RASPBERRY PI 3 WITH REAL TIME KERNEL

By Won Yong Ha
Master of Systems Engineering
Cornell University

Advisor By Daniel Lee
Post-Doctoral at Mechanical Engineering
Cornell University

Abstract: An autonomous vehicle is required a tremendously lots of sensors for recognizing objects and making decision like a human. One of the key sensors used in the Cornell Autonomous Systems Lab is Lidar Sensor manufactured by Velodyne. Unlike other sensors such as compact Lidar or Radar, the Lidar Sensor sent much more data: around 3 Mbytes per seconds. Since the data sends via Ethernet port, data will not lose from the wiring. This paper is explaining the difficulty of processing a large amount of data with compact MCUs (Microcontroller Unit) and alternative solution with Raspberry Pi 3. Installing the Real Time Kernel, the Raspberry Pi 3 accepts time data from Timeserver and Lidar Sensor. After accepting those data, Raspberry Pi 3 adds the time according to the Timeserver tick and data and sends to specific assigned address.

Introduction

All the sensors on the autonomous car in the Autonomous Systems Lab are very critical among all the researches process through the car. Each of the sensor has their unique characteristics and importance of research for autonomous driving. Although most of the sensor send all data as soon as possible to main computer or certain devices, they are not able to guarantee the exact time that main computer received. Receiving and sending take certain amount of time and each processing spends significantly more time than receiving or sending buffer time.

Recognizing time considers as important as getting accurate data because the time is also a certain format of each data. Therefore, each sensor is connected with different MCU with time server, the sensor can send data with exact time. The basic concept of connecting time server is acquiring time when the data received and combine the sensor data and time. Then the MCU sends the combined data to requested computer or main computer.

In the Autonomous Vehicle Lab, there are two cars for variety of researches in several different kinds fields. Camera is connected to computer directly since the data amount is tremendously larger than any other sensor also since the camera data is format of frames, it does not need to combine with time. Besides camera sensor, the other sensors are very sensitive at time which means even very small amount of time gap may can cause huge difference of the research result.

Among the all the sensors on the autonomous vehicle, the Lidar sensor is one of the largest sensors. Current autonomous vehicle is using Velodyne HDL – 64E: one of the strongest Lidar in the world. Since it uses strong Lidar sensor, it receives lots of data: approximately ten thousand time more than general Lidar sensor using in the common education lab.

The major goal of this project is how to make stable MCU that accept both time sever data and Velodyne Lidar Sensor data, also how to combine two data and send to main computer.

General Information

Velodyne Lidar Sensor

Velodyne Lidar Sensor has one Ethernet port and power cord. Ethernet is a main data transfer method for Velodyne Lidar Sensor.

The Velodyne Lidar Sensor description refers the sensor is able to send around one million points per second. The data has specific format: package. One package is 1206 Bytes and consisted by 12 blocks with a spin counter and firmware version information at the end of the package. Each block is around 100 Bytes which is 32 laser points information.

Point	
Total Size	3 Bytes

Block	
Header Info Size	2 Bytes
Rotation Info Size	2 Bytes
Points Size	96 Bytes (32 Points)
Total Size	100 Bytes

Package	
Blocks Size	1200 Bytes (12 Blocks)
Status Info Size	6 Bytes
Total Size	1206 Bytes

Sending through the Ethernet, the Velodyne Lidar Sensor will send approximately 2400 package per seconds, which is almost one million points.

Also, the Velodyne Lidar Sensor sends each package through Ethernet as UDP (User Diagram Protocol) data. It sends each package as soon as possible when a package is ready to send.

The Major difference between TCP (Transmission Control Protocol) and UDP data is protocol method. TCP has function that as a message makes its way across the Internet from one computer to another. Therefore, TCP is connection-based protocol. TCP will be suited for applications that high reliability and transmission time is relatively less critical. But UDP has function that one program can send a load of packets to another and that would be the end of the relationship. So, UDP is not connection based. UDP is suitable for applications

that need fast, efficient transmission (Kate 2017). This feature of UDP is the major reason Velodyne Lidar Sensor is using UDP method than TCP method.

Time Server

A time server is a small device for informing the time data and each second's starting point. The output of time server is consisted by one package through Ethernet and GPIO tick each second.

Time Server	
Ethernet Type	UDP
Ethernet Data Size	2 Bytes per Second
GPIO Signal	Beginning of a Second

On the timeserver it is possible that the sending the data via GPIO. Since the Ethernet responding time is significantly longer than GPIO, the Ethernet can be replaced to GPIO signal.

Main Computer

The main computer that will accepts the data from the MCU has to accept through Ethernet port. The main computer will receive the fixed data that includes time data front of the Lidar data.

Main Computer	
Ethernet Type	UDP
Minimum Ethernet Port	802.3u

Main computer is connected to the switch that accepts all other sensors, including Velodyne Lidar Sensor.

General Programs

Velodyne Lidar Sensor Simulator (Appendix Code 1)

This experiment requires a Velodyne Lidar sensor simulator. Since the Velodyne Lidar sensor is not portable from the laboratory, it should be alternated by computer. Most of the computer which is already installed Ethernet port has ability to send approximately same data amount with Velodyne Lidar sensor. Also, the Velodyne Lidar sensor uses Ethernet port (802.3u) which is also used by most of personal computer.

To simulate the Velodyne Lidar Sensor, the computer has to send one package in every 0.4 milliseconds (400 microseconds). Using python code, simulator can send example data very precise and accurate.

Main Computer Simulator

In real autonomous car, there is actual computer that calculate all of the information to make decision; however, the simulator especially limit amount of hardware, it is necessary to make main computer simulator.

Since this experiment does not require to neither calculate nor make decision, it only needs a accepting program for data from MCU.

Attempt 1 - Arduino Due with Ethernet Shield

MCU Name	AT91SAM3X8E
Architecture	ARM Cortex-M3
Clock Speed	84 MHz
SRAM / RAM	96 KB
Storage	512 KB
Ethernet	X
Operating System	X

Since the device is a MCU, the device can measure very accurate real time. Therefore, it does not necessary to be installed specific feature or software in order to measure precise time.

The Arduino experience needs two boards, Arduino Due, and Ethernet Shield, and two software, sensor simulator program, and Arduino program.

Hardware Procedure (Appendix Sketch 1)

Step 1:

Since Arduino Due does not have Ethernet port, it needs an Ethernet shield. Ethernet shield should be installed on Arduino Due. Ethernet shield does not use any of GPIO port; however, it uses ICSP port.

Step 2:

Arduino Due should be connected with computer upon USB. Arduino Due has two micro-USB ports: Programming port and Native port. Both of port are working for inserting program inside but recommends the Programming port.

The major difference between Programming and Native ports is connection of the microcontroller. Programming port makes the microcontroller processor acting as a serial to USB adapter and programs the 32-bit chip (Arduino Due has 32-bit microcontroller). But the Native port is connected directly to the microcontroller. The biggest advantage for using Native port is that programmer can emulate a USB device on Native while reprogramming the Arduino with the Programming port (Lewis 2015).

Step 3:

Ethernet shield should be connected with computer upon Ethernet port. Since computer will assign the Arduino's IP address, therefore, computer will not be necessary to insert IP address manually.

Step 4:

Oscilloscope may can connect on each of LED: not necessary.

Software Procedure (Appendix Code 2)

Step 1:

Arduino Due and Ethernet shield's libraries are already written. Therefore, the before beginning the programming, developer should import all the libraries from Arduino webpage or importing tool from the Arduino IDE.

Step 2:

For sending data, Arduino must know the IP address of the Raspberry Pi 3 as well as Arduino Ethernet shield's MAC Address. If IP address is not precise, it is possible to insert the default IP address on the Arduino.

Step 3:

Import all requirement libraries and code the simple acceptance of data from Ethernet.

Attempt 2 - Raspberry Pi 3

Processor Name	BCM2837
Architecture	ARM Cortex-A53
Clock Speed	1.2 GHz
SRAM / RAM	1 GB
Storage	N/A – MicroSD Card
Ethernet	O – 802.3u
Operating System	O – Raspbian

Unlikely other MCU, Raspberry Pi 3 is actual computer has separate processor and runs with certain operating system. Since the device is very small and fixed amount of memory as well as process speed, it is using Linux operating system call "Raspbian": based on Debian Linux. Also, Raspberry Pi 3 requires a MicroSD card for storage: 16 GB recommended.

Hardware Procedure (Appendix Sketch 4)

Raspberry Pi 3 has only one pre-installed Ethernet port. Since the sensor, time server, and main computer is using Ethernet port to connect, Raspberry Pi 3 needs at least two Ethernet ports. Using USB to Ethernet adapter, the Raspberry Pi 3 can accept two Ethernet ports.

Step 1:

Since the Arduino is now time sever, developer must reprogram the Arduino as time ticking. (Appendix Code 3)

Step 2:

From the Arduino, the pin 13 connect to breadboard. On the breadboard, pin 13 must connect with a LED. Connection for pin 13 is to recognize the time server working correctly by LED light blinking. Also ground port needs to connect to negative axis on the same breadboard. Now each a second, Arduino will send the signal to pin 13 to blink the LED.

Step 3:

Since the Raspberry Pi 3 should receive the tick signal from time server simulator, it should connect with GPIO pin 18 (WiringPi pin 1). This pin connection is receiving port from the time server.

Step 4:

To check the Raspberry Pi 3 receiving the GPIO signal, GPIO pin 18 should connect with another LED. Blinking of the LED will show the Raspberry Pi 3 receive the time server signal properly.

Step 5:

After time server connection, the Raspberry Pi 3 should connect for data port which is Ethernet port. Originally Raspberry Pi 3 needs three Ethernet ports, since the Velodyne Lidar Sensor simulator and Main Computer simulator are actually same computer, it only needs two Ethernet ports. Therefore, connecting USB to Ethernet adapter, Raspberry Pi 3 is now able to connect to one Ethernet port for computer and the other Ethernet port for time server.

Step 6:

If connecting Ethernet is complete, Raspberry Pi 3 needs to check each data from the Velodyne Lidar Sensor simulator. Therefore, Raspberry Pi 3 GPIO pin 23 (WritingPi 4) connects to breadboard and it connects with LED. Each blinking represents the receiving data from simulator.

Step 7:

Finally, oscilloscope may can connect on each of LED: not necessary.

Software Procedure (Appendix Code 6)

Raspberry Pi 3 is required to install several other programs: WirignPi and Real Time Kernel.

Step 1:

Raspberry Pi 3 must be installed the Real Time Kernel which is Preempt RT.

Step 2:

After installation of Preempt RT, it must be installed the Patch the Kernel. Kernel Patch is actually does not necessary. The reason of one more patching is Raspberry Pi 3 will be very unstable for the Real Time patching. Therefore, second patching is to be stabilizing the Raspbian Linux system but sometimes it causes the priority error on the Real Time. Therefore, theoretically, Kernel Patching may can cause the a little inaccurate of the real time.

Step 3:

Developer now may can install the WiringPi on the Raspberry Pi 3. WritingPi should be used for compiling the any C files contains the pin mode.

Installing Real Time Kernel

Using Raspbian, Rapsberry Pi 3 is required certain operating system. Processor unit, which contains operating system, is usually not able to run with real time because the processor will make the operating system as priority for the scheduling. If user can control the scheduling on the Raspberry Pi 3, it can also make Raspberry Pi 3 to real time.

In order to make Raspberry Pi 3 as real time operator with using operating system, it is requirement to install real time kernel. There is several different real time programs in order to make Raspberry Pi 3 to control the scheduling: Preempt RT, and Xenomai.

In order to easy installation and programing in C language, Preempt RT is best choice. Developer may can download the program from the GitHub and install the program following the instruction.

Procedure – Cornell ECE5725 Lab 4

The procedure of Preempt RT installation takes around 2 hours. Also, Raspberry Pi 3 should have at least 2 GB of free space.

Step 1:

Before Raspberry Pi 3 installs the Preempt RT, it needs to clean the unnecessary programs, such as pre-installed Microsoft Minecraft, Linux Libre Office, and Google Chromium Brower. However, if Raspberry Pi 3 has over 16 GB of storage, it does not need to erase any program. After remove all unnecessary programs, Raspberry Pi 3 has to be set as auto cleaning. Following command is setting auto cleaning.

```
$ sudo apt-get clean
```

```
$ sudo apt-get autoremove
```

Step 2:

Since the Linux Kernel does not have auto-installation software, it has to be downloaded from GitHub page. Following command is downloading the installation files.

```
$ cd /home/pi
$ time git clone –b 'rpi-4.4.y'
https://github.com/raspberrypi/linux.git
```

The downloading time usually takes 20 to 30 minutes.

Step 3:

After finishing downloading, it needs to start installing. Following code is for installing the Linux Kernel on the Raspberry Pi 3. Following command is for initiating the installation.

```
$ head makefile
```

Step 4:

If installing Kernel is just initial step of the real time patch, installing Preempt RT is main part of the patch. Raspberry Pi 3 should have the installation resources from the Preempt RT's GitHub page and may start the installation. Following command is downloading and installing the Preempt RT.

```
$ cd ~/home/pi/linux
$ zcat patch-4.4.50-rt63.patch.gz | patch
-p1 > /home/pi/patch.log 2>&1
$ make clean
$ make mrproper
```

After patch, it is necessary to install the ncurses development headers (required by the upcoming menuconfig command). Then it needs to give the kernel image a name (kernel7). The other command insures that the default configuration for the Raspberry Pi 3 is created.

```
$ sudo apt-get install libncurses5-dev
$ KERNEL=kernel7 $ make bcm2709_defconfig
```

Step 5:

The above "make" command creates a skeleton config file to be used by the menuconfig command. Then, it should run make menuconfig to check/change settings for the upcoming compilation. Following command is to run the configuration.

```
$ make menuconfig
```

The menuconfig procedure usually takes more than 90 minutes if there is no error occur.

Using WiringPi

Required to using GPIO signal for time server, Raspberry Pi 3 has to active the GPIO pins. Raspberry Pi 3 pins are pre-installed on the board, but it still needs to be assigned by program such as which pin is doing what function. For this function installing and compile with WiringPi in necessary.

WPi	RPi	Header	RPi	WPi
	3.3v	1	2	5v	
8	SDA	3	4	5v	
9	SCL	5	6	0c	
7	4	7	8	GRD	15
	0v	9	10	GRD	16
0	17	11	12	18	1
2	RV	13	14	0v	
3	22	15	16	23	4
	3.3v	17	18	24	5
12	MOSI	19	20	0v	
13	MISO	21	22	25	6
14	SCLK	23	24	CE0	10
	0v	25	26	CE1	11

Attempt 2 Figure 1: WiringPi pin number

Attempt 2 Figure 1 shows that how to wire the GPIO pin. In addition, in the program script, if the program would compile by WiringPi, the program must be written in proper pin number for example the original pin 17 should be written as pin 0 on WiringPi pin active mode.

Procedure – Cornell ECE5725 Lab 4

Step 1:

First, it is necessary to download the installation files from the WiringPi's GitHub page. Following command is to download the installation files.

```
$ git clone git://git.drogon.net/wiringPi
```

Step 2:

After downloading the files, it needs to be install. Following procedure is installation commands.

```
$ git pull origin
$ cd wiringPi
$ ./build
```

The installation is not take more than 5 minutes. After installation reboot the Raspberry Pi 3 is recommended.

Applying

For applying WiringPi on the file, it has to use specific compiling command line. Following command line is applying WiringPi on the certain program.

```
$ gcc –o BINARY_FILE_NAME FILE_NAME.c -
lwiringPi
```

Test Attempts

GPIO Delay Test

GPIO connection in the MCU is very critical because of the usage of time server. Time server uses not only Ethernet but also GPIO for sending precise signal and data. The main reason of GPIO response testing is the trip between each sensor data is very small: 0.4 milliseconds (400 microseconds).

The main sketch for GPIO Delay Test is Appendix Sketch 2. The GPIO signal generator is the Arduino Due; for the GPIO Delay Test, Ethernet Shield is not necessary. Also, the GPIO signal receiver is Raspberry Pi 3. To detect each signal,

oscilloscope is necessary. Oscilloscope must be connected with all the LED light on the breadboard.

Procedure – Appendix Sketch 2, Appendix Code 4
Step 1:
Arduino Due sends a GPIO signal to pin 13. Pin 13 connected to one LED and Raspberry Pi 3 GPIO pin 18 (WiringPi pin 1).

Step 2:
If Raspberry Pi 3 detects the GPIO signal, it will send another GPIO signal to pin 17 (WiringPi pin 0).

Step 3:
Check the oscilloscope with voltage max 5 each of signal.

Step 4:
Oscilloscope may can connect on each of LED.

Ethernet Delay Test

Since all of the sensors, includes Velodyne Lidar Sensor, is using Ethernet port to send the data, their delaying time for sending comes critical for combining time server data. If the data from the sensor is much smaller than MCU or processor speed is extremely faster than standard recommended processor speed, the data will send with stable time gap. For those general sensors or not big data sensor, the delaying time through transferring data is ignorable.

But Velodyne Lidar Sensor is different. The gap between each package is too small that the Ethernet response speed may effect on the real time. Therefore, it is important to test the Ethernet speed between one device to Raspberry Pi 3.

The main sketch of Ethernet Delay Test is Appendix Sketch 3. The Ethernet data generator is the Arduino Due with Ethernet Shield. Also, the Ethernet data receiver is Raspberry Pi 3. To detect each signal, oscilloscope is necessary. Oscilloscope must be connected with all the LED light on the breadboard.

Procedure – Appendix Sketch 3, Appendix Code 5
Step 1:
Arduino Due sends both a GPIO signal to pin 13 and Ethernet data through the Ethernet shield's port. The IP Address is assigned for Raspberry Pi 3.

Step 2:
If Raspberry Pi 3 detects the Ethernet data, it will send a GPIO signal on pin 17 (WiringPi pin 0).

Step 3:
Check the oscilloscope with voltage max 5 each of signal.

Step 4:
Oscilloscope may can connect on each of LED.

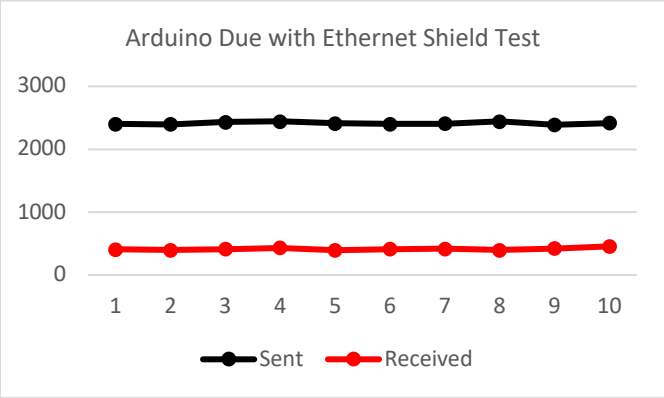
Results

Arduino Due Attempt

Attempt of Arduino Due with Ethernet shield does not satisfy the standard. Although test simulator confirms that the data sent by uniform time, the receiver simulator only received less than 400 packages out of 2400.

Attempt	Sent	Received	Ratio
1	2403	405	16.9 (%)
2	2395	398	16.6
3	2430	411	16.9
4	2442	432	17.7
5	2412	394	16.3
6	2401	412	17.2
7	2405	416	17.3
8	2444	395	16.1
9	2389	423	17.7%
10	2415	455	18.8%
Average	2413.6	414.1	17.2%

Result Figure 1: Arduino Due Attempt test table



Result Figure 2: Arduino Due Attempt test chart

Result Figure 1 shows that the Arduino Due with Ethernet shield is only able to receive around 17% of the data. Result Figure 2 shows that the result is very constant, which means either procedure or software does not cause the low ratio of the receiving.

Raspberry Pi 3 Real Time Kernel Test

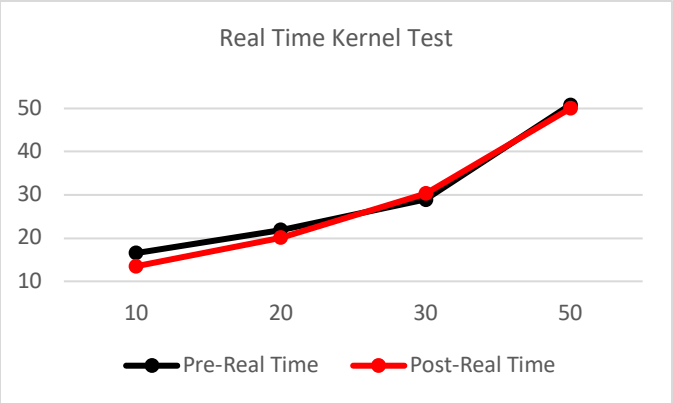
After finish installing the Preempt RT on the Raspberry Pi 3, it is important that the Raspberry Pi 3 is actually working as real time. It is critical that how different is between non-real time and real time.

Time Interval	Measured Time
10 microseconds	16.54 microseconds
20 microseconds	21.87 microseconds
30 microseconds	28.92 microseconds
50 microseconds	50.84 microseconds

Figure 3: Pre-real time installation

Time Interval	Measured Time
10 microseconds	13.50 microseconds
20 microseconds	20.12 microseconds
30 microseconds	30.33 microseconds
50 microseconds	50.01 microseconds

Result Figure 4: Post-real time installation



Result Figure 5: Pre-real time and post-real time comparison

Result Figure 3 refers that the minimum time the normal Raspberry Pi 3 can measure is around 18 microseconds. Also, even if the time is much larger from minimum time, the time has very high volatility.

After installation of the Preempt RT, Result Figure 4, the minimum possible measurement time does improve: from 18 microseconds to 15 microseconds. The difference is only 3 microseconds but since the unit is very small, three microseconds improvement will affect a lot on final result. Furthermore, Result Figure 5 shows the time accuracy is significantly increased after 20 microseconds. By the measurement, the volatility becomes significantly reduced.

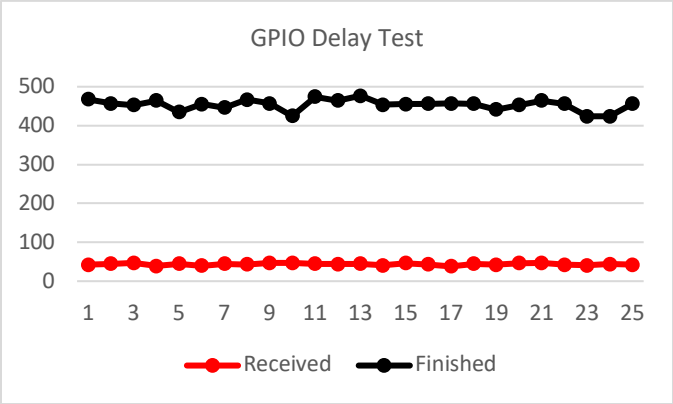
GPIO Delay Test

All the programs have some delay for receiving or sending delay since the processing time is significant. Although the delaying time is very small comparing other processing time but it is importance for Velodyne Lidar Sensor receiving time since the time between one package and the other package is also very small.

Attempt	Received Delay	Finished Delay
1	42 (microseconds)	468 (microseconds)
2	45	457
3	47	453
4	39	464
5	45	435
6	40	455
7	45	446
8	43	467
9	47	457
10	47	425
11	45	474
12	44	464
13	45	477
14	41	454
15	46	455
16	43	456
17	38	457
18	45	456
19	42	441
20	46	453
21	47	464
22	42	456
23	41	424
24	44	424
25	42	456
Average	43.6	453.5

Result Figure 6: GPIO Delay Test Result

Upon the Result Figure 6 shows that the GPIO receiving response time is very short. Since each package is coming through within in 400 microseconds, the package will not be interrupted by GPIO signal acceptance.



Result Figure 7: GPIO Delay Test Graph Result

It is true that the GPIO Delay should be more uniform to confirm for no interruption for Velodyne Lidar Sensor Data. Result Figure 7 is showing that the finished signal has certain volatility, besides received signal is very constant. Finished signal means when Raspberry Pi 3 detects Arduino stops sending signal, therefore, the finished signal time has only trivial effect on the data receiving.

Ethernet Delay Test

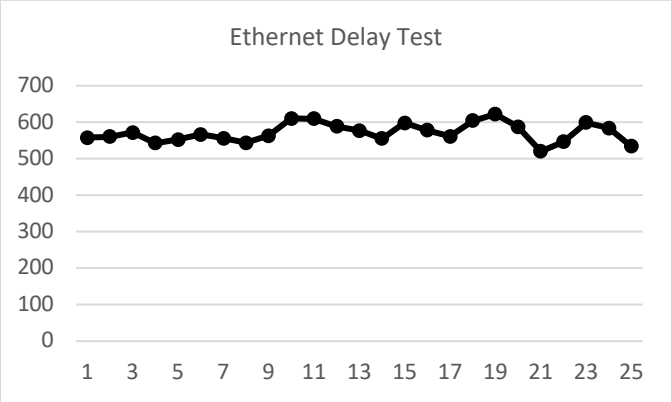
The delay of the Ethernet receiving is very critical. It is well known fact that the sending speed of the Ethernet is extremely fast, but response time is very slow. Using Ethernet port, Velodyne Lidar Sensor has always very small amount of delay.

Attempt	Received Delay
1	558 (microseconds)
2	560
3	572
4	543
5	552
6	566
7	556
8	544
9	563
10	610
11	609
12	588
13	577
14	555
15	598
16	578
17	561
18	604
19	622
20	587
21	520

22	546
23	599
24	584
25	534
Average	571.4

Result Figure 8: Ethernet Delay Test Result

Unlikely GPIO receiving, the Ethernet takes much more time: approximately ten times more. Delaying more than 400 microseconds has tremendous meaning that one package may be miss the time slot.



Result Figure 9: Ethernet Delay Test Graph Result

One of the critical characteristics in the Ethernet receiving is the volatility. The average delay time is approximately 570 microseconds; however, it does not always nearest number each result: sometimes delaying times in going down to 530s and up to 610s.

The major reason of high volatility is waiting time. Unlikely TCP protocol, receiving UDP data has waiting time.

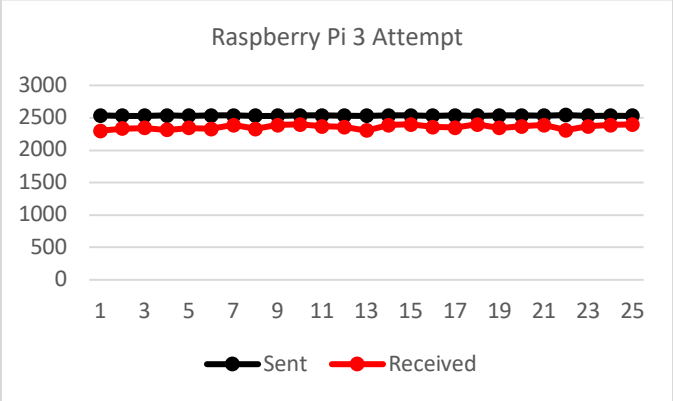
Raspberry Pi 3 Attempt

Checking whether Raspberry Pi 3 is working or not is very obvious. Built a program to count how many sent from the simulator and received by Raspberry Pi 3.

Attempt	Send	Received	Ratio
1	2534	2301	90.8 (%)
2	2533	2333	92.1
3	2533	2345	92.5
4	2534	2312	91.2
5	2533	2342	92.5
6	2534	2330	91.9
7	2534	2387	94.2
8	2533	2327	91.9
9	2533	2387	94.2
10	2534	2399	94.7
11	2535	2369	93.4
12	2532	2356	93.0
13	2533	2303	90.9
14	2534	2386	94.2
15	2534	2397	94.6
16	2533	2359	93.1
17	2536	2349	92.6
18	2533	2397	94.6
19	2534	2344	92.5
20	2534	2367	93.4
21	2532	2389	94.4
22	2543	2308	90.8
23	2533	2368	93.5
24	2532	2387	94.3
25	2533	2399	94.7
Average	2533.8	2357.6	93.0

Result Figure 10: Raspberry Pi 3 Attempt Result

Upon the table, Result Figure 10, the result ratio is around 93.0 %. The number itself is pretty high but the Raspberry Pi 3 still misses some of the data.

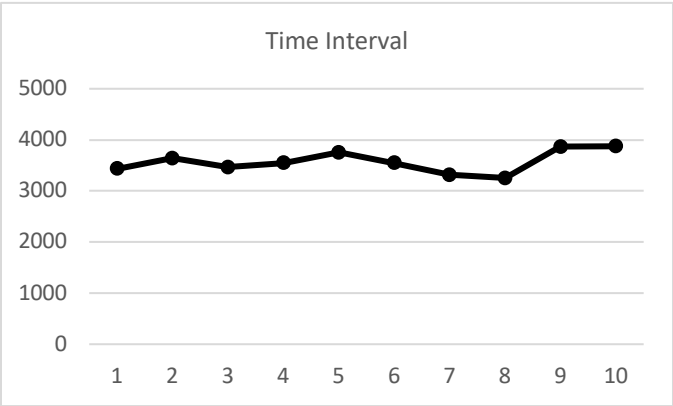


Result Figure 11: Raspberry Pi 3 Attempt Chart Result

The Result Figure 11 shows that the rate is very low volatility. Comparing the result of Arduino Due with Ethernet Shield, The Raspberry Pi 3 gives very steady and accurate of the time.

Package Order	Mean Delta Time	Received Ratio
1	3432 (microseconds)	93.4
2	3643	91.2
3	3464	90.2
4	3544	94.3
5	3755	93.2
6	3545	92.3
7	3312	94.6
8	3254	94.7
9	3867	93.4
10	3874	95.3
Average	3569	93.26

Result Figure 12: Time Interval Result



Result Figure 13: Time Interval Chart Result

The Result Figure 12 shows that the time interval of each package is very steady and fixed. It is possible that the USB

adapter cause the several delays on the UDP data. Although there are some missed data, the improvement is significant. Also Result Figure 13 shows that the volatility is significantly lower than Arduino Due with Ethernet Shield.

Conclusion

The experiment of Raspberry Pi treating as MCU, proves that the even processor can be used as MCU unless successful installation of Real Time. It is true that the Raspberry Pi 3 is not recommended for real time installation, but it is true that the performance is not as bad as the criticisms. The major concern of the installation of real time on the Raspberry Pi 3 is now stabilizing the system. Since the Raspberry Pi 3 has only minimum requirements of the computer, once it causes an error, whole system will stop working. An error will be caused by the real time installation very often. Therefore, even the system can handle more accurate time ticking, it should reduce the performance for stabilizing.

Besides the real time installation, there are two critical concepts when developing the MCU: how to find a proper MCU and how to handle the UDP data.

Finding a proper MCU is one of the primer tasks before start concerning about algorithms or methods. Before selecting MCU, it is one of the requirements jobs that calculate approximate requirement space and clock speed. The approximation of those performance depends on the receiving and sending data amount with calculation. Since the MCU does not have operating system, those amount of data is directly and highly influenced on requirement performance. After calculating the approximation of specification, it is proper that searching MCU based on the specification. It is also better that choosing a little higher performance than approximation. When selecting MCU, concerning about combining several other board, like shield, is not proper idea. Connecting shield may cause serious data sending delay between shield and actual board. Therefore, it is highly recommended select one board that all the feature already installed.

After selecting MCU, the other critical key is knowing data flowing method. For this experiment, all the data transmission is using UDP. Since the data is relatively large and time delay is critically important, the usage of UDP is necessary. But one of the hardest parts of using UDP method is waiting time. Using a MCU does not need to consider the waiting time since it does not have operating system, but using Raspberry Pi 3 as MCU has to concern the waiting feature of the UDP data since it is using Raspbian (Linux) as operating system. To solve this problem, usage of flagging is the best. The program does not need to wait any time if program is using flagging instead of signal for whole UDP data.

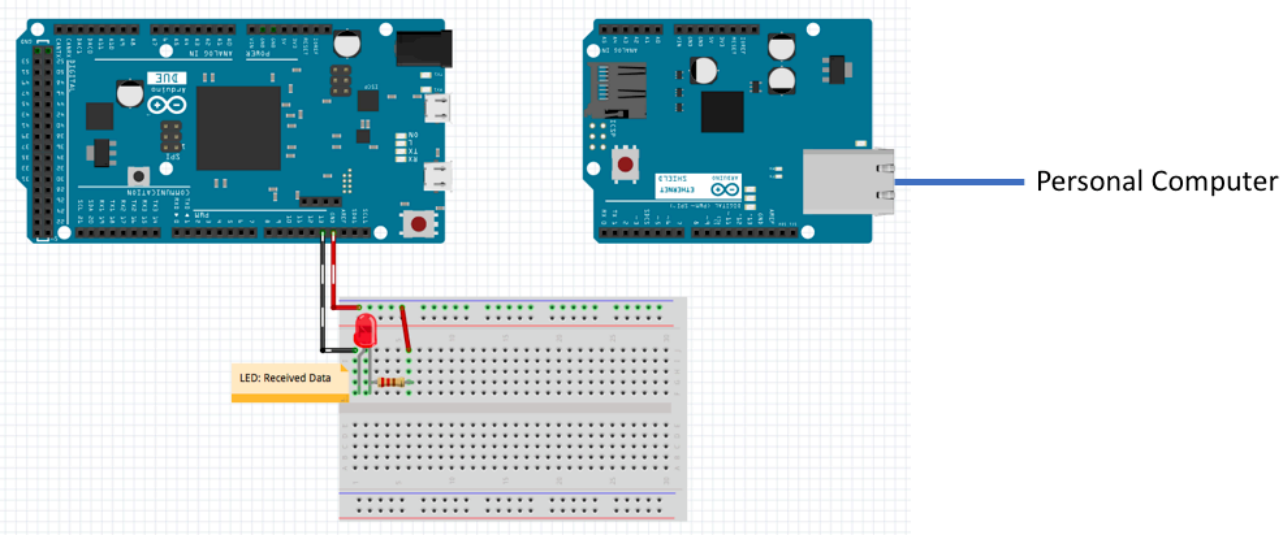
Raspberry Pi 3 with real time installation was one of the critical improvements of the MCU feature. Unfortunately, Raspberry Pi 3 is not enough performance to calculate or process any other feature besides combining time server data, it has highly potential to apply on other process. If someday Raspberry improves high enough to calculate other processes, it is meaningful project to concern how to embed real time kernel than now.

Reference

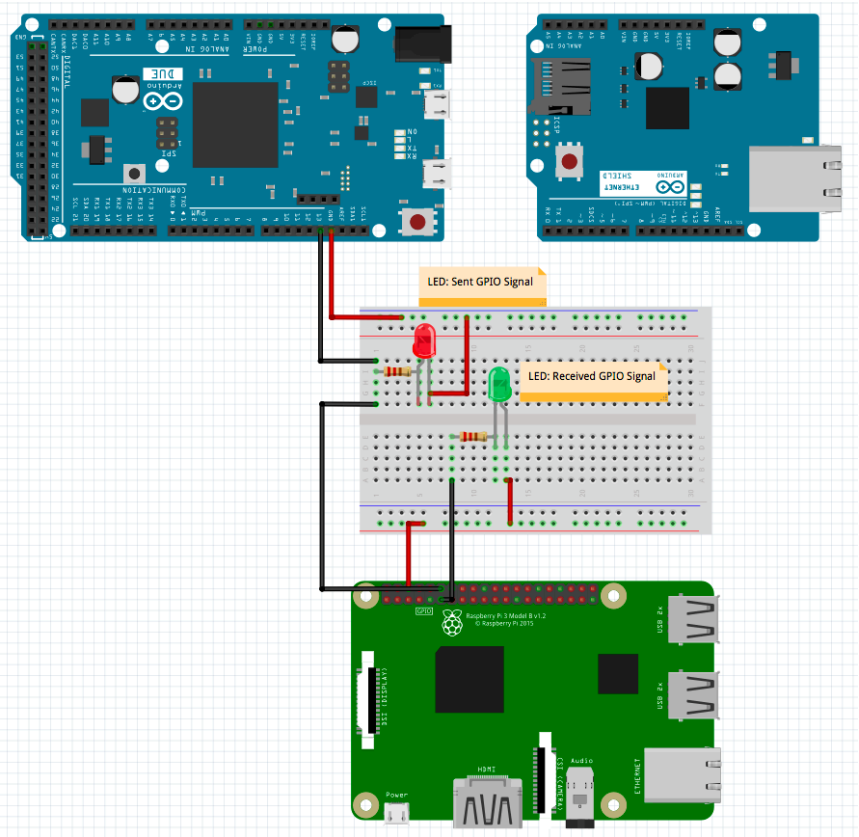
Kate T., Pooja Sehgal, Nikhilesh Jasuja, Leonel Guillermo Mendoza Estrada, Haroon Malik, Abhivendra Singh, Anujan. (2017, Dec 4). TCP vs UDP [Blog post]. Retrieved from https://www.diffen.com

Lewis, James. (2015, November 11). Difference between Programming and Native Ports [Blog post]. Retrieved from https://www.baldengineer.com.

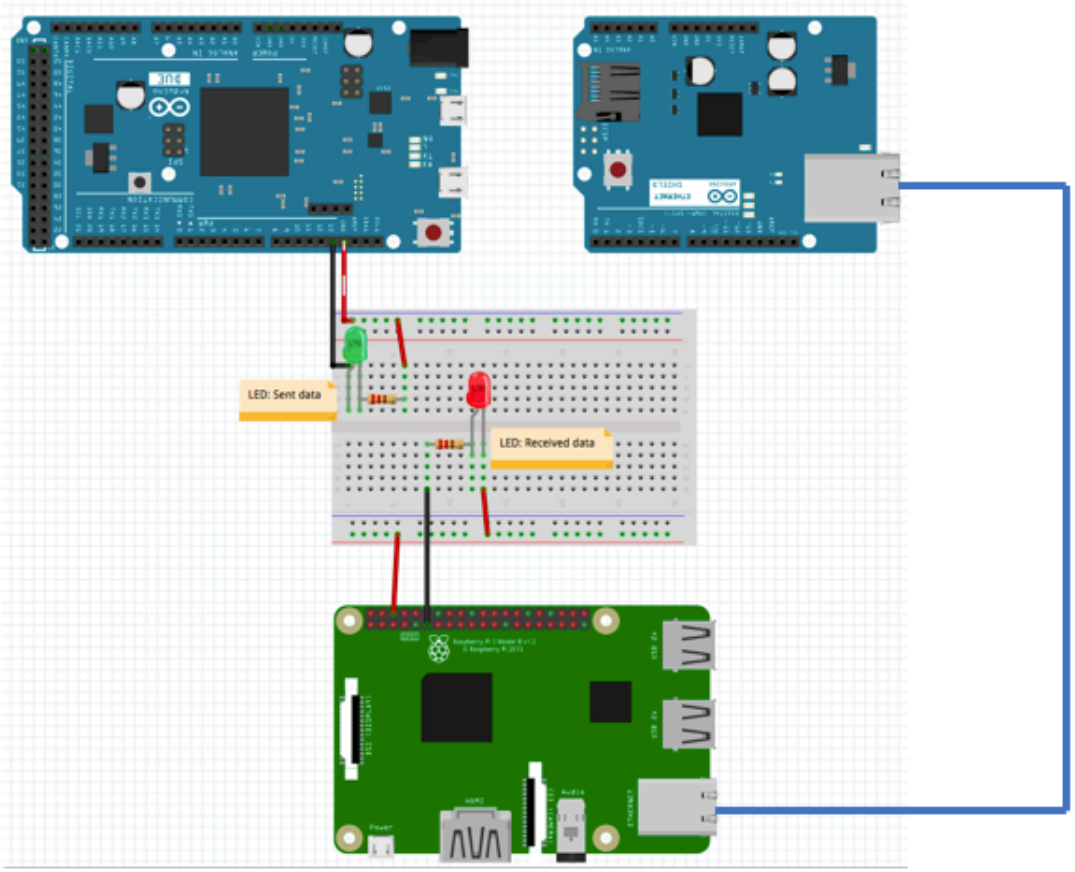
Appendix Sketch 1 – Arduino Due Attempt



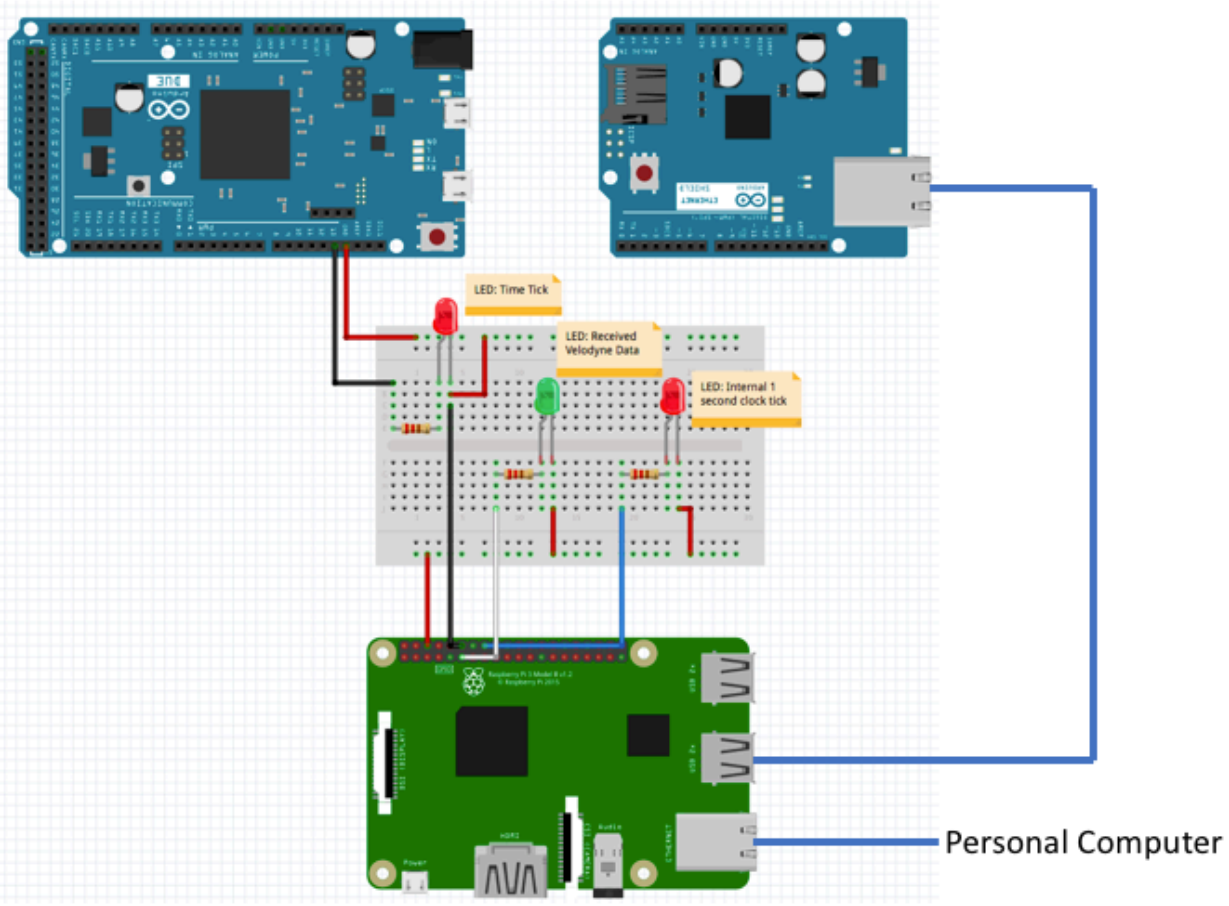
Appendix Sketch 2 – GPIO Delay Test



Appendix Sketch 3 – Ethernet Delay Test



Appendix Sketch 4 – Raspberry Pi 3 with Real Time



Appendix Code 1 - RandomDataMCULidarSend.py

```
#################################################################
#
# RandomDataMCULidarSend.py
#
# Won Yong Ha
#
# Sep 20 2017
#
# This program is a Velodyne Lidar simulator sending
# approximately three MB per second.
#
#################################################################

from socket import *
import time
import random

IPADDRESS = 'IP_Address' #Receiver IP Address
PORTNUMBER = Port_Number #Receiver Port Number

address = (IPADDRESS, PORTNUMBER)
client_socket = socket(AF_INET, SOCK_DGRAM)
client_socket.settimeout(1)

# Making random data
tail = ""
for i in range(1200): # 1200 Byte
  tail += "0"

count = 0;

time_end = time.time() + 60

# Sending data
while(time.time() < time_end):
  data = ('%06d' % count) + tail # 6 Byte

  try:
    client_socket.sendto(data, address)
    print data[0:6]
    count += 1
  except:
    print "fail"
  time.sleep(0.00035) #Approximately 2500 packages
```

Appendix Code 2 – MCU_Lidar.ino

```
################################################################
#
# MCU_Lidar.ino
#
# Won Yong Ha
#
# Sep 22 2017
#
# This program is a Arduino MCU Velodyne Lidar Sensor data
# accepter.
#
################################################################

#include <Ethernet3.h>
#include <EthernetUdp3.h>
#include <SPI.h>

//Ethernet Setting
byte mac[] = {}; // MAC Address for Ethernet Shield
IPAddress ip {}; // Current IP Address to send
unsigned int localPort = ; // Port Number
char packetBuffer[1212];
String datReq;
int packetSize;
EthernetUDP Udp;

void setup() {
  Serial.begin(19200);
  Serial.println("Setting Start");

  Serial.println("Ethernet Setting");
  Ethernet.begin(mac, ip);
  Udp.begin(localPort);
  Serial.println("Done");

  Serial.println("Setting Finished");
  delay(500);
}

void loop() {
  packetSize = Udp.parsePacket();
  if(packetSize > 0) {
    Udp.read(packetBuffer, packetSize);
    Serial.print(packetBuffer);
    Udp.beginPacket(Udp.remoteIP(), Udp.remotePort());
    Udp.print(packetBuffer);
    Udp.endPacket();
  }
}
```

Appendix Code 3 - EthernetSendingRPiData.ino

```
#############################################################
#
# EthernetSendingRPiData.ino
#
# Won Yong Ha
#
# Sep 22 2017
#
# This program is a Arduino Time Server Simulator
#
#############################################################

#include <Ethernet3.h>
#include <EthernetUdp3.h>
#include <SPI.h>

//Ethernet Setting
byte mac[] = {}; //MAC Address for Ethernet Shield
IPAddress ip {}; //Arduino Address
IPAddress RasPiIP {}; //RaspberryPi Address
unsigned int localPort = ; //Listening Port
char packetBuffer[2];
String datReq;
int packetSize;
EthernetUDP Udp;

char seconds_msb;
char seconds_lsb;

unsigned int sec;

void setup() {
  Serial.begin(19200);
  Serial.println("Setting Start");

  Serial.println("Ethernet Setting");
  Ethernet.begin(mac, ip);
  Udp.begin(localPort);
  Serial.println("Done");

  Serial.println("Setting Finished");
  delay(500);
  sec = 0;

  pinMode(13, OUTPUT);
}

void loop() {
  sec++;
  digitalWrite(13, HIGH);

  Serial.println("TEST");
  Serial.println(sec);

  seconds_msb = (sec >> 8);
  seconds_lsb = sec & 0xff;
  packetBuffer[0] = seconds_msb;
  packetBuffer[1] = seconds_lsb;
  packetBuffer[1] = '\0';
  Serial.println(seconds_msb);
  Serial.println(seconds_lsb);

  unsigned int temp;
  temp = packetBuffer[0] << 8;
  int seconds = temp|packetBuffer[1];
  Serial.println(seconds);

  Udp.beginPacket(RasPiIP, 5555);
  Udp.write(packetBuffer);
  Udp.endPacket();
  delay(100);
  digitalWrite(13, LOW);
  Serial.println(packetBuffer);
  delay(900);
}
```

Appendix Code 4 - RealTime_GPIOCommunicationTest.c

```
////////////////////////////////////////////////////////////////
//
// RealTime_GPIOCommunicationTest.c
//
// Won Yong Ha
//
// Oct 21 2017
//
// This program is a GPIO Delay Testing for Raspberry Pi 3
//
////////////////////////////////////////////////////////////////

#define _POSIX_C_SOURCE 200112L

#include <stdlib.h>
#include <stdio.h>
#include <sched.h>
#include <sys/mman.h>
#include <time.h>
#include <string.h>
#include <wiringPi.h>

#define NSEC_PER_SEC    (1000000000) /* The number of nsecs per sec. */

#define LEDIN 0
#define LEDOUT 1


// RT parameters
#define MY_PRIORITY (49)
#define MAX_SAFE_STACK (8*1024)

void stack_prefault(void) {
  unsigned char dummy[MAX_SAFE_STACK];
  memset(dummy, 0, MAX_SAFE_STACK);
  return;
}

struct timespec t;
struct sched_param param;

int main (int argc, char** argv)
{
  int period = 50000; // 50 us
  int PinValue = 0;  // hi/low indication of output Pin
  unsigned int current_sec, start_sec;
  float freq;

  if (argc>=2 && atoi(argv[1])>0 ) {  // if we have a positive input value
     period = atoi(argv[1]);
  }
  printf ("Set 1/2 period to %d nanoseconds\n",period);
  freq = NSEC_PER_SEC * ((float)1/(2*period));
  printf ("   Frequency =  %f Hz\n",freq);

  //writingPi Setup
  wiringPiSetup();
  pinMode(4, OUTPUT);

  // Set priority and scheduler algorithm
  param.sched_priority = MY_PRIORITY;
  if (sched_setscheduler(0, SCHED_FIFO, &param) == -1) {
    perror("sched_setscheduler failed");
    exit(-1);
  }

  // Lock memory
  if (mlockall(MCL_CURRENT|MCL_FUTURE) == -1) {
    perror("mlockall failed");
    exit(-2);
  }

  // Prefault the stack
  stack_prefault();

  clock_gettime(CLOCK_MONOTONIC ,&t); // setup timer t
```

```
  t.tv_nsec += period;   // add in initial period
  printf ( "sec = %d \n", t.tv_sec);
  start_sec = t.tv_sec;
  current_sec = 0;

  pinMode (LEDOUT, OUTPUT);
  pinMode (LEDIN, INPUT);

  while(current_sec < 300000) {
    clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &t, NULL); // delay a bit...
     PinValue = PinValue ^ 1;
     t.tv_nsec += period;   // add in initial period

     while (t.tv_nsec >= NSEC_PER_SEC) {   // This accounts for 1 sec rollover
        t.tv_nsec -= NSEC_PER_SEC;
        t.tv_sec++;
        current_sec = t.tv_sec - start_sec;  // how many seconds since we started?
     }
     if(digitalRead(LEDIN)){
       printf("Seconds: %d\t\t", current_sec);
       digitalWrite(LEDOUT, HIGH);
     } else {
       printf("No Signal\t\t");
       digitalWrite(LEDOUT, LOW);
     }
  }
  printf ("stopped at %d seconds\n", current_sec);
  return 0 ;
}
```

Appendix 5 - RealTime_EthernetCommunicationTest.c

```
////////////////////////////////////////////////////////////////
//
// RealTime_GPIOCommunicationTest.c
//
// Won Yong Ha
//
// Oct 20 2017
//
// This program is a Ethernet Delay Testing for Raspberry Pi 3
//
////////////////////////////////////////////////////////////////

#define _POSIX_C_SOURCE 200112L

#include <stdlib.h>
#include <stdio.h>
#include <sched.h>
#include <sys/mman.h>
#include <time.h>
#include <string.h>
#include <wiringPi.h>

#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <unistd.h>
#include <errno.h>
#include <arpa/inet.h>

#define NSEC_PER_SEC    (1000000000) /* The number of nsecs per sec. */

#define LEDOUT 0

// RT parameters
#define MY_PRIORITY (49)
#define MAX_SAFE_STACK (8*1024)

void stack_prefault(void) {
  unsigned char dummy[MAX_SAFE_STACK];
  memset(dummy, 0, MAX_SAFE_STACK);
  return;
}

struct timespec t;
struct sched_param param;

int main (int argc, char** argv)
{
  int period = 50000; // 50 us
  int PinValue = 0;  // hi/low indication of output Pin
  unsigned int current_sec, start_sec;
  float freq;

  if (argc>=2 && atoi(argv[1])>0 ) {  // if we have a positive input value
     period = atoi(argv[1]);
  }
  printf ("Set 1/2 period to %d nanoseconds\n",period);
  freq = NSEC_PER_SEC * ((float)1/(2*period));
  printf ("   Frequency =  %f Hz\n",freq);

  //writingPi Setup
  wiringPiSetup();
  pinMode(4, OUTPUT);

  // Set priority and scheduler algorithm
  param.sched_priority = MY_PRIORITY;
  if (sched_setscheduler(0, SCHED_FIFO, &param) == -1) {
    perror("sched_setscheduler failed");
    exit(-1);
  }

  // Lock memory
  if (mlockall(MCL_CURRENT|MCL_FUTURE) == -1) {
    perror("mlockall failed");
    exit(-2);
  }
```

```
  // Prefault the stack
  stack_prefault();

  clock_gettime(CLOCK_MONOTONIC ,&t); // setup timer t
  t.tv_nsec += period;   // add in initial period
  printf ( "sec = %d \n", t.tv_sec);
  start_sec = t.tv_sec;
  current_sec = 0;

  //network check
  int sockfd = 0, n = 0;
  char recvBuffer[1206];
  struct sockaddr_in serv_addr;

  memset(recvBuffer, '0', sizeof(recvBuffer));
  if((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) { //SOCK_
      printf("\n Error : Could not create socket \n");
      exit(-2);
  }

  serv_addr.sin_family = AF_INET;
  serv_addr.sin_port = htons(5555);
  serv_addr.sin_addr.s_addr = inet_addr("0.0.0.0");

  if(bind(sockfd, (struct sockaddr *) & serv_addr, sizeof(serv_addr)) < 0) {
    printf("\n Error : Connection failed \n");
    exit(-2);
  }

  pinMode (LEDOUT, OUTPUT);

  while(current_sec < 300000) {
    clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &t, NULL); // delay a bit...
    t.tv_nsec += period;   // add in initial period
    digitalWrite(LEDOUT, LOW);

     while (t.tv_nsec >= NSEC_PER_SEC) {   // This accounts for 1 sec rollover
        t.tv_nsec -= NSEC_PER_SEC;
        t.tv_sec++;
        current_sec = t.tv_sec - start_sec;  // how many seconds since we started?
     }

     printf("Working\n");
     //digitalWrite(LEDOUT, HIGH);

     if((n = read(sockfd, recvBuffer, sizeof(recvBuffer) - 1)) > 0) {

       printf("Seconds: %d\n", current_sec);
       char headStr[7];
       strncpy(headStr, recvBuffer, 6);
       headStr[7] = '\0';
       printf("Seconds: %s\n", headStr);
       digitalWrite(LEDOUT, HIGH);
     }
  }
  printf ("stopped at %d seconds\n", current_sec);
  return 0 ;
}
```

Appendix 6 – RealTime_Lidar.c

```
////////////////////////////////////////////////////////////////
//
// RealTime_Lidar.c
//
// Won Yong Ha
//
// Nov 14 2017
//
// This program is a Lidar MCU for Raspberry Pi 3
//
////////////////////////////////////////////////////////////////

#define _POSIX_C_SOURCE 200112L

#include <stdlib.h>
#include <stdio.h>
#include <sched.h>
#include <sys/mman.h>
#include <time.h>
#include <string.h>
#include <wiringPi.h>

#include <sys/socket.h>
#include <sys/types.h>
#include <sys/unistd.h>
#include <sys/fcntl.h>
#include <sys/time.h>
#include <sys/select.h>
#include <netinet/in.h>
#include <unistd.h>
#include <errno.h>
#include <arpa/inet.h>

#define LEDIN 1
#define LEDOUT 0
#define LEDOUT2 4

#define RXTIMESIZE 10
#define RXDATASIZE 1206

// RT parameters
#define MY_PRIORITY (49)
#define MAX_SAFE_STACK (8*1024)

void stack_prefault(void) {
  unsigned char dummy[MAX_SAFE_STACK];
  memset(dummy, 0, MAX_SAFE_STACK);
  return;
}

struct timespec t;
struct sched_param param;

int main (int argc)
{
  //writingPi Setup
  wiringPiSetup();

  // Set priority and scheduler algorithm
  param.sched_priority = MY_PRIORITY;
  if (sched_setscheduler(0, SCHED_FIFO, &param) == -1) {
    perror("sched_setscheduler failed");
    exit(-1);
  }

  // Lock memory
  if (mlockall(MCL_CURRENT|MCL_FUTURE) == -1) {
    perror("mlockall failed");
    exit(-2);
  }

  // Prefault the stack
  stack_prefault();

  clock_gettime(CLOCK_MONOTONIC ,&t);
  printf ( "sec = %d \n", t.tv_sec);
```

```
//NETWORK CHECK
int sockfdTime = 0, nTime = 0, sockfdData = 0, nData = 0;
char recvBufferTime[RXTIMESIZE];
char recvBufferData[RXDATASIZE];
struct sockaddr_in serv_addrTime;
struct sockaddr_in serv_addrData;
socklen_t fromlenTime, fromlenData;

serv_addrTime.sin_family = AF_INET;
serv_addrTime.sin_port = htons(5555); // TimePort : 5555
serv_addrTime.sin_addr.s_addr = inet_addr("0.0.0.0");

serv_addrData.sin_family = AF_INET;
serv_addrData.sin_port = htons(8888); // DataPort : 8888
serv_addrData.sin_addr.s_addr = inet_addr("0.0.0.0");

memset(recvBufferTime, '0', sizeof(recvBufferTime));
if((sockfdTime = socket(AF_INET, SOCK_DGRAM, 0)) < 0) { //SOCK_DGRAM or SOCK_STREAM
  printf("\n Error : Could not create Time socket \n");
  exit(-2);
}

memset(recvBufferData, '0', sizeof(recvBufferData));
if((sockfdData = socket(AF_INET, SOCK_DGRAM, 0)) < 0) { //SOCK_DGRAM or SOCK_STREAM
  printf("\n Error : Could not create Data socket \n");
  exit(-2);
}

// Timeout set
struct timeval timevalue;
timevalue.tv_sec = 0;    // second
timevalue.tv_usec = 10;  // micro second

if(setsockopt(sockfdTime, SOL_SOCKET, SO_RCVTIMEO, (const char*) &timevalue, sizeof(struct timeval)) < 0) {
  printf("\n Error : Could not set time out \n");
  exit(-2);
}
printf("Timeout : %d second \t", timevalue.tv_sec);
printf("%d microsecond \n", timevalue.tv_usec);

// Connecting set
if(bind(sockfdTime, (struct sockaddr *) & serv_addrTime, sizeof(serv_addrTime)) < 0) {
  printf("\n Error : Time connection failed \n");
  exit(-2);
}
fromlenTime = sizeof serv_addrTime;

if(bind(sockfdData, (struct sockaddr *) & serv_addrData, sizeof(serv_addrData)) < 0) {
  printf("\n Error : Data connection failed \n");
  exit(-2);
}
fromlenData = sizeof serv_addrData;

fd_set set;

pinMode (LEDIN, INPUT);
pinMode (LEDOUT, OUTPUT);
pinMode (LEDOUT2, OUTPUT);
digitalWrite(LEDOUT, LOW);
digitalWrite(LEDOUT2, LOW);

unsigned int seconds_msb;
unsigned int seconds_lsb;

int testCount = 0;
int bufferTime = 0;
long fixedNSEC = 0;
long currentNSEC = 0;
long NSEC = 1000000000;
long preNSEC = 0;

printf("Loop Start\n");
while(1) {
  if(digitalRead(LEDIN)) {
    if(recvfrom(sockfdTime, recvBufferTime, sizeof(recvBufferTime), 0, &serv_addrTime, &fromlenTime) > 0) {

 seconds_msb = recvBufferTime[0];
 seconds_lsb = recvBufferTime[1];
```

```
   bufferTime = (seconds_msb << 8) | seconds_lsb;

   clock_gettime(CLOCK_MONOTONIC ,&t);
   fixedNSEC = t.tv_nsec;
   digitalWrite(LEDOUT2, HIGH);
   testCount = 0;
      }
    }

    FD_ZERO(&set);
    FD_SET(sockfdData, &set);
    int received = select(sockfdData + 1, &set, NULL, NULL, &timevalue);
    printf("%d", received);
    if(received == -1) {
      perror("\n Error : Select data failed \n");
      exit(-2);
    }

    if(FD_ISSET(sockfdData, &set)) {
      digitalWrite(LEDOUT, HIGH);
      clock_gettime(CLOCK_MONOTONIC ,&t);
      printf("\n");
      if(recvfrom(sockfdData, recvBufferData, sizeof(recvBufferData), 0, &serv_addrData, &fromlenData) > 0) {
   printf("SSeconds: %d\t\t", bufferTime);
   if(t.tv_nsec > fixedNSEC) {
     currentNSEC = t.tv_nsec - fixedNSEC;
   } else {
     currentNSEC = NSEC + t.tv_nsec - fixedNSEC;
   }
   printf("NSeconds: %.9ld\t", currentNSEC);
   if(currentNSEC > preNSEC) {
     printf("DSeconds: %.9ld\t", currentNSEC - preNSEC);
   } else {
     printf("DSeconds: %.9ld\t", currentNSEC + NSEC - preNSEC);
   }
   printf("Data: %.*s\t", 6, recvBufferData + 0);

   preNSEC = currentNSEC;
      }
    }

    digitalWrite(LEDOUT2, LOW);
    digitalWrite(LEDOUT, LOW);
  }

  return 0 ;
}
```