SLAMDUNK REFERENCE MANUAL

Version 1.0

Thomas Aven
Lasse Agentoft Eggen
Ole Alexander Hole
Odd Kristian Kvarmestøl
Aslak Sheker Mkadmi
Jan Olav Moltu
Sigve Sjømæling Nordgaard
Daniel Romanich
Tord Standnes
Hallvard Stemshaug

November 2018

Contents

1 Introduction to SLAMDUNK - Simultaneous Localization and Mapping by a Dynamic Uniform Normalized Kernel 3
1.1 System Summary . . . 3
1.2 Application method . . . 3
1.3 Assumptions . . . 3

2 SLAMDUNK setup from scratch 3
2.1 Downloading and installing software . . . 3
2.2 Setting up network . . . 4
2.2.1 Setup script . . . 4
2.2.2 Network via ethernet . . . 4
2.2.3 Network via WiFi . . . 5
2.3 Project deployment . . . 6
2.4 PCB (I/O-card) . . . 7

3 System description 7
3.1 SLAMDUNK system . . . 7
3.1.1 build . . . 7
3.1.2 efm32 . . . 8
3.1.3 libs . . . 8
3.1.4 pcb . . . 8
3.1.5 pynq . . . 8
3.1.6 pynqslam . . . 8
3.1.7 slamvis . . . 8
3.1.8 tools . . . 8
3.2 Communication . . . 9
3.3 SLAM algorithm . . . 9
3.4 System limitations . . . 10
3.5 Potential improvements . . . 11

4 Printed Circuit Board 11
4.1 PCB Design . . . 11
4.2 Components . . . 12
4.2.1 MCU . . . 12
4.2.2 Compiling . . . 12
4.2.3 Drivers . . . 12
4.2.4 LIDAR . . . 13
4.2.5 GPIO . . . 13

5 FPGA components 13
5.1 SPI . . . 14
5.2 Double buffering . . . 14
5.3 Preprocessing . . . 14
5.4 Computing sin and cos . . . 15

1 Introduction to SLAMDUNK - Simultaneous Localization and Mapping by a Dynamic Uniform Normalized Kernel

1.1 System Summary

The SLAMDUNK system tackles the simultaneous localization and mapping (SLAM) problem. It does this by using a light detection and ranging unit (LIDAR) and the hector-SLAM algorithm. The system consists of a PYNQ, a custom PCB and hardware components.

The system outputs a live 2D map based on the LIDAR data.

1.2 Application method

There are several ways to solve the SLAM problem. We evaluated several approaches throughout the project. We ended up with only using LIDAR data as input after selecting hector-SLAM as our primary algorithm. The LIDAR data is streamed and processed through the system, going from LIDAR to EFM32 to Cortex A9 to a computer with the visualizer. The EFM32 chip is also set up to communicate with an IMU and other sensors, but due to the properties of hector-SLAM and the inaccuracy of the IMU, we have not used these sensors in the application. The hector-SLAM algorithm reads segments of LIDAR data to make a map. Each new segment is compared to the map and a transformation is calculated, so that the segment can be placed correctly on the map.

1.3 Assumptions

We assume the unit moves in a horizontal plane, which excludes mapping of multiple floors. This is an assumption we make due to the nature of the 2D map output.

2 SLAMDUNK setup from scratch

2.1 Downloading and installing software

Install the PYNQ image from xilinx github repo: `https://files.digilent.com/Products/PYNQ/pynq_z1_v2.1.img.zip`

Flash the image to the SD-card and your PYNQ should be ready. Connect to the PYNQ using on of these options:

- Serial - connect to the PYNQ via microUSB. You can now communicate to it over its serial port using minicom:

  ```
  sudo minicom -D /dev/ttyUSBX -d 115200
  ```

  with "X" as the number of the connected USB interface.

- SSH - Find the PYNQ's link-local ipv6 address by pinging all hosts on your ethernet interface: (you need a dynamic ipv6 address on the interface for this)

```
ping6 ff02::1%<your eth interface>
```

  If the PYNQ is the only thing connected to your computer the, address marked with "(Dup!)" is the PYNQ. SSH to this address with username and password "xilinx":

```
ssh xilinx@fe80::218:3eff:fe02:6ae5%<eth>:
```

2.2 Setting up network

2.2.1 Setup script

We have created a bash script, slamdunk/pynq/config/pynq_setup/setup.sh, that can be run to setup the PYNQ with a working network configuration and new password. The following two subsections is a manual description of the setup which mirrors what the script does.

2.2.2 Network via ethernet

It is possible to setup a static IP-address on the PYNQ so you can ssh to a set address each time. Doing this, it is also possible to connect the PYNQ to the internet by bridging network from our connected computer.

First assign the eth interface on the PYNQ to a static IP-address in the /etc/network/interfaces file:

```
auto eth0
iface eth0 inet static
address 192.168.2.99
netmask 255.255.255.0
gateway 192.168.2.89
```

and do the same on the ethernet port on your own compurer with an address on the same subnet. For example:

```
auto <eth interface>
iface <eth interface> inet static
address 192.168.2.89
netmask 255.255.255.0
```

Add the address to your ssh config (/.ssh/config):

```
Host pynq
    Hostname 192.168.2.99
    User xilinx
```

you can now ssh and rsync using "pynq" instead of the ip address.

To connect the PYNQ to the internet setup iptable rules on your host computer and enable ipv4 forwarding:

```
sudo iptables -t nat -A POSTROUTING \
    -o <wifi interface> \
    -s 192.168.2.0/24 -j MASQUERADE
sudo iptables -I FORWARD -o <eth interface> \
    -s 192.168.2.0/24 -j ACCEPT
sudo iptables -I INPUT -s 192.168.2.0/24 -j ACCEPT
sudo sysctl -w net.ipv4.ip_forward=1
```

Finaly the PYNQ needs a nameserver for dns. For simplicity sake we hardcode NTNUs nameserver with google as a backup. Add this to the interfaces-file:

```
dns-nameservers 129.241.0.200
dns-nameservers 129.241.0.201
dns-nameservers 8.8.8.8
```

2.2.3 Network via WiFi

Using a wifi dongle we can give the PYNQ wifi capabilities. We are using the edimax N150 [1]. To make it work we need to first compile the driver for the xilinx kernel.

```
git clone https://github.com/lwfinger/rtl8723bu
```

the linux-xilinx kernel headers in the PYNQ image is sadly broken, and we have to import them directly from xilinx:

```
apt install bc
git clone https://github.com/Xilinx/linux-xlnx.git \
    --depth 1 --branch xilinx-v2017.4
cd ~/linux-xlnx && make headers_install
```

make the lib-files point to the correct headers:

```
ln -sf /home/xilinx/linux-xlnx \
/lib/modules/4.9.0-xilinx/build
```

and finaly compile the driver:

```
cd ~/rtl8723bu
make
```

load the driver:

```
cd ~/rtl8723bu
sudo insmod 8723bu.ko
```

[1] https://www.edimax.com/edimax/merchandise/merchandise_detail/data/edimax/in/wireless_adapters_n150/ew-7811un/

setup autoloading of the driver on boot:

```
cd ~/rtl8723bu
cp 8723bu.ko /lib/modules/4.9.0-xilinx/kernel/driver/
echo "8723bu" >> /etc/modules
sudo depmod -a
```

To SSH to the PYNQ, we have devised a couple of ways. The easiest way is using our own router, connecting both PYNQ and your computer to our own network. You can then set a static DHCP address for the PYNQ based on its mac address.
Connect to the router, edit /etc/wpa_supplicant/wpa_supplicant.conf:

```
network={
    ssid="SLAMDUNK"
    psk="password"
    proto=RSN
    key_mgmt=WPA-PSK
    pairwise=CCMP
    auth_alg=OPEN
}
```

and add the new network interface to the /etc/network/interfaces file

```
auto wlx74da38930a18
allow-hotplug wlx74da38930a18
iface wlx74da38930a18 inet dhcp
wpa-conf /etc/wpa_supplicant/wpa_supplicant.conf
```

As a backup if the router fails, it is possible setup a reverse ssh tunnel to an external server to bypass NAT-rules. This is currently done using a private server with ip 35.196.88.68, forwarding port 19999 on the server to port 22 on the PYNQ.

```
ssh -f -N -T -g -R 19999:localhost:22 \
dmpro@35.195.88.68
```

2.3 Project deployment

Clone the newest version of the SLAMDUNK software, residing in the GitHub repository [2]. The Makefile residing at the root level provides deployment options; set BOARD_URI to point to the remote PYNQ host. Each subproject in turn contains its own Makefile. `make deploy` will rsync the required source code for building, while `make pynqslam` will deploy the code, and further log onto the PYNQ to build the shared library and the PYNQSLAM application. There are additional rules to control the application from a remote host, but this requires logging onto the PYNQ and running the `install.sh` script from `slamdunk/pynq`, as it will install necessary configuration to run PYNQSLAM as a Systemd service.

[2] `https://github.com/NTNU-TDT4295/slamdunk`

2.4 PCB (I/O-card)

The system uses a custom I/O-card. This card captures data from the LIDAR (and possibly other sensors) and transfers it to the PYNQ. Place this card on the PYNQ and connect the LIDAR to the card.

The microcontroller on the board must be flashed using a J-Link debugger or a similar debugger that supports SWD. Connect this to the ARM 20 Pin header on the card. The code to be flashed can be found in the GitHub repository in the `efm/` directory.

The PCB has two available power sources. The EFM is powered by the PYNQ but the LIDAR can be powered by an external source or the PYNQ. This was originally because we didn't know if the PYNQ could deliver the needed power, but it turned out OK, so the external power should not be needed.

JP9 and JP10 are the switches to change the power to the LIDAR. Take care when using an external power source. If the external power source is used to power the LIDAR scanner the voltage of the external power must be between 5V and 6V. If the external power is used only for the motor, the voltage can be between 5V and 9V.

The schematics and the layout of the card is available on the GitHub repository of the project, in the `pcb/` directory. If you want to solder a board for yourself, the bare minimum components that needs to be soldered is the following:

- EFM32GG980.
- 20-pin Debug connector.
- Molex connector for the LIDAR.
- Decoupling: Capacitors C1-C10. Ferrite bead L1. Resistor R5.
- Header pins for the PYNQ.

3 System description

3.1 SLAMDUNK system

The SLAMDUNK system is built to map out environments, using a custom built PCB meant for a Xilinx PYNQ-Z1. The following section describes the directory structure of the project.

3.1.1 build

Contains common recipes for building the modules of SLAMDUNK (e.g. makefiles).

3.1.2 efm32

Code to run MCU on the custom PCB. Functionality includes reading sensor data from RPLIDAR A1, BNO055, HC-SR04, and communicating over serial protocols such as UART, I2C and SPI.

3.1.3 libs

- **slamcommon:** Hector SLAM algorithm and common net utilities.
- **window:** library for creating X11 windows with OpenGL, and other 3D-rendering utilities.

3.1.4 pcb

Schematics for the custom PCB shield.

3.1.5 pynq

General utilities used for deployment to PYNQ. e.g. kernel modules for Wi-Fi and virtual serial port on USB, and other scripts to get started.

3.1.6 pynqslam

The actual SLAMDUNK implementation. All functionality related to running the Hector SLAM algorithm on the PYNQ resides there. This includes an FPGA implementation of an SPI slave, with double buffering of the incoming data, as well as the entire implementation required to run SLAM on a PYNQ. Also contains compiled bit files needed to deploy to FPGA fabric.

3.1.7 slamvis

Software running a remote machine for visualizing the output from pynqslam.

3.1.8 tools

Utilities for testing etc., mainly used in previous iterations of the system:

- **lidar:** visualize the raw output from a LIDAR.
- **lidar_usb:** receive RPLIDAR output from USB.
- **pointcloud:** legacy program for visualizing 3D pointclouds.
- **simulator:** legacy program for simulating LIDAR data.
- **slam:** alternative implementation of pynqslam for running on non-PYNQ machine.

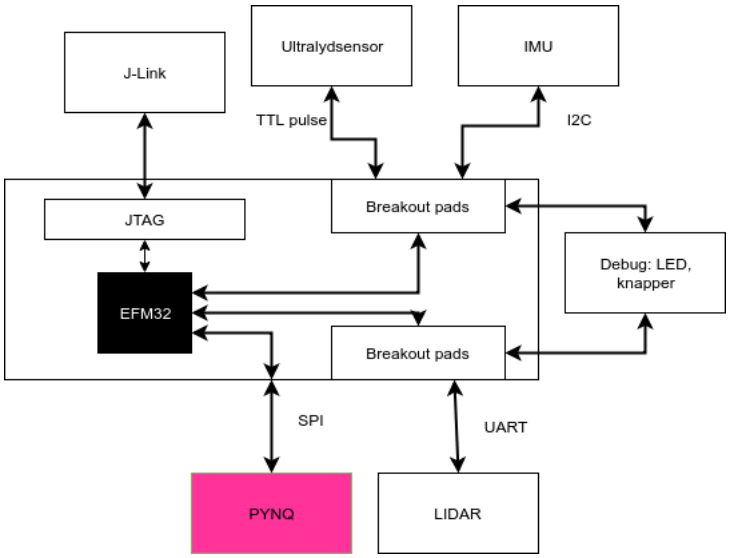

Figure 1: System overview

3.2 Communication

The SLAM algorithm communicates with the visualization host via a custom made protocol. This protocol provides means of updating the visualization of the map that is currently displayed. In addition, it also supports live visualization of the raw output from the LIDAR sensor. The map can be updated in two different ways: either full synchronization of the map or subdivided into tiles. For a more detailed description of the protocol, refer to the source code [3].

3.3 SLAM algorithm

The SLAMDUNK SLAM-algorithm is a C++ implementation of the core concepts of Hector-SLAM, an odometry free, 3DOF (three degrees of freedom) SLAM-algorithm developed at Technische Universität Dramstadt[4]. While Hector-SLAM was written using ROS (robot operating system) as middle-ware, our algorithm works independent of any third-party software.

The algorithm preforms computations based on data from movement in 3DOF, but is able to handle movement in 6DOF by filtering data based on the z-axis, only using points within a desired plane. This lets our system handle movement that occurs when the device is carried by hand.

The core function of the algorithm is computing the most probable transformation between the set of previously registered points and a new set of points in

[3] `https://github.com/NTNU-TDT4295/slamdunk/blob/master/libs/slamcommon/src/slam_vis_net.h`

[4] `https://github.com/tu-darmstadt-ros-pkg/hector_slam`

a sensor reading, effectively aligning the two set of points. The observed points are represented by an occupancy grid, functioning as a 2D-map. A cell in the occupancy grid can be set as either occupied or not occupied, this is determined by the probability of a point observed within the area represented by the cell being a correct observation. This probability is found by using bilinear filtering with its closest neighbouring points. As shown in figure 2 the map is only updated if the pose change between the points in the new reading and the map is sufficiently large.

Since the algorithm uses a gradient approach to mapping it has an inherent risk of getting stuck in a local minima, the way this is solved by iterating over and matching the scans with multiple maps of increasing resolutions. The resolutions and number of maps can be varied as well as the number of iterations over each map. In our implementation we have used three maps with the resolutions shown in figure 3.

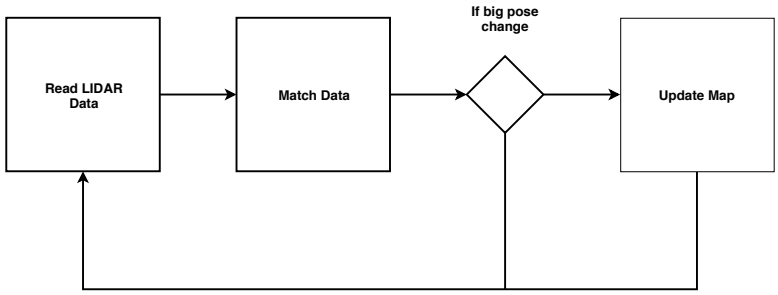

Figure 2: Hector-SLAM control flow. The map is only updated if the pose change when comparing a new LIDAR reading to the current map is sufficiently big.

3.4 System limitations

The current system does not fuse the SLAM algorithm with the orientation data from the BNO055 IMU residing on the custom PCB. This limits the maximum angular velocity of which a user will be able to turn the system without desynchronization within the algorithm.

The maximum size of the occupancy grid is limited to a fixed size set at compile time. The default configuration of 1024 x 1024 pixels with a resolution of 0.025m per pixel results in a maximum area of 26 x 26m. The system has been tested with a configuration of 2048x2048, which worked seamlessly.

The system does not currently support persistence of the scanned map, which means the system is unable to resume from a previously obtained state.

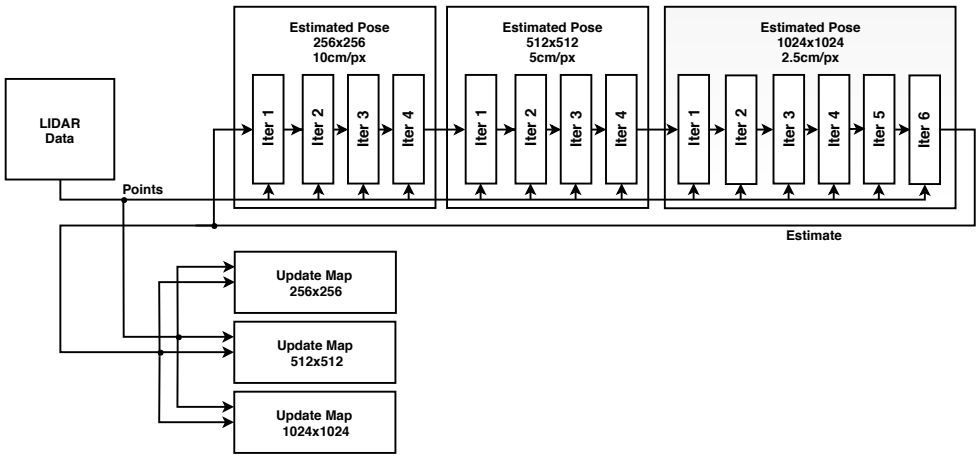

Figure 3: Hector-SLAM Data Flow. This figure shows the data flow in the Scan Matching step of the algorithm. A new data reading is iteratively matched with representation of increasing resolution of the current map. The last map size is matched against two extra times to ensure increased accuracy.

3.5 Potential improvements

By utilizing the IMU data, as explained to be a system limitation, it is possible to make the system as a whole more tolerant to rapid changes of orientation. Another improvement would be to implement a dynamic tiling system for the occupancy grid, as to allow for scaling of the resulting map, such that it resembles minimaps implemented in many video games. Adding support for saving the current state of the system could be used as a check point generator, in case of system failure. In addition, such serialized maps could be used to further extend the system such that it is able to detect its own position within a previously mapped room.

4 Printed Circuit Board

4.1 PCB Design

The main task of the PCB is to take in data from sensors and send it to the PYNQ. Originally the implementation was supposed to use ultrasound sensors and an IMU in addition to the LIDAR. To accommodate for all of these sensors, the PCB has a large range of input pins. Some of them are specific, while others are GPIO.

The PCB communicates with the PYNQ through SPI and has three predefined pins for this task. In addition to the pins, there are four buttons and four LEDS on the PCB. Two buttons and two LEDs are directly connected to the MCU, while the rest are connected to pins going into the PYNQ.

Figure 4.2.5 shows the schematic for the PCB.

4.2 Components

4.2.1 MCU

The main component on the PCB is the EFM32GG980F1024. Its primary job is to read from the sensor pins and transfer the data to the PYNQ through SPI. However, if needed, it can also be used to send IMU data.

4.2.2 Compiling

For getting code unto the EFM32, we utilize the open source project efm32-base [5]. The author of the project describes it as: "this base project is designed to provide a quick and platform independent method of building projects for Silicon Labs EFM32 microcontrollers." It is a unix CLI tool which includes an embedded ARM Compiler/Toolchain for compiling code, and JLink tools for flashing code to the microcontroller.

4.2.3 Drivers

The source code for our drivers can be found in the efm/source directory. This includes:

serial.c implements UART, I2C and SPI communication for the EFM32, setting up the necessary buffers and interrupts, and provides a serial communication interface to other drivers.

lidar.h communicates with the LIDAR using UART-functions from serial.c. It includes functions sending the correct setup, reset and debugging of the LIDAR.

setup.c Setup of general EFM32 clocks, and provides delay functions.

gpio.c Setup of GPIO such as pins and register callbacks.

main.c Main program loop being flashed to the EFM32.

imu.c Currently unused code using I2C to talk to the BNO055 IMU chip. It includes setup and gathering of data.

sonar.c Currently unused code communicating with a sonar sensor.

[5] `https://github.com/ryankurte/efm32-base`

4.2.4 LIDAR

The LIDAR used in our system is the RPLIDAR A1 fram Slamtec [6]. This is a 360 degree range scanner with a range of 12m. The LIDAR communicates with the MCU serially over UART. The motor speed of the LIDAR can be configured with PWM, but is currently set to run at the default frequency of 5.5Hz. Scan initialization is done from the MCU, and the LIDAR's standard mode is to send each sample in a burst of five bytes, each containing distance, angle and a measure of the quality of the sample. For further specification of the protocol, consult the data sheet [7].

4.2.5 GPIO

The PCB was designed with scalability in mind. Therefore there are several GPIO pins on the board. The GPIO pins are equipped with level-shifters to improve compatibility with sensors.

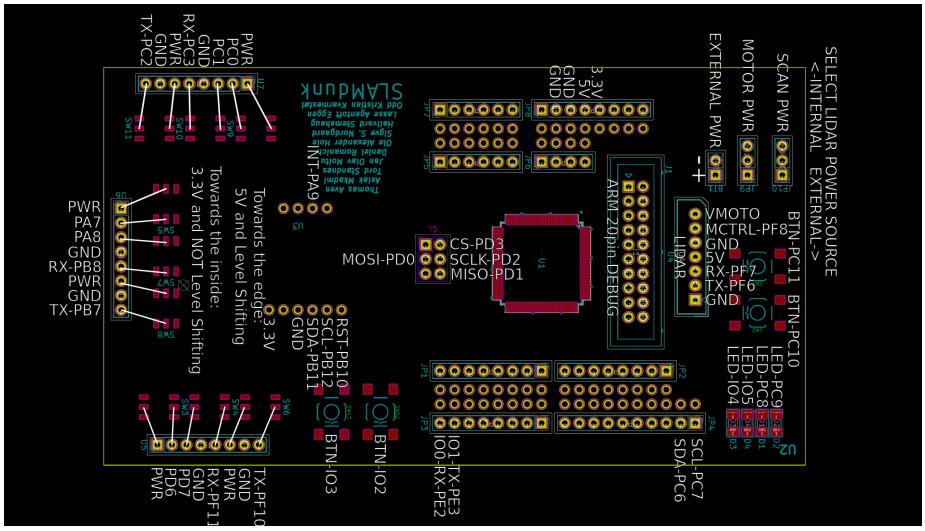

Figure 4: Schematic of the SLAMDUNK PCB.

5 FPGA components

There are several reasons not to partially or fully accelerate the algorithm in FPGA.

[6] http://www.slamtec.com/en/lidar/a1

[7] http://bucket.download.slamtec.com/b42b54878a603e13c76a0a0500b53595846614c6/LR001_SLAMTEC_rplidar_protocol_v1.1_en.pdf

- Parts that are embarrassingly parallel reside deep in the algorithm pipeline. Extracting these parts is likely to incur performance penalties greater than the gains from parallelization. It demands moving data with uncertain latencies.
- The algorithm is inherently iterative. There are loop-carried dependencies in several stages, e.g. the iterative nature of the estimations within the occupancy grid, which base their estimates on previous estimates. The inherently sequential nature of this algorithm restricts freedom in parallelization.
- Storing three sparse matrices for the maps would require about 10 MB for 256, 512 and 1024 square maps. That's about 20 times the amount of low-latency BRAM in the FPGA. Due to this restriction a cache hierarchy needs to be implemented to compete with the cache architecture of the CPU.
- NEON, ARM's SIMD instructions (e.g. `vmul`, `vadd`, `vsub`, etc.), makes floating-point operations up to four times faster on the CPU. Utilizing this feature further reduces the incentive to implement any part of the algorithm in FPGA.

5.1 SPI

The MCU is the master in the SPI setup. Due to data being one-way transmitted, from the MCU to the PYNQ, the SPI hardware only needs to map to three external pins as input: `mosi`, `clk` and `ss`. The SPI module samples the input signals and handles potential metastability issues with shift registers for all inputs. Effectively this results in three-bit shift registers for `clk` and `ss` and a two-bit shift register for the `mosi` signals. This makes it possible to detect clock rise and chip select which makes it trivial to read data on the `mosi` line.

5.2 Double buffering

To seamlessly synchronize with the CPU-run SLAM algorithm, complete revolutions of LIDAR data are stored in the BRAM. Each revolution is stored as a separate contiguous memory entity. Each new revolution a bit is flipped to signal which buffer is currently valid.

5.3 Preprocessing

The quality of each received data point is checked by inspecting the first byte of each sample, as specified by the LIDAR protocol. The sample is discarded if it doesn't meet expected quality.

5.4 Computing sin and cos

An implementation of the CORDIC algorithm was made to compute the sine and cosine of the angle of the LIDAR measurements. This module is not currently integrated with the system.