	**21.512**

Figure 5 - FIFO Test read and save to SD card. The Arduino Due's average run time was 800 ms per kiloWord.

Below are extended development notes for the program. Appendix A contains the wiring diagram for the control bus and the data bus from the Raspberry Pi to the FPGA. Apendix XXX contains the code for FIFOTest.cpp v 1.1.2, the most current version.

v 1.0
The Arduino code was changed to work with the Raspberry Pi.

Bitwise and a char array was used rather than strings. Getting the string functionality to work similarly to the Arduino program was difficult. Using strings was thought to be inefficient since a string array is an array of pointers to string arrays. Instead of having all of the pointers be passed around and a lot of extra memory used for each word read from the FPGA, bitwise operations were used to store the data. This generates a binary file in a .txt file.

v 1.0.1
Timing was placed in its own class, called speed. This class can be used to time any part of a program by instantiating the class, calling the Start and Finish functions. The end function outputs the time the part of the program took between Start and Finish to the screen, with nanosecond precision. It uses CLOCK_MONOTONIC_RAW to time, since this clock called is not subject to NTP adjustments and cannot be inadvertently changed by the user. These keywords are specific to Linux. The library <time.h> must be included for this functionality.

v 1.1
Tried to use an unsigned short to match the 16-bit line, which would have eliminated a struct and a loop. Had trouble printing in such a way that retained the binary, and so abandoned method, returning to char method.

v 1.1.1
The indices of loops were changed to read data in the correct order.

v 1.1.2
Changed the program to count up everywhere, making [0] MSB and [x] LSB, in keeping with the FIFO style.

GPS Functionality

Ran out of time to get this fully working. Started in on changing the SDCardHandler function to work on the Raspberry Pi. Had the thought that all of the setup functions could be made into constructors. Pins intended for interfacing with the GPS are marked in the pin diagram in Appendix A.

Power Testing

Tests were run to determine the power draw of the Raspberry Pi 0.

With Resistor

The Raspberry Pi was powered through the GPIO pins, one for +5 VDC output (physical pin 2) and the other for ground (pin 39). A 30 mΩ resistor was placed in series with the Raspberry Pi on the ground side, and a Tektronix MDO3024 oscilloscope hooked up to read voltage across the resistor. The setup is shown below, in Figure 6.

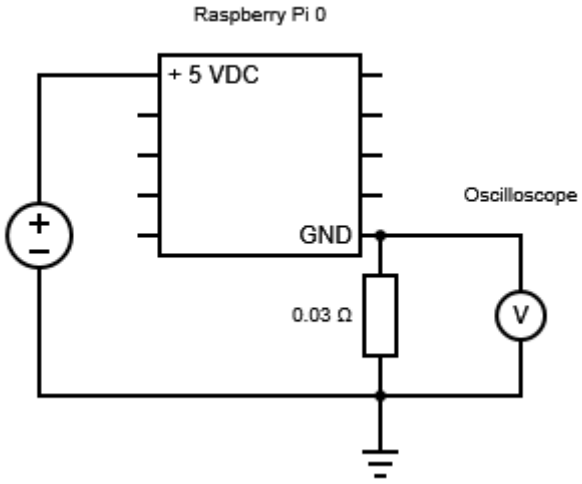

Figure 6

This setup was used because, had the resistor been on the high side of the Raspberry Pi, the current may have found ground through the oscilloscope rather than through the microcontroller, which would have created a short through the oscilloscope. This setup forces ground to be at the end of the circuitry, and protects equipment.

The voltage read by the oscilloscope was multiplied by 1000 and divided by 0.03 to give the current draw, in miliAmps, of the Raspberry Pi using Ohm's Law, V = IR. Readings were taken for 4 different tests: powering the Raspberry Pi on, plugging in a USB dongle an external flash drive connected, SSHing into the Raspberry Pi via WiFi, and transferring a 43 MB file onto the Raspberry Pi SD card from the USB flash drive. The graphs of the current draw of the Raspberry Pi 0 is contained below in Figures 7 through 10. The black line in all of these is a moving average of 10 points. The command to move the file from the flash drive to the Raspberry Pi SD card was given via SSH. The HDMI port circuitry was turned off for all of these measurements to conserve power.

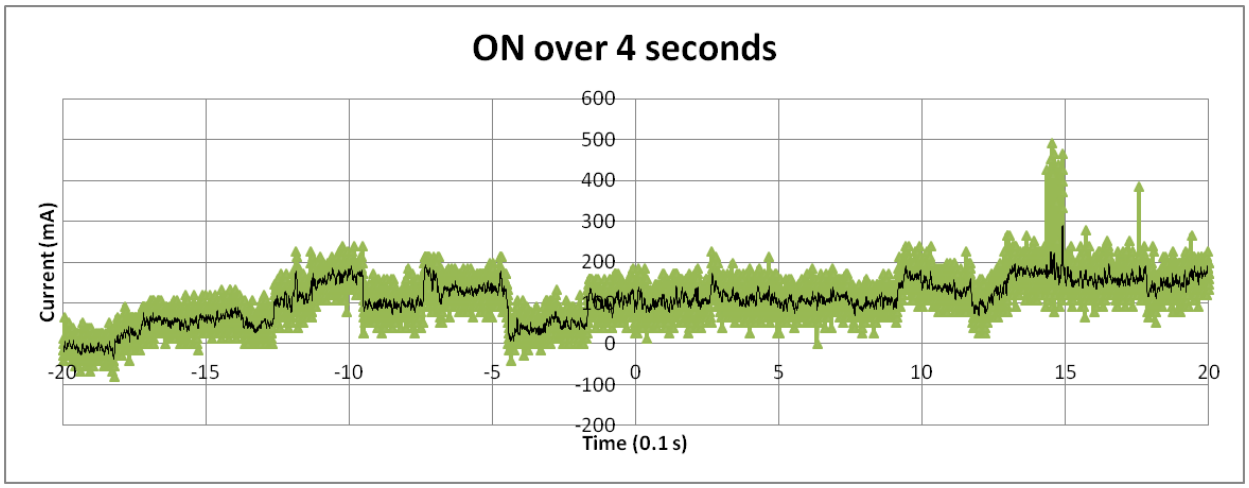

Figure 7 - 4 second recording of the boot sequence. Notice the large current spike of about 500 mA at t = +1.5 s.

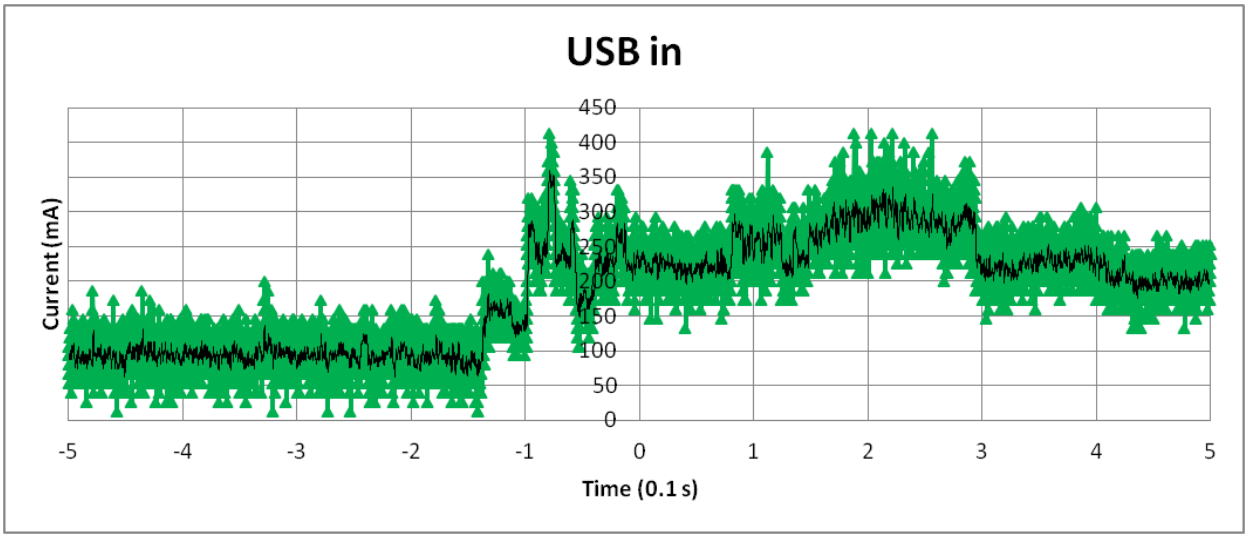

Figure 8 - this figure also provides about 0.5 seconds of the Raspberry Pi 0 at idle, from t = -0.5 to t = -0.15

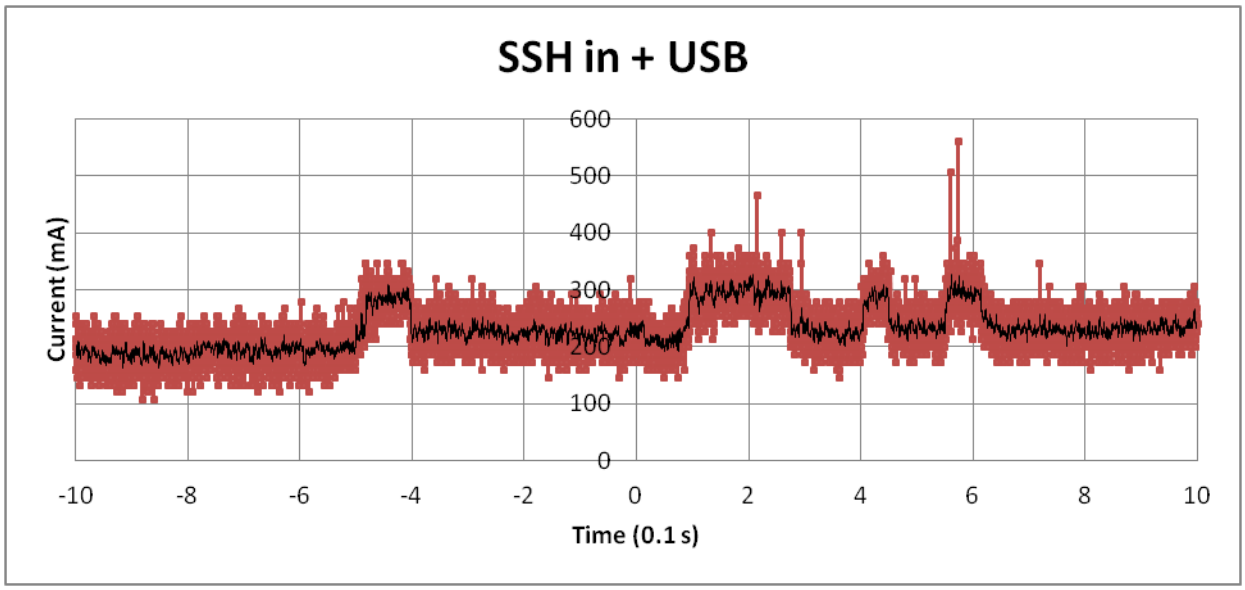

Figure 9

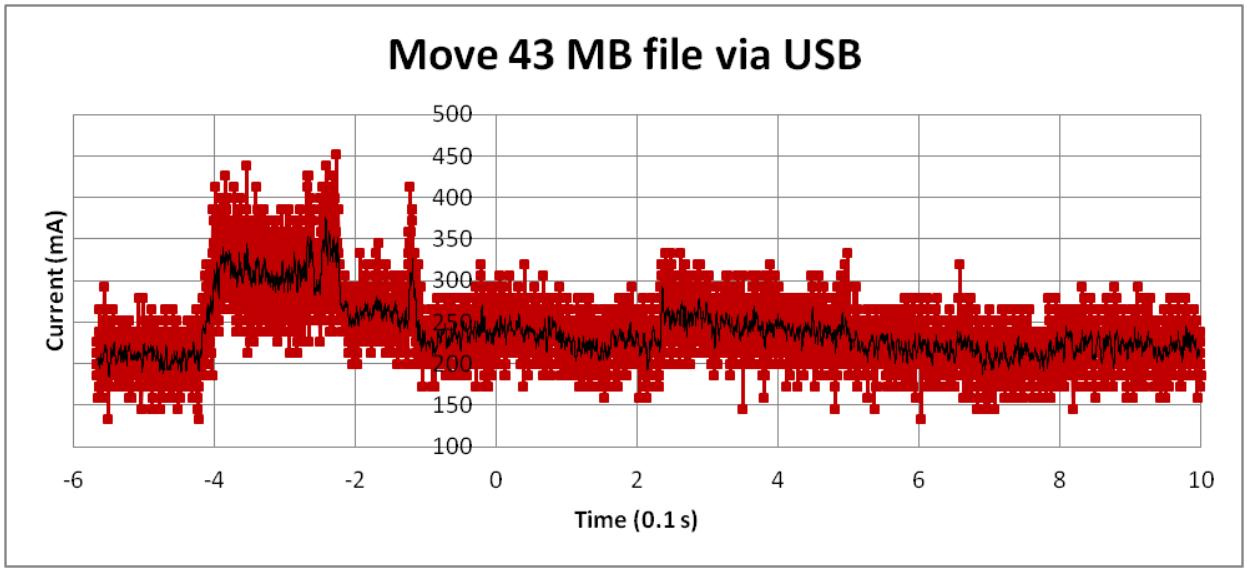

Figure 10 - notice the y-axis starts at 100 mA to better see the data. This test was done with the USB dongle in, with the command to move the file given via SSH.

Figure 8 shows the idle current draw of the Raspberry Pi 0 to be about 100 mA. A blog post made by Jeff Geerling (website) claimed the Raspberry Pi 0 only drew 80 mA at idle. This was using Raspbian Lite, which would give good reason to look into changing from Raspbian with PIXEL, to another, smaller OS. Geerling also turned off the HDMI circuitry and ACT LED light.

Below, these operations are explained. It may be possible to disable the USB circuitry as well, it should especially done if the Pi 0 w is used because one could just SSH for a system check.

Important: when power the Raspberry Pi 0 through the GPIO pins, all protective circuitry provided by the micro USB port is bypassed. As long as the source provides a steady voltage, this should not be a problem.

Turning off HDMI Port

This should drop about 20 mA of current draw.

Edit the file: /etc/rc.local

Add to disable HDMI:

```
/usr/bin/tvservice -o          (change -o to -p to reenable)
```

Turning off ACT LED

This should drop about 5 mA of current draw, though you lose your LED check-it-is-on light.

Edit the file: /boot/config.txt

Add to the end:

```
dtparam=act_led_trigger=none
dtparam=act_led_activelow=on
```

bootText Program/BASH script

A program called BootText was written which would write the string "It booted." to a .txt file called BootText, followed by a time stamp. This was initially done to see if the Raspberry Pi fully turned on when powered through GPIO pins. This method was not used because recording the voltage across a small resistor in time with an oscilloscope provided much more data about the boot sequence. This is important information, though, because the final DataLogger program should be attached to the boot sequence so that it begins to run automatically as soon as the system is powered on.

There are 5 options for doing this:

- rc.local
- .bashrc
- init.d tab
- systemd
- chrontab

systemd was used because it most likely runs last in the boot sequence, when most or all of the Raspberry Pi's resources and libraries (like wiringPi) are available. A simple BASH script was

used to run the program. It's code is shown below, as well as in Appendix B. For sample code to run the other alternatives, see this website: www.dexterindustries.com.

```
[Unit]
Description = Runs bootText program
After = multi-user.target

[Service]
Type = idle
ExecStart = /home/pi/bootText

[Install]
WantedBy = multi-user.target
```

Then type into the command line:

```
~$ sudo systemctl daemon-reload
~$ sudo systemctl enable bootText.service
```

Miscellaneous

Starting Up a Raspberry Pi 0 w

1. Load NOOBS file from the Raspberry Pi webesite onto a FAT 32 formatted micro SD card (unzipped file).
2. Insert into Raspberry Pi
3. Select which OS you want to install
4. As long as there is an internet connection, the OS will set itself up automatically
5. Makes sure to change the keyboard settings to USA

To do this without NOOBS, use an OS imager, like Win32 Disk Imager.

SSH

Default username: pi
Default password: raspberry

How to SSH using USB:

http://www.makeuseof.com/tag/directly-connect-raspberry-pi-without-internet/

Edit: config.txt
add: dtoverlay=dwc2 to end of file

Edit: cmdline.txt
add: modules-load=dwc2,g_ether after 'rootwait', in the same line, separated by spaces

use the port labeled USB for this, and use raspberrypi.local in the SSH software.

SSH Using WiFi:

Best bet is to set the IP address of the Raspberry Pi to a static number. Many sources online have instructions for doing this.

Thus far, the software MobaXterm was used to SSH into the Pi over WiFi. It has a network search tool, which scanned a range of IP addresses (it helps to have an idea of the Pi's IP address first). When the Pi's IP address shows up, click it to start a new session.

'Baking' A USB Port For Ethernet

This is not as scary as it sounds, it is purely software. There is a project called the "Pi - Kitchen" and using it, by analogy, "bakes" your version of an operating system to suit your needs. In this case, it configures a USB port to act like an Ethernet connection for SSH. Instructions are at the web address below.

URL: https://pihw.wordpress.com/guides/pi-kitchen/startbaking/

Appendix

Appendix A - Rasberry Pi 0 pinout

Rpi_0			
Label	Physical Pin		Label
3.3 V	1	2	5 V
F	3	4	5 V
E	5	6	GND
D	7	8	TxD
GND	9	10	RxD
C	11	12	open
B	13	14	GND
A	15	16	morData
3.3 V	17	18	Full
9	19	20	GND
8	21	22	RS0
7	23	24	RS1
GND	25	26	RS2
6	27	28	Trig
5	29	30	GND
4	31	32	rSTn
3	33	34	GND
2	35	36	rAdd++
1	37	38	PPS
GND	39	40	0

Opal Kelly			
Label	Physical Pin		Label
GND	GND	GND	GND
VREF 5	VREF 5	VREF 2	VREF 2
E	MC2-A1	MC2-A0	F
C	MC2-Bp 1	MC2-Bp0	D
A	MC2-Bn 1	MC2-Bn0	B
8	MC2-Bp3	Bp2	9
6	MC2-Bn3	Bn2	7
4	MC2-Bp5	Bp4	5
2	MC2-Bn5	Bn4	3
GND	GND	GND	GND
0	MC2-A3	MC2-A2	1
Full	MC2-Bp7	MC2-Bp6	morData
RS1	MC2-Bn7	MC2-Bn6	RS0
Trig	MC2-Bp9	MC2-Bp8	RS2
rSTn	MC2-Bn9	MC2-Bn8	rAdd++
	MC2-Bp11	MC2-Bp10	PPS
	MC2-Bn11	MC2-Bn10	
	MC2-Bp13	MC2-Bp12	
	MC2-Bn13	MC2-Bn12	
	GND	GND	

Appendix B - BootText.cpp

C++ code:

```
#include <iostream>
#include <fstream>
#include <ctime>

using namespace std;

// Get current date/time, format is YYYY-MM-DD.HH:mm:ss
const std::string currentDateTime() {
    time_t     now = time(0);
    struct tm  tstruct;
    char       buf[80];
    tstruct = *localtime(&now);
    // Visit http://en.cppreference.com/w/cpp/chrono/c/strftime
    // for more information about date/time format
    strftime(buf, sizeof(buf), "%Y-%m-%d.%X", &tstruct);

    return buf;
}

int main()
{
        //cout<< "\nBoot Script is running.\n";
        ofstream bootFile;
        bootFile.open("/home/pi/bootFile.txt", ios::app);
        bootFile<< "\nIt booted.  ";
        bootFile<< currentDateTime() << endl;

        //time_t t = time(0);   // get time now
    //struct tm * now = localtime( & t );
    //bootFile << (now->tm_year + 1900) << '-'
         //<< (now->tm_mon + 1) << '-'
         //<<  now->tm_mday << '-'
         //<< endl;

        bootFile.close();

        return 0;
}
```

Bash Script:

```
[Unit]
Description = Runs bootText program
After = multi-user.target

[Service]
Type = idle
ExecStart = /home/pi/bootText

[Install]
WantedBy = multi-user.target
```

Then type into the command line:

~$ sudo systemctl daemon-reload
~$ sudo systemctl enable bootText.service

Appendix C - Timing Function

```
class Speed {
private:
	struct timespec start, finish;
public:
	double elapsed;

	void Start() {			//get time at start
		clock_gettime(CLOCK_MONOTONIC_RAW, &start);
		printf("\ntime start");
	}

	void Finish() {			//get time at end and output time
		clock_gettime(CLOCK_MONOTONIC_RAW, &finish);
		elapsed = (finish.tv_sec - start.tv_sec);
		elapsed += (finish.tv_nsec - start.tv_nsec) / 1000000000.0;
		printf("The program took %f seconds", elapsed);
		printf("\nfinish time\n");
	}
};
```

Appendix D - FIFOText.cpp

```
/*Light and Fast Terestrial Gamma-Ray Recorder
 * FIFO interface for testing with Raspberry Pi 0 w
 * Montana State University
 * author Jonathan Johnson
 * edited by Scott Hunter
 * created 13 July 2017
 * version 1.0
 *
 * v 1.0
 * - Changed Arduino code to work with the Raspberry Pi
 * - Used bitwise and a char array rather than strings
 *
 * v 1.0.1
 * -put timing in its own class
 *
 * v 1.1
 * -tried to use an unsigned short to match the 16-bit line
 * -could not print into binary
 * -abandoned method
 *
 * v 1.1.1
 * -changed indicies to read correctly
 *
 * v 1.1.2
 * -counts up everywhere, making [0] MSB and [x] LSB
```

```
Note: a FIFO word is 48 bits, or 6 bytes
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <wiringPi.h>
using namespace std;
const int BUS[16] = {8,9,7,0,2,3,12,13,14,30,21,22,23,24,25,29};     // pins for bus
line, left side of pi, starting from 3, plus pin 40, where 40 is LSB
const int AMT_DATA = 32*6;         //fifoWords * bytes


struct busWord {
       char bus[2];
};
struct fifoWord {
       busWord fifo[3];
};

class FIFOInterface {
private:
       int RS0_pin;  //pins, 3 talking to FPGA
       int RS1_pin;
       int RS2_pin;
       int rAddr_pin;       //read address pin, goes high to read next data
       int rSTn_pin; //reset (goes high to reset) allows taking the 48 bits by section.

public:
       FIFOInterface(int rs0, int rs1, int rs2, int rA, int rS) {    //requires pin
numbers as input
              RS0_pin = rs0;
              RS1_pin = rs1;
              RS2_pin = rs2;
              rAddr_pin = rA;
              rSTn_pin = rS;
       }

       /*
       16-bit BUS is integrated in the set-up portion of the function
       */

       void setUp() {
              wiringPiSetup();     //sets up wiringPi
              pinMode(RS0_pin, OUTPUT);
              pinMode(RS1_pin, OUTPUT);
              pinMode(RS2_pin, OUTPUT);
              pinMode(rAddr_pin, OUTPUT);
              pinMode(rSTn_pin, OUTPUT);
              printf("\nfifo COMM");

              // setting up the 16 - bit BUS line starting from digital pin 22 to 37
              //Raspberry Pi pins
              for (int n = 0; n < 16; n++) {
                     pinMode(BUS[n], INPUT);
```

```
                        pullUpDnControl(BUS[n], PUD_DOWN);        //didn't matter PUD_UP
because I think that Arduino only does Pull UP Resistors.
                }
                printf("\nfifo BUS");

        }

        fifoWord FIFODataWord() {
                fifoWord data;

                mostSigWord();
                data.fifo[0] = read16BitBUS();
                midSigWord();
                data.fifo[1] = read16BitBUS();
                leastSigWord();
                data.fifo[2] = read16BitBUS();

                incrementAddress();

                /*printf(" %i", ret.fifo[2].bus[1]);
                printf(" %i", ret.fifo[2].bus[0]);
                printf(" %i", ret.fifo[1].bus[1]);
                printf(" %i", ret.fifo[1].bus[0]);
                printf(" %i", ret.fifo[0].bus[1]);
                printf(" %i", ret.fifo[0].bus[0]);*/

                return data;
        }

        busWord read16BitBUS() {
                busWord ret;
                ret.bus[0] = 0;
                ret.bus[1] = 0;


                //8 bits at a time, two char because int is less consistent between
machines
                for (int x = 0; x <8; x++) {
                        //Upper 8 bits
                        if (digitalRead(BUS[x]) != LOW) {
                                ret.bus[0] = (ret.bus[0] << 1) | 0x01;
                        } else {
                                ret.bus[0] = (ret.bus[0] << 1) & 0xFE;
                        }
                        //Lower 8 bits
                        if (digitalRead(BUS[x+8]) != LOW) {
                                ret.bus[1] = (ret.bus[1] << 1) | 0x01;
                        } else {
                                ret.bus[1] = (ret.bus[1] << 1) & 0xFE;
                        }
                }
                //printf(" %i", ret.bus[0]);
                //printf(" %i", ret.bus[1]);
                return ret;
        }

        void mostSigWord() {
                digitalWrite(RS0_pin, LOW);
```

```
		digitalWrite(RS1_pin, LOW);
		digitalWrite(RS2_pin, LOW);
	}
	void midSigWord() {
		digitalWrite(RS0_pin, HIGH);
		digitalWrite(RS1_pin, LOW);
		digitalWrite(RS2_pin, LOW);
	}
	void leastSigWord() {
		digitalWrite(RS0_pin, LOW);
		digitalWrite(RS1_pin, HIGH);
		digitalWrite(RS2_pin, LOW);
	}

	void getAddress() {
		digitalWrite(RS0_pin, HIGH);
		digitalWrite(RS1_pin, HIGH);
		digitalWrite(RS2_pin, LOW);
	}

	// Initiate code for the FIFO within the FPGA
	void reset() {
		digitalWrite(rSTn_pin, LOW);
		digitalWrite(rSTn_pin, HIGH);
		printf("\nfifo reset");
	}

	void incrementAddress() {
		digitalWrite(rAddr_pin, LOW);
		digitalWrite(rAddr_pin, HIGH);
		digitalWrite(rAddr_pin, LOW);
	}
};

	//writes file to SD CARD after buffer has filled
	void writeFile(const char* dataArray, int sizeOfData){
		FILE *dataFile = 0;
		//printf("\nwriteFile initialized");

		dataFile = fopen("/home/pi/RAMtest.txt", "wb");
		for (int i = 0; i < sizeOfData; i++) {
			//printf(" %c", dataArray[i]);
			fputc(dataArray[i], dataFile);
		}

		fclose(dataFile);
	}

class Speed {
private:
	struct timespec start, finish;
public:
	double elapsed;

	void Start() {			//get time at start
		clock_gettime(CLOCK_MONOTONIC_RAW, &start);
		printf("\ntime start");
	}
```

```
	void Finish() {			//get time at end and output time
		clock_gettime(CLOCK_MONOTONIC_RAW, &finish);
		elapsed = (finish.tv_sec - start.tv_sec);
		elapsed += (finish.tv_nsec - start.tv_nsec) / 1000000000.0;
		printf("The program took %f seconds", elapsed);
		printf("\nfinish time\n");
	}
};

int main() {
	//sets up a place in the buffer to store information
	char data[AMT_DATA];
	//printf("\narray initialized");


	//sets up fifo
	FIFOInterface fifoInterface(6,10,11,27,26);	//physical pins 22, 24, 26, 36, 32
	fifoInterface.setUp();
	fifoInterface.reset();				// start state of the fifo interface
	//printf("\nfifo initialized");

	//get time at start
	Speed duration;
	duration.Start();

	//read data
	for (int fifoword = 0; fifoword < AMT_DATA/6; fifoword++) {

		fifoWord curWord = fifoInterface.FIFODataWord();

		for (int busline = 0; busline < 3; busline++) {
			for (int byte = 0; byte < 2; byte++) {
				data[fifoword*6 + busline*2 + byte] =
curWord.fifo[busline].bus[byte];
				//printf("%i ", data[fifoword*6 + busline*2 + byte]);
			}
		}
	}
	printf("\nread data");

	//put data in a file to look at later
	writeFile(data, AMT_DATA);
	printf("\nfile written");

	//get time at end and output time
	duration.Finish();


	return 0;
}
```