Development of a Realtime Control Framework for DIMA

Prepared by:

Vikyle Naidoo

NDXVIK005

Department of Electrical Engineering
University of Cape Town

Prepared for:

Dr. Amir Patel

Department of Electrical Engineering
University of Cape Town

November 2020

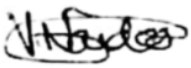
Submitted to the Department of Electrical Engineering at the University of Cape Town in partial fulfilment of the academic requirements for a Bachelor of Science degree in Bsc Eng (Mechatronics)

Key Words: *realtime, robotic, control, PREEMPT_RT, linux*

Declaration

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.
2. I have used the IEEE convention for citation and referencing. Each contribution to, and quotation in, this final year project report from the work(s) of other people, has been attributed and has been cited and referenced.
3. This final year project report is my own work.
4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as their own work or part thereof

Name: **VIKYLE NAIDOO**

Signature: 

Date: 11 November 2020

Acknowledgements

I would like to express my gratitude towards my parents who have fed and sustained me since birth.

A special thanks goes to Dr. Amir Patel who has provided invaluable help and guidance in completing this project.

I would also like to pay my respects to all my lecturers, teachers, and Gurus who have imparted their knowledge and wisdom to me thus far.

Lastly I would like to thank Nedbank for sponsoring the entire four years of my studies.

Abstract

The DIMA robot emulates the movement of a cheetah, by improving maneouverability with the use of a tail action. There is an existing control system running on an STM32F4, but a new control system needs to be designed to exploit the processing power of the Jetson Nano in analysing and processing the gathered sensor data. The objective of this project is to develop a control system framework for the DIMA robot, using an existing IMU+GNSS sensor board and an NVIDIA Jetson Nano board for data processing and logging, as well as provide a means of remote telemetry for the user.

An investigation into existing methods of implementing realtime control systems was conducted by means of a literature review, where the various methods were analysed and compared in order to decide the most suitable implementation.

The design of three separate subsystems aimed to produce a modular system. A custom firmware was developed for the sensor board. The Jetson Nano was pathced with PREEMPT_RT to operate with real time capabilities. Wireless communication to a host PC was implemented to allow human input commands to be sent to the control system remotely, using an XBEE PRO S2C. The framework used a servo driver (Pololu Micro Maestro) to interface with the existing hardware of the robot.

Testing was performed through various experiments and benchmarks, to evaluate if the system met the specifications created earlier. The results showed that although the system achieved realtime performance, the speed of the control loop needed to be reduced to compensate for the lack of performance from the servo driver.

Table of Contents

Table of Figures

1. Introduction

1.1 Background to the study

The DIMA robot emulates the movement of a cheetah, by improving manoeuvrability at high speeds with the use of a tail action. Realtime control is a common theme in high performance robotics, with the purpose of achieving fast and reliable control systems.

1.2 Objectives of this study

1.2.1 Problems to be investigated

Develop a control system framework for the DIMA robot, using an existing IMU+GNSS sensor board and an Nvidia Jetson Nano board for data processing and logging. Firmware would need to be developed for the sensor board, and the Jetson Nano needs to operate with real time capabilities, and wireless communication to a host PC needs to be implemented to allow human input commands to be sent to the control system.

1.2.2 Purpose of the study

There is an existing control system for DIMA, running on an STM32F4 with FreeRTOS installed, but a new control system needs to be designed to exploit the processing power of the Jetson Nano.

1.3 Scope and Limitations

The focus of this project is only on developing the control system framework required to implement a controller for the DIMA robot. It is limited to gathering and logging the sensor data, providing means to actuate the wheel and tail motors, and providing a means of remote operation and communication to the system. **The design and implementation of a working feedback controller is not included in this study**.

The project required the use of an NVIDIA Jetson Nano as the main control computer. The project also required the use of an existing sensor board with STM32F4 microcontroller, developed by Do Yeou Ku [1].

1.4 Plan of development

The Literature Review section investigates previous methods of implementing control frameworks, and compares the various options available for achieving real time performance.

The Methodology section outlines how the project was undertaken, and presents the user requirements and specifications. It describes the approach taken to design the system, and how the specifications were tested to evaluate the system.

The Design & Development section describes in detail the design of the system and its subsystems, as well as the practical work that went into developing the system.

The Testing & Results section describes the testing procedures used to evaluate the system specifications, and presents the results thereof.

The Discussion of Results section discusses the results of the testing phase, and provides commentary on the outcomes and how it affects the system.

The Conclusions & Recommendations section relates the results to the initial problem statements, and suggests how further work on this system may be conducted.

The Appendices contain extra information that is useful or relevant but wasn't included in the main body of the report.

2. Literature Review

The previous version of DIMA was run from a STM32F4 MCU running FreeRTOS as a real time system, in order to control the motion of the robot, as well as receive remote telemetric commands, and log data.

In order to improve the performance and robustness of the system, a Jetson Nano was used which provided more computational power and additional hardware such as GPU to be available for online data processing in the future. The Jetson Nano is running a GNU/Linux OS call Linux4Tegra (v32.3.1), using kernel version 4.9 based on Ubuntu (18.04)

Through investigation of previous systems and research, this section aims to answer following questions:
-what are existing/commonly used control frameworks for robotic control systems?
-what is meant by real time requirements in robotic control?
-what is the best method of implementing a realtime OS?

2.1 Control Frameworks for Robotic Systems

There are various approaches to implementing a framework for robotic control systems. In this subsection, some previous existing frameworks are investigated, and common trends among them are explored.

2.1.1 Control System for ATRIAS

The first system that was investigated was designed for the control of a bipedal robot, ATRIAS [2]. The objective of this system was to achieve a hard real-time, and high frequency control solution, using freely available open-source software [3]. Figure 1 provides an outline of this system. An emphasis was placed on the modularity, therefore the system was divided into three parts: physical, control, software.

Linux (Xubuntu) was the operating system used, due to the being open source, with a large community, and support for most software. In order to achieve real time performance, Xenomai was used with the Linux based OS. Robotic Operating System (ROS) was a robotics framework used to facilitate the communication between the robot and a host computer. Open Robot Control Software (OROCOS) was used to generate components to enable real time operations. A GUI based user application was also used to remotely control the robot.

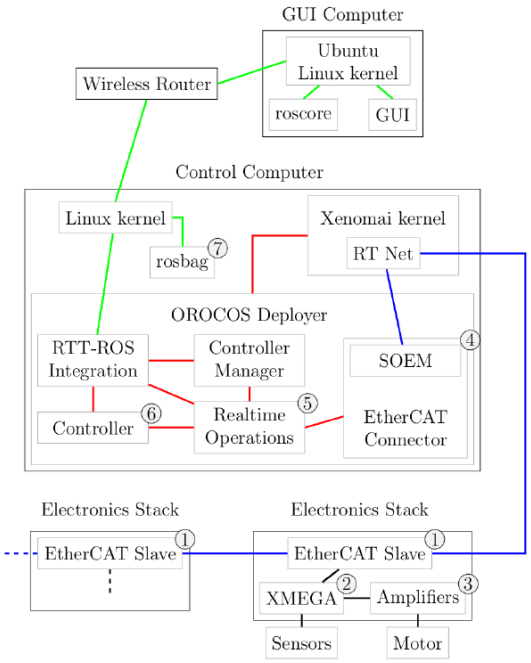


Figure 1: Outline of software structure for ATRIAS control system

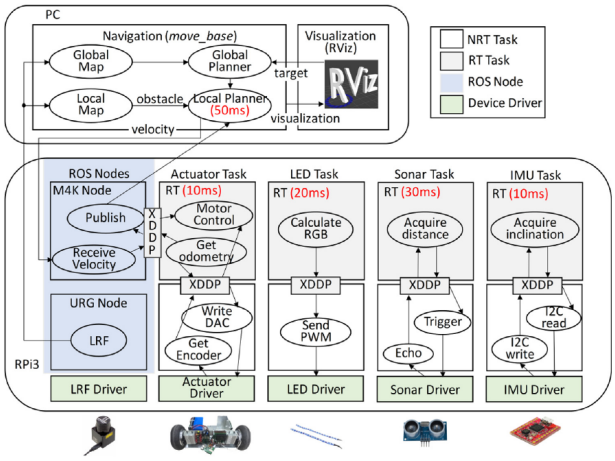


Figure 2: outline of real time control architecture for controlling MK4

2.1.2 Control System for a Service Robot

The aim of this system, was to develop a real time control architecture with hard real time capabilities, to control a telepresence robot called M4K [4] [5].

The environment consisted of a Raspberry Pi 3, running a Linux OS (Raspbian). Xenomai provided realtime performance. ROS was the robotic framework used for performing navigation operations. Independent threads were dedicated to performing realtime tasks such as actuation, and sensing. Non realtime tasks communicated with the realtime tasks using a communication mechanism called cross-domain datagram protocol (XDDP). A user PC was also used control and visualise the motion of the robot. Figure 2 outlines the control framework for this system.

2.1.3 Control System for Unmanned Autonomous Helicopters

The purpose of this control system was to develop a control architecture for the Kyosho RC Helicopter with hard realtime performance [6].

The approach used here was a four layered architecture. The hardware layer consisted of sensors, motors etc; the execution layer ran the real time control tasks; the remote interface layer allows the user to interact with the system; the service agent layer facilitated the state transition of the system based on

information received from the remote layer. Figure 3 outlines these layers. The real time capabilities of this system was implemented using RTLinux.

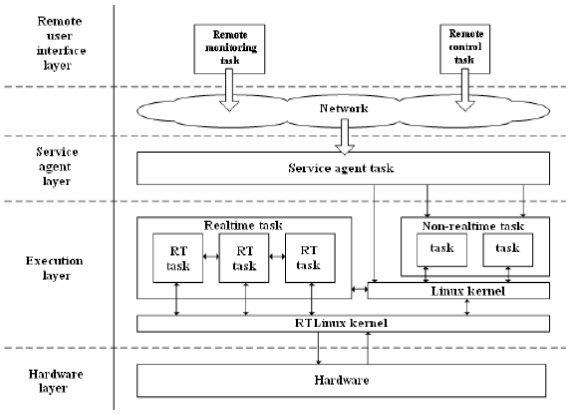


Figure 3: four layers of the UAV control system architecture

2.2 Real Time Requirements

There are two commonly used approaches when it comes to implementing realtime performance on Linux. These are a dual kernel approach (Xenomai), and modifying the Linux kernel itself (PREMPT_RT patch).

2.2.1 what is meant by real time requirement

A real time requirement simply means there is a *deterministic* latency from an event to a response [7]. This means that a task will respond within a guaranteed time specification, to an event. It is desirable to have a low latency as possible, but more important is the deterministic aspect of the latency; that is to say if our system responds within the acceptable latency specification we define, then our realtime requirement is realised.

Acceptable jitter and latency

Jitter can be described as the maximum variation in latency of the process [7]. As such it is an important metric to consider when evaluating deterministic latencies.

Scheduling and interrupt metrics

The scheduling policy of an operating system decides the order in which to run concurrent processes, depending on their priority. As such, the latency of a process is affected by the scheduling policy. Real time operating systems, in contrast to a general purpose OS, use an explicitly unfair scheduling policy, which guarantees that the highest priority process will run first [7].

Preemption is the mechanism of a higher priority process interrupting a lower priority process. This is the quintessential characteristic of a realtime

scheduler. The granularity of preemption (size of “time slices” the scheduler can see) affects the latency, with finer grained resulting in smaller, more predictable latencies [7].

Priority inversion
Priority inversion is a serious issue that can plague realtime systems, resulting in latencies, and sometimes deadlocks. It occurs when a high priority task is indirectly waiting on lower priority task, hence the term priority inversion. A classic example of this is when a high priority task requires a resource from a low priority task, but a medium priority task interrupts the low priority task, thus resulting in the high priority task indirectly being preempted by the medium priority task [8].

2.2.2 Xenomai3

Xenomai takes the dual kernel approach, with a higher priority xenomai kernel running alongside the linux kernel. The xenomai layer communicates with the hardware using the ADEOS I-Pipe system [9]. The higher priority of the xenomai kernel means that any real time threads will have the opportunity to respond before the linux kernel itself, and similarly, any blocking operation doen by the linux kernel will not have an effect on the xenomai threads.

There are two possible approaches to implementing Xenomai 3, as discussed below.

Dual kernel (cobalt)
This is the traditional dual kernel approach associated with Xenomai. The Cobalt core has a higher priority over the native linux kernel, and can be used to handle tasks such as realtime scheduling, interrupt handling, and any other time critical activities [10].

Single kernel / native linux (mercury)
Mercury is a newer option made available in Xenomai 3. Its approach is completely different to the dual kernel, in that it aims to exploit the realtime capabilities of the linux kernel itself. As such, it is based on the PREEMPT_RT patch, and allows non POSIX Xenomai APIs to be used [10].

2.2.3 PREEMPT_RT patch

The second major approach to implementing realtime systems in Linux is to make the Linux kernel itself more capable of realtime tasks.

The PREEMPT_RT patch does this by replacing the the *spin locks*, inherent in the normal kernel, with *sleeping locks,* except for some extremely critical sections of the kernel, where some spin locks are kept. Since spin locks do not rely on the scheduler while waiting, it means preemption (and consequentially realtime operation) is not possible in the system. By replacing the spin locks

with sleeping locks, it allows the possibility of preempting any lower priority tasks [7]. The issue of priority inversion is solved by priority inheritance [8].

By applying the PREEMPT_RT patch, one is able to use normal Linux and POSIX APIs while achieving realtime capabilities. All user tasks and IRQ handlers need to be implemented in threads, which are preemptable, and assigned a priority.

2.2.4 Comparison

Knutson [7] reported that there was minimal difference in performance between the realtime patch and Xenomai approaches, with Xenomai having a marginally better interrupt latency of 247μs compared to the 273μs for PREEMPT_RT. However the RT patch performed better when it came to scheduling latency at 153μs compared to Xenomai's 271μs, according to Knutson.

An experiment conducted by Brown & Martin [9] compared the interrupt latencies and jitters of the two approaches.

Config	Experiment	# samples	Median	95%	100%
stock	linux-chrt-user	1018391	-2μs	69μs	1205μs
rt	linux-chrt-user	1018387	-1μs	47μs	158μs
xeno	xeno-user	1018318	-1μs	34μs	57μs
stock	linux-kernel	1018255	0μs	17μs	504μs
rt	linux-kernel	1018249	0μs	24μs	98μs
xeno	xeno-kernel	1018449	-1μs	23μs	41μs

Table 1: Comparison of jitter, relative to expected time period between successive falling edges, for stock linux , RT patched linux, and Xenomai. [Brown, Martin]

Config	Experiment	# samples	Median	95%	100%
stock	linux-chrt-user	1840823	67μs	307μs	17227μs
rt	linux-chrt-user	1849438	99μs	157μs	796μs
xeno	xeno-user	1926157	26μs	59μs	90μs
stock	linux-kernel	1259410	7μs	16μs	597μs
rt	linux-kernel	1924955	28μs	43μs	336μs
xeno	xeno-kernel	1943258	9μs	18μs	37μs

Table 2: Latency from input GPIO change to corresponding output GPIO change for stock, rt patched, and xenomai approaches [Brown, Martin]

Looking at the linux-chrt-user and xeno-user lines, it is evident that the xenomai approach does in fact offer significantly better performance when it comes to interrupt latency and jitter. These metrics are of course dependant on the hardware used, and will be needed to be tested for the specific platform that will be used.

2.3 Summary

An investigation, on popular implementations of achieving real time performance from a real time system, was conducted in order to answer some key questions posed earlier. The conclusions drawn from this investigation is summarised here.

Due to the constraints on this project, it was decided to use a Linux operating system, running on the Jetson Nano; the advantages being driver support from NVIDIA, and an active open source community. The project will break the system into modular subsystems, which will be able to operate independently and communicate with each other.

Predictability is the central philosophy around creating realtime systems. A task needs to respond within a guaranteed period of time. The performance of real time control can be evaluated by two important metrics: latency and jitter.

Two popular methods of implementing realtime capabilities in a Linux environment is the dual kernel Xenomai approach, and the PREEMPT_RT patch on the Linux kernel itself. While Xenomai has undoubtedly better real time performance, the performance offered by the PREEMPT_RT patch is sufficient for this project, which only requires a 250Hz control loop. Furthermore, the RT patch allows the use of standard POSIX APIs which removes the added complexity in developing with an unfamiliar Xenomai API. For these reasons it was decided to make use of the PREEMPT_RT patch for this project.

3. Methodology

This project consists of three modular subsystems: The sensorboard, the Jetson Nano control framework, and the remote pc user application. This chapter aims to describe how the design and implementation processes were undertaken, and how the system was tested to meet the requirements.

3.1 User Requirements and Specification

The main stakeholder, and end user in this project was Dr. Amir Patel (the supervisor). Through meetings and discussions, the following user requirements were obtained, and specifications were derived.

3.1.1 User Requirements

(1) Realtime operation for a 250Hz control loop.
(2) Ability to read IMU sensor data fast and accurately enough for the control loop.
(3) Ability to read GNSS data.
(4) Ability to log data for viewing/processing later.
(5) Wireless/remote user operation, with Graphical User Interface.
(6) Ability to interface the new system with the existing hardware of the car and its motors.

3.1.2 Functional Specifications

(1) Real time specifications
  (1.1) latency of less than 10% of the realtime period. ie latency<400μs
  (1.2) jitter less than 10% of the real time period will yield acceptable peformance.

(2) IMU sensor data specifications
  (2.1) Read IMU data @ 250Hz
  (2.2) less than 1 out of 100 packets of invalid data

(3) GNSS data specifications
  (3.1) read GPS data @ 10Hz

(4) Data logging specifications
  (4.1) log data to the Jetson Nano filesystem

(5) Remote operation specification
  (5.1) can send and receive data wirelessly to a remote GUI application.

(5.2) can reach 50m line of site operation, with less than 1s of no communication.

(6) Motors interfacing specifications
(6.1) PWM output pulse length can achieve the full range of 1ms to 2ms.
(6.2) PWM output can operate at a frequency of 250Hz.

3.2 Design Process

The majority of this project was focused on software and embedded firmware development. As such common software design methods were utilised. For each subsystem, a use case diagram was created, which ensured that the user requirements were sufficiently addressed. Flow charts were then created to provide a guideline for implementing the code.

Weekly meetings with the supervisor (end user), presenting the progress, helped to ensure that the user requirements were correctly understood. If a modification was necessary, the specification was remade to match the updated user requirement, and the design was consequently modified. This cyclical design approach ensured that the end product never diverged too far from the end user's expectation.

After the completion of each subsystem, tests were conducted to verify that the functional specifications were met.

3.3 Testing and Evaluating Procedures

Various tests were conducted, each with the aim of evaluating one or more functional specifications.

Specification	**Testing Procedure**
1.1	Latency will be measured using an open source tool called cyclictest.
1.2	Jitter will be measured by toggling a GPIO pin and viewing the maximum deflection of the waveform on an oscilloscope.
2.1	Data will be requested every 4ms from the sensor board. If the received data is valid at this speed, by verifying the checksum, the test is passed.
2.2	A counter will keep track of how many data packets are invalid (checksum incorrect), and valid, for at least 100000 packet requests. The ratio of invalid packets to valid packets can thus be determined.
3.1	Will read and log the GPS data from the data packet and if new data is present every 10ms the test is passed.

4.1	If the logged data can be opened and read, this test is successful.
5.1	If the user can send and receive data between the Jetson and a remote pc/laptop, without any wires connecting them, the test is passed.
5.2	The user will attempt communication at increasing distances from the Jetson, up to 50m. The system will timeout if connection is lost for more than 1s.
6.1	The signal will be connected to an oscilloscope to evaluate the range of pulse lengths.
6.2	The signal will be connected to an oscilloscope to evaluate the frequency of the signal.
7.1	Each subsystem will be run without communication to the other subsystems. It should not crash.

4. Design & Development

4.1 System Overview

In order to create a modular design, the system is divided into 3 subsystems, which should function independently. The block diagram in Figure 4 provides a high level overview of the three subsystems and how they communicate/interface with each other, as well as the hardware of the robot.

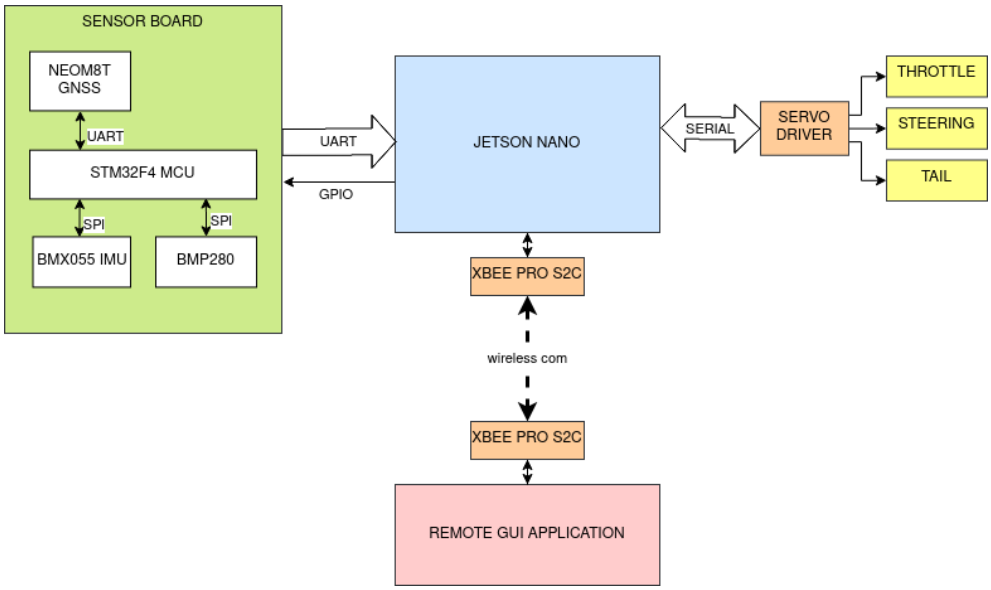


Figure 4: block diagram outlining top level system overview

4.2 Developing Sensorboard Firmware

In order to receive data such as IMU and GPS measurements, an existing sensor board developed by Do Yeou Ku [1] was used.

The board contains the following significant components:
-Inertial Measurement Unit (BMX055)
-GNSS Module (NEO-M8T)
-Barometer (BMP280)
-Microcontroller (STM32F4)

It was originally developed for use as part of a motion capture system, however the existing firmware was unsuitable for use in this project, so a new firmware needed to be developed, that is better suited for this system.

The firmware was written in C, using Atollic TrueStudio 9.0.0 IDE.

4.2.1 Firmware Design

The use case diagram in Figure 5 outlines how the STM32F4 microcontroller interacts with the other sensor modules on the board, and the Jetson Nano.

Each sensor module will be configured by writing to the relevant configuration registers. When requested, the data from the sensors will be read, and stored in a global data struct. The Jetson may request data by toggling a GPIO line, which will trigger the STM32F4 to send a data packet containing the sensor data to the Jetson.

The flowchart in Figure 6 outlines the program flow of the sensor board firmware.

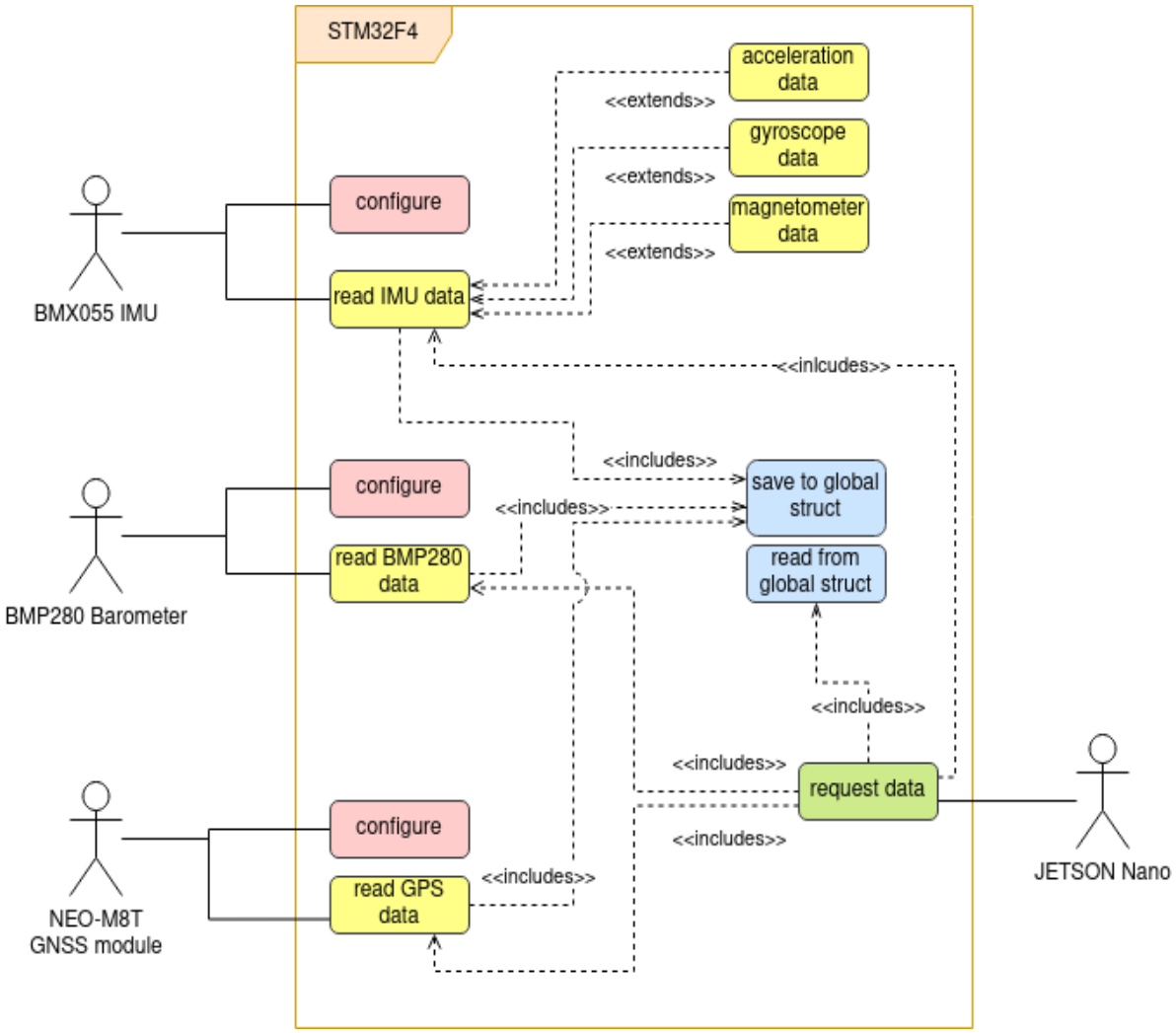


Figure 5: Use case diagram of the STM32F4 firmware for the sensor board

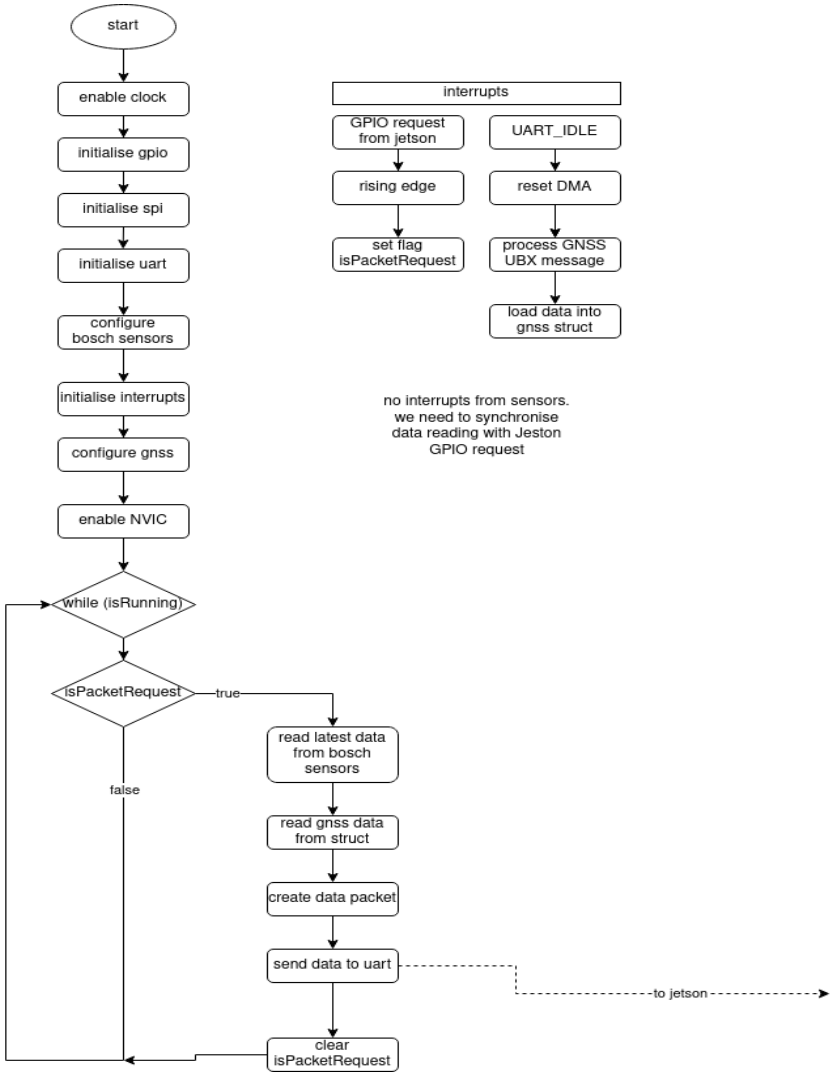


Figure 6: Sensorboard firmware flowchart

4.2.2 Communication Protocols

Serial Peripheral Interface:

The SPI protocol was used to communicate with the BMX055 and BMP280 chips. The following pins were used:

Name	**Pin on MCU**
SCK	PA5
MISO	PA6
MOSI	PA7
CS_Acc	PE7
CS_Gyro	PE8
CS_Mag	PE9
CS_Baro	PE10

The SPI was configured as follows:

Mode:	Full duplex
CPOL:	1
CPHA:	1
Master:	STM32F4
Slaves:	Gyro, Acc, Mag, Baro (4 total)
Datasize:	8 bits, MSB first
SCK rate:	500KHz

Universal Asynchronous Receiver Transmitter:
UART protocol was used to receive data from the NEO-M8T GNSS module and to send data to the Jetson Nano. The configuration is as follows:

Parameter	**UART_GNSS**	**UART_Jetson**
UARTx	USART1	USART3
Baud rate	115200	460800
Rx pin	PB7	PD9
Tx pin	PB6	PD8
DMA	DMA2	DMA1
DMA_Channel	Channel 4	Channel 4
WordLength	8 bits	8 bits
Stopbits	1 bit	2 bits
Parity	none	none

Both the UART channels were connected to a DMA channel to facilitate quick data transfer without loading the CPU.

Data is received by the MCU from the GNSS module every 10ms (10Hz).

The Jetson will request data from the STM32F4 every 4ms (250Hz) by raising an interrupt line (connected on PD10). The global data packet (132 bytes) is then sent over UART to the Jetson.

4.2.3 Global Data Packet

The global data packet contains all the information from the sensorboard which needs to be sent to the Jetson. There is a total of 132 bytes, the contents of which is detailed as follows:

Byte	**contents**		**Byte**	**contents**
1	start_token_1 ($)		17	MagY_LSB
2	start_token_2 ($)		18	MagY_MSB
3	AccX_LSB		19	MagZ_LSB
4	AccX_MSB		20	MagZ_MSB
5	AccY_LSB		21	MagHall_LSB
6	AccY_MSB		22	MagHall_MSB
7	AccZ_LSB		23	BaroPress_MSB
8	AccZ_MSB		24	BaroPress_LSB

9	GyroX_LSB		25	BaroPress_XLSB
10	GyroX_MSB		26	BaroTemp_MSB
11	GyroY_LSB		27	BaroTemp_LSB
12	GyroY_MSB		28	BaroTemp_XLSB
13	GyroZ_LSB		29-128	UBX-NAV-PVT message[1]
14	GyroZ_MSB		129-132	CRC32 checksum
15	MagX_LSB			
16	MagX_MSB			

This data was packed into a union between struct and array, and the array was sent over UART. A CRC32 checksum was calculated automatically using the dedicated CRC hardware unit from the STM32F4, and added to the data packet. The algorithm for such can be found in
Appendix A: CRC32 Algorithm (From STM32F4) .

4.2.4 BMX055 (IMU)

This device contains three sensors in one chip: a 12bit triaxial accelerometer, 16bit triaxial gyroscope, and a geomagnetic sensor. Each sensor is individually configurable, and operate independently.

Configuring this device is done by writing to the 8 bit configuration registers. All communication with the device is done the SPI protocol. There are 2 separate memory blocks, one for the accelerometer, and the other for the gyroscope and magnetometer.

The following parameters summarise the configuration options for the accelerometer

Accelerometer Configuration			
Setting name	**Setting value**	**Register Value**	**Register address**
Range	+-2g	0x03	0x0F
Rate (bandwidth)	1000Hz	0x0F	0x10
Power mode	Normal	0x00	0x11

The range determines the maximum bounds of acceleration that can be measured before saturating. The data resolution for the chosen range is 0.98mg/LSB. The bandwidth is the rate at which new data will be read by the sensor; this also effectively acts as a low pass filter since the device cannot detect data changes faster than its bandwidth. Normal mode is the default power mode the device is in after startup. In this mode it continuously reads data, without sleeping.

1 See datasheet: neo_m8t_interface_guide pg332

The remaining acceleration configurations are all left at their default setting; there was no need to modify these settings during the configuration process.

configuring gyroscope:
The following parameters summarise the configuration options for the gyroscope

Gyroscope Configuration			
Setting name	**Setting value**	**Register Value**	**Register address**
Range	+-2000 deg/s	0x03	0x0F
Rate (bandwidth)	2000Hz	0x81	0x10
Power mode	Normal	0x00	0x11

Similar to the accelerometer, the gyro range specifies the maximum data bounds before saturation, and the rate is the frequency at which new data is acquired. Normal Mode means there will be continuous data acquisition.

configuring magnetometer
The magnetometer allows a configuration choice from 4 recommended presets: Low Power, Regular, Enhanced, and High Accuracy[2]. The low power preset was chose because it had a maximum output data rate of over 300Hz, in forced mode. The device is by default in sleep mode, and enters forced mode, when commanded to, in order to acquire a data sample. It will then return to sleep mode. This differs to the normal mode operation where data is continuously acquired at an output data rate of 10Hz, which is not sufficiently fast for this application.

In order to configure the magnetometer for Low Power preset, the value 0x01 was written to register address 0x51 (for x y axes), and value 0x02 was written to register address 0x52 (for z axis).

reading accelerometer, gyro, mag
SPI was used to read from the 8 bit data registers of these devices.

The accelerometer data consists of 8bits MSB data, and 4bits LSB data for each of the 3 axes. A burst read was performed from register address 0x02 to 0x07 (6 bytes), which contain the data for the x, y, z axes.

The Gyroscope data consists of 8bits MSB and 8bits LSB for each of the 3 axes. A burst read was done from register 0x02 to 0x07 which contains the data for x, y, z axes.

2 See table37 pg122 BMX055 datasheet

The magnetometer data consists of 8bits MSB and 5bits LSB for the x, y axes (13bits); 8bits MSB and 7bits LSB for the Z axis (15bits); 8bit MSB and 6bits LSB for the Hall resistance. A burst read was performed from registers 0x42 to 0x49 which contains the data for x, y, z axes and Hall resistance.

4.2.5 BMP280 (BAROMETER)

configuring

This device contains both a barometric pressure sensor, and a temperature sensor. Configuring this device is done by writing to the configuration registers using the SPI protocol.

The device was set to “Normal mode” operation which means it is continuously acquiring new data at the set measurement rate (ODR). The oversampling settings changes the data resolution and noise. The t_standby parameter affects the output data rate.

In order to achieve the maximum possible ODR from this device (166.67Hz) , The oversampling setting was chosen as “ultra low power”, which corresponds to x1 oversampling for both temperature and pressure; this allows a resolution of 16bit/2.62Pa for pressure, and 16bit/0.005℃ for temperature. The t_standby was chosen as 0.5 ms. Table 14, pg19 of the BMP280 datasheet details the available measurement rates, depending on these options.

In order to achieve these settings, the value of (0x04 | 0x20 | 0x03) was written to the register at address 0xF4.

reading data

Data is obtained from the device by performing a burst read from address 0xF7 to 0xFC. This data consists of 20 bits each for temperature and pressure: 8bits MSB + 8bits LSB + 4bits XLSB.

4.2.6 NEO-M8T (GNSS MODULE)

The NEO-M8T is a Global Navigation Satellite System (GNSS) module with the ability to receive data at fixed time intervals. The U-Center-20.06.01 software from U-Blox was use to modify the frimware settings of this device.

GPS was the chosen GNSS constellation due to the extensive satellite coverage around South Africa. The frequency of data acquisition was set to the maximum possible of 10Hz. The module was set to automatically send the message over uart.

UBX message structure

The data was formatted according to the UBX Protocol (u-blox proprietary)[3] , outlined in Figure 7.

The first 2 chars are always 0xB5, 0x62 . Message class and ID indicate the type of message that will follow. Lenght indicates the length of the payload. The 2 byte checksum is calculated by the formula found in Appendix A: UBX checksum algorithm.

Specifically, the UBX-NAV-PVT message was chosen due to position, velocity, and heading data that it provides. Detailed structure of this message can be found in Appendix B: UBX-NAV-PVT Message structure.

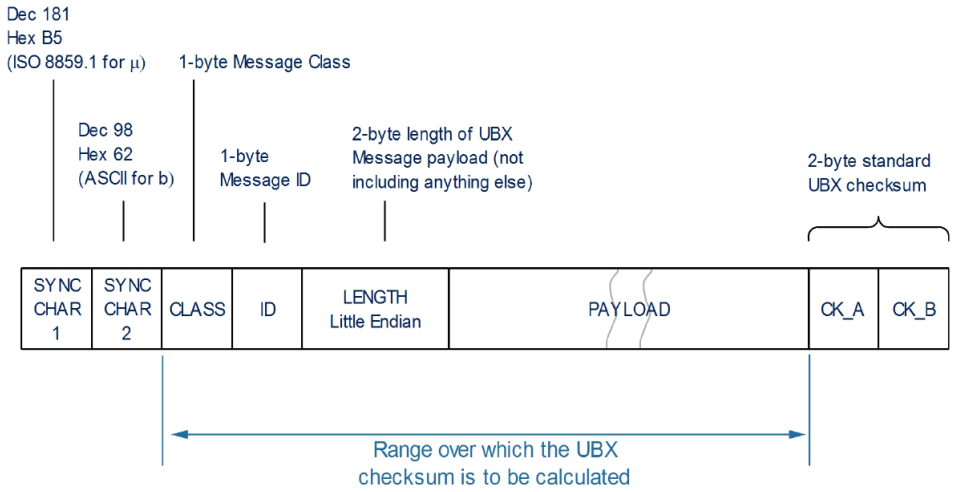


Figure 7: The structure of a basic UBX message

The STM32F4 communicates with the NEO-M8T via a UART interface, detailed under the section 4.2.2. The 100 bytes of data is automatically sent by the NEO-M8T to the UART input buffer every 10Hz., and DMA facilitates fast transfer to memory. The Global Data struct is then populated with the new data, after verifying the validity with the received checksum.

4.3 Jetson Nano Realtime Control Framework

The NVIDIA Jetson Nano revision A02 was the chosen device for implementing the control framework. A GNU/Linux OS call Linux4Tegra (v32.3.1), using kernel version 4.9 based on Ubuntu (18.04), developed specifically for use on NVIDIA embedded devices, was installed. The installations files for such may be found on the NVIDIA website[4].

3 neo_m8t_interface_guide pg143
4 https://developer.nvidia.com/l4t-3231-archive

4.3.1 Patching Linux Kernel with PREEMPT_RT

In order to realise real time performance, the linux kernel needed to be modified by applying the PREEMPT_RT patch. This required rebuilding of the Linux kernel with the new configurations.

The L4T Jetson Driver Package, L4T Sample Root File System, L4T Sources, and GCC Tool Chain for 64-bit BSP needed to be downloaded from https://developer.nvidia.com/l4t-3231-archive . The kernel build was performed on an external laptop running Linux Mint 20, kernel 5.4.0-48-generic, Intel x86_64 architecture. The detailed procedure can be found in Appendix C: Procedure for building real time patched kernel .

4.3.2 Developing Control Framework

The control framework was written in C and compiled using gcc 7.4.0 .

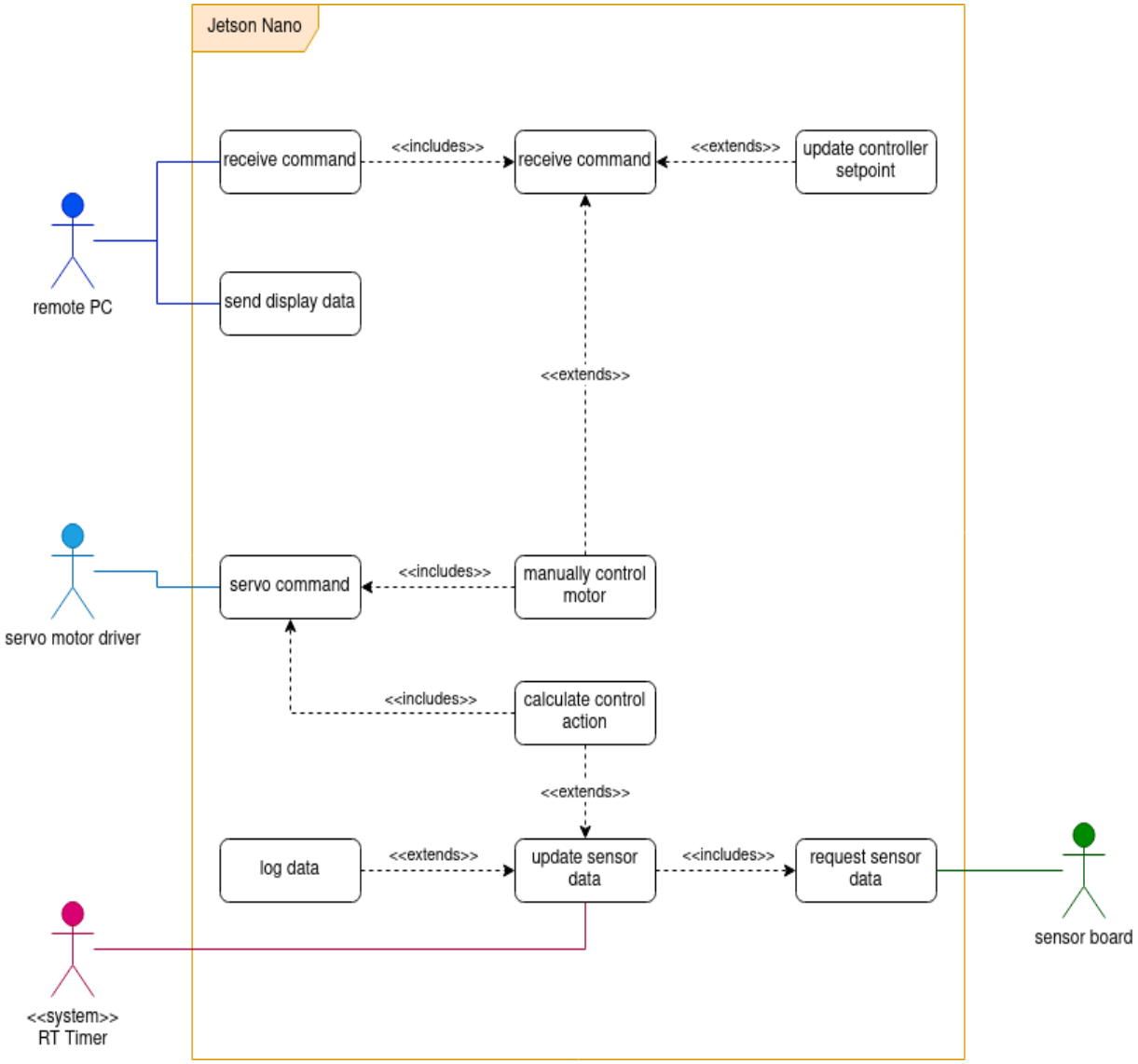


Figure 8: Use Case Diagram of Jetson Nano control framework

The use case diagram in Figure 8 demonstrates the high level overview of how the system interacts with its actors, and the main events that are to be performed.

A realtime system timer will trigger a thread to update the data from the sensors at a fixed rate. In order to ensure synchronisation, the data is requested from the sensorboard by toggling a GPIO line. The remote PC application can send certain control commands to the Jetson system, and receive certain data to display for the user. The system interfaces with the servo motors via a pwm driver, Pololu Micro Maestro.

The flowchart in Figure 9 outline the program flow and logic implemented in the system.

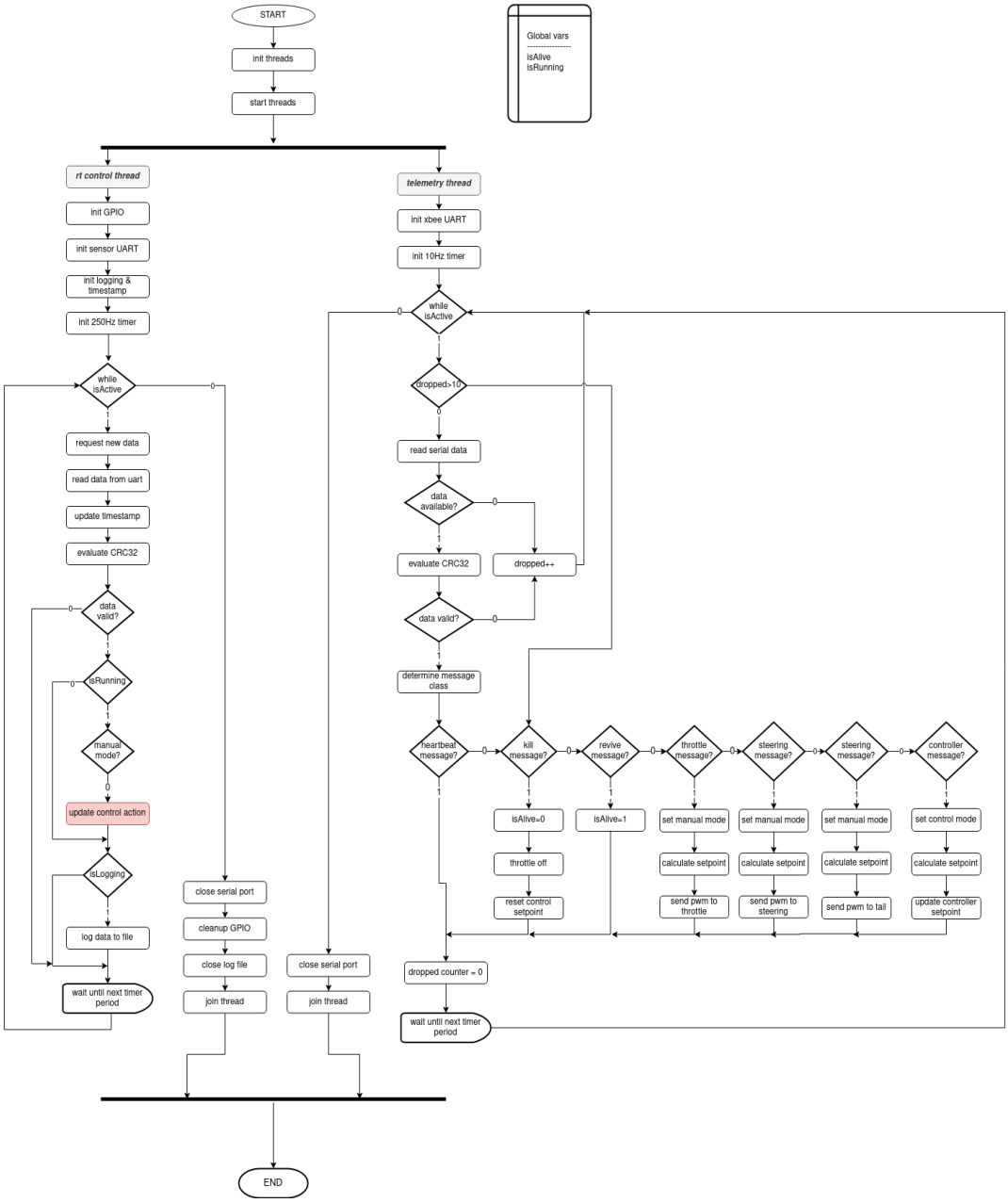


Figure 9: Flow chart of Jetson Nano realtime subsystem

Design of a realtime system in Linux requires all realtime processes to be run as threads. As such, there are two threads central to the design of this system. The high priority control thread (priority 98) is used to request and update data from the sensor board, process and log the data, and calculate the new control action to be sent to the motors. The timer is set to trigger every 4ms (250Hz). Data is requested by toggling the GPIO line connected to the sensorboard, and data is received via a UART connection. The integrity of the data is validated using the CRC32 checksum from the last 4 bytes in the data packet [5]. The algorithm for calculating the CRC32 checksum can be found in Appendix A: CRC32 Algorithm (From STM32F4) . If the data is correct, it is logged, with a timestamp, to the filesystem, and the control action will be updated[6].

A lower priority telemetry thread is used for communicating with the remote PC application. This thread both receives commands from the pc user app, and sends certain onboard data to be displayed to the user. The commands received are then processed accroding to the message type, where it will either manually update the throttle/steering/tail, or set a controller setpoint for speed / heading, or killed or revived.

A flag is used to determine if the car has been killed or revived (killed state essentially disables all motor outputs). Another flag determines whether the system is in manual or control mode. Control mode means the controller will update the motor outputs automatically according the the setpoint and sensor data, whereas manual mode means the motors are directly controlled by the user inputs.

The data structure of the received commands message is as follows:

Name	$	type	data	crc32
bytes	1	1	4	4

Total message size: 10bytes
start token ($) to identify beginning of the message.

‘Type’ indicates what kind of message is being sent:

Message type	0	1	2	3	4	5	6	7
description	Heart -beat	Kill	Revive	Throttle	Steering	tail	Speed setpoint	Heading setpoint

the CRC32 checksum used in this message is slightly different to the one from the STM32, as it it based on the RFC 1952 specification. The algorithm for

5 The format of the data packet is detailed in section 4.2.3
6 Implementation of a controller and calculation of the control action is out of the scope of this project.

calculating this checksum is found in
Appendix A: CRC32 Algorithm (RFC 1952) .

The display data sent to the remote PC application follows the following format. Total size is 68 bytes .

Name	Size (bytes)	Description
accX	2	Accelerometer x axis
accY	2	Accelerometer y axis
accZ	2	Accelerometer z axis
gyroX	2	gyroscope x axis
gyroY	2	Gyroscope y axis
gyroZ	2	gyroscope z axis
magX	2	Magnetometer x axis
magY	2	Magnetometer y axis
magZ	2	Magnetometer z axis
magHall	2	Magnetometer hall value
baroPress	4	pressure
baroTemp	4	temperature
GNSS_lat	4	latitude
GNSS_lon	4	longitude
GNSS_velN	4	Velocity north
GNSS_velE	4	Velocity east
GNSS_velD	4	Velocity down
GNSS_height	4	Height above ground
GNSS_hMSL	4	Height above sealevel
GNSS_gSpeed	4	groundspeed
GNSS_headMot	4	heading
CRC32_checksum	4	CRC32 checksum (RFC 1952)

4.3.3 Interfacing Jetson Nano with SensorBoard

A UART connection was used to interface the Jetson with the sensorboard, in order to send the data packet. The configuration is as follows:

Parameter	Value
Port	ttyTHS1

Baud rate	480600
WordLength	8 bits
Stopbits	2 bits
Parity	none

Data only needed to be transmitted in one direction (sensor to jetson). The rx pin 10 on the Jetson was connected to the tx pin PD8 of the STM32F4. See section 4.2.2 for details from the STM32F4 perspective.

Pin 40 of the Jetson was connected to PD10 of the STM32F4 in GPIO mode, which acted as a request line to synchronise the request for new sensor data with the real time control loop.

4.3.4 Interfacing Jetson with Motors

The Jetson Nano unfortunately does not have any PWM channels available, so in order to control the servo motors, a 6 channel hardware module called Pololu Micro Maestro was used a servo driver.

The Jetson communicated with this device using the serial protocol (via USB). The serial configuration was as follows:

Parameter	**Value**
Port	ttyACM0
Baud rate	9600
WordLength	8 bits
Stopbits	1 bit
Parity	none

The Pololu Maestro was configured using the software MaestroControlCenter provided by Pololu. The range of pulse length desired is between 1000 and 2000 (μs) to get the full range from all the motors. In order to be able to control the servos at the speed of the control loop, the PWM frequency needs to be at least 250Hz.

It should be noted that the position of a servo motor is determined only by the length of the pulse sent to it, irrespective of the frequency (provided the input signal meets the specifications of the motor).

The commands sent to the Pololu Maestro follow the ‘compact protocol’[7] which is 4 bytes in length. The first byte specifies the type of command; the

7 Pololu Maestro Servo Controller User’s Guide: pg 51

second byte specifies the channel (0 to 5); the last two bytes contain the data for the command.

4.4 Wireless Communication/Telemetry

In order for the Jetson Nano to communicate remotely with the user via a PC application, an XBEE PRO S2C was used. There is one device connected to the Jetson, and another connected to the laptop which runs the user application. This device communicates with its host using the standard serial interface.

The configuration of the serial comms is as follows:

Parameter	**Value**
Port	ttyUSB0
Baud rate	9600
WordLength	8 bits
Stopbits	1 bit
Parity	none

Configuration of this device was done using the Digi XCTU[8] application design for this purpose. The XBee connected to the Jetson was configured as a ‘coordinator’, and the XBee connected to the remote laptop/pc was configured as a ‘router’. ‘Transparent’ communication mode was chosen, which allows to the Xbee modules to only relay the information sent to it. The Xbee handles all the transmission and reception automatically, and behaves like a normal serial connection, from the host’s perspective, with an input buffer to read data from and output buffer to write data to.

The remote GUI app sends a message every 10Hz, and the Jetson polls or that message at 20Hz. The display data is sent from the Jetson to the remote GUI every 2Hz.

8 https://www.digi.com/products/embedded-systems/digi-xbee/digi-xbee-tools/xctu

4.5 Remote User Interface PC App

The User Interface application was designed to be a graphical way for the user to interact remotely with the robot, by sending commands and viewing onboard data. It was decided to build this application using Java (JDK 15), because it can be run on any user computer due to this cross platform nature. Netbeans 12.1 was the chosen IDE due to the GUI building functionality.

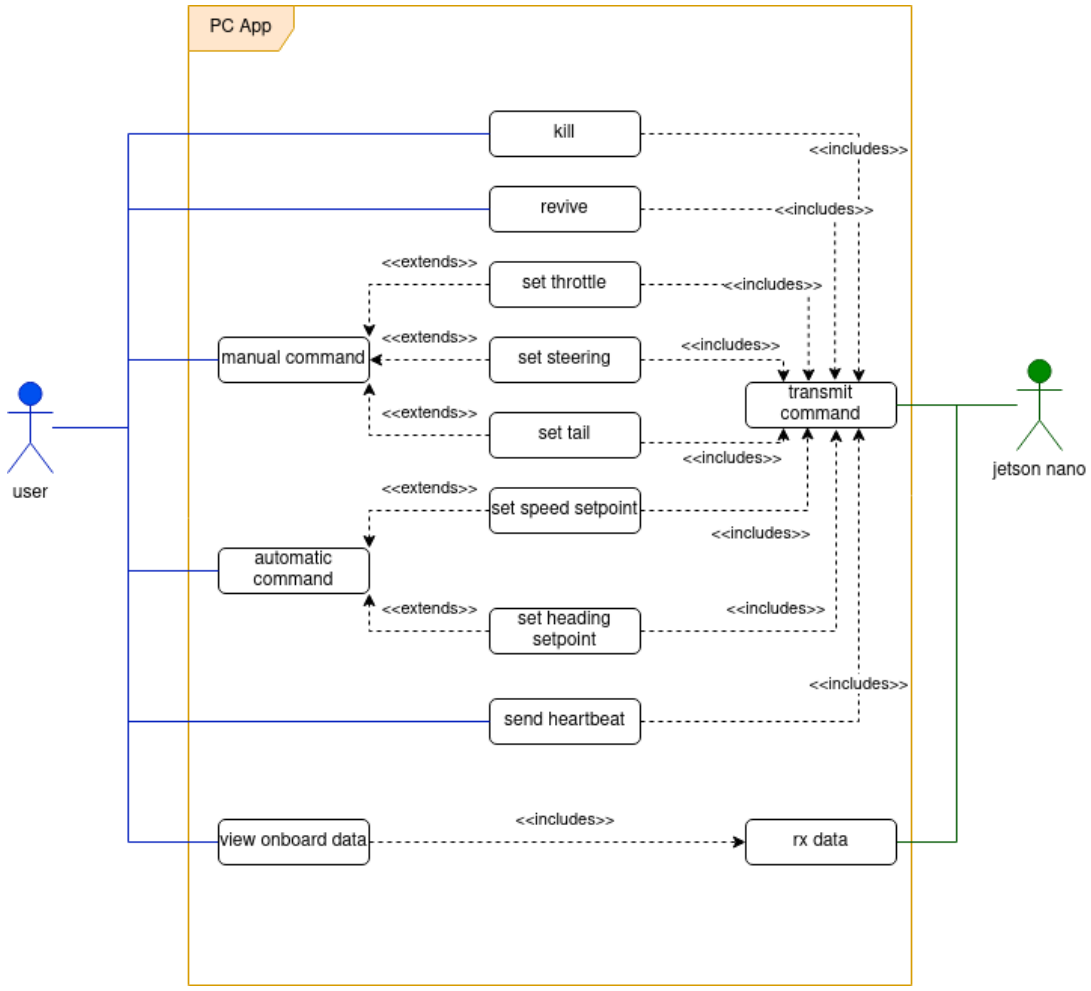


Figure 10: Use case diagram of the PC based user interface remote application

The use case diagram in Figure 10 outlines how the user can interact with the application, and the functionality that it provides. The user can send manual commands to directly control the position of the robot motors, or the user can send an automatic control command which will be used as a setpoint (speed and heading) for the controller to adjust the motion of the robot to. Certain selected onboard data is displayed and viewed from the GUI . The application will process the user's commands and send it wirelessly to the Jetson Nano. The onboard data will also be periodically received from the Jetson to update the GUI display.

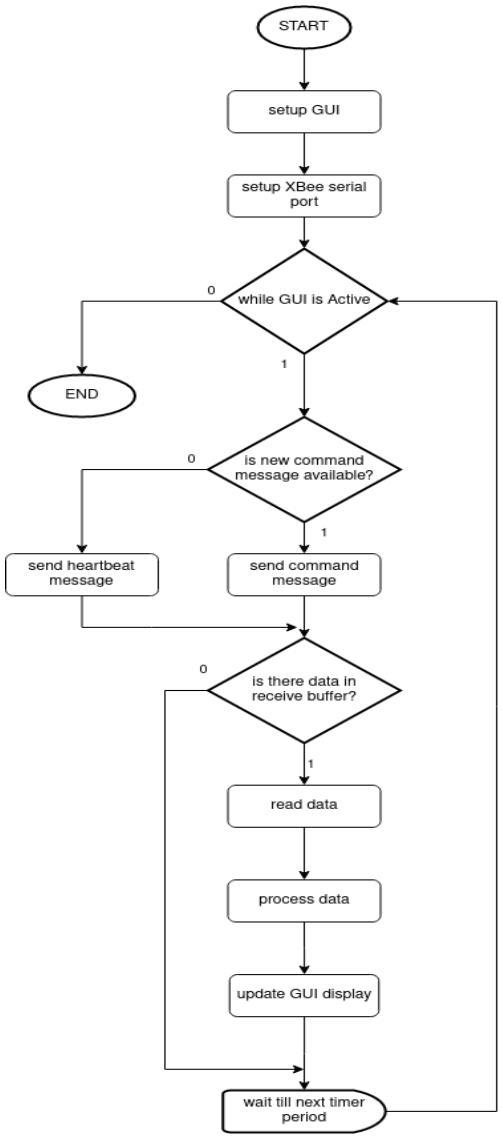


Figure 11: Flowchart of remote user interface application

The flowchart in Figure 11 outlines the program flow and logic of this subsystem. A heartbeat message is sent to the Jetson every 10Hz. If a command message is available, the that will be sent instead of the heartbeat. The format of the command messages and data is described in section 4.3.2.

In order to communicate with the XBee module, through the serial interface, an external (3rd party) library called JSSC 2.7.0 had to be imported. This is because Java does not offer libraries for serial communication with the USB ports. A combo box was also created to allow the user to choose which serial port they had connected the Xbee to.

The GUI that was created for this application, using the Netbeans GUI building tool, is illustrated in Figure 12.

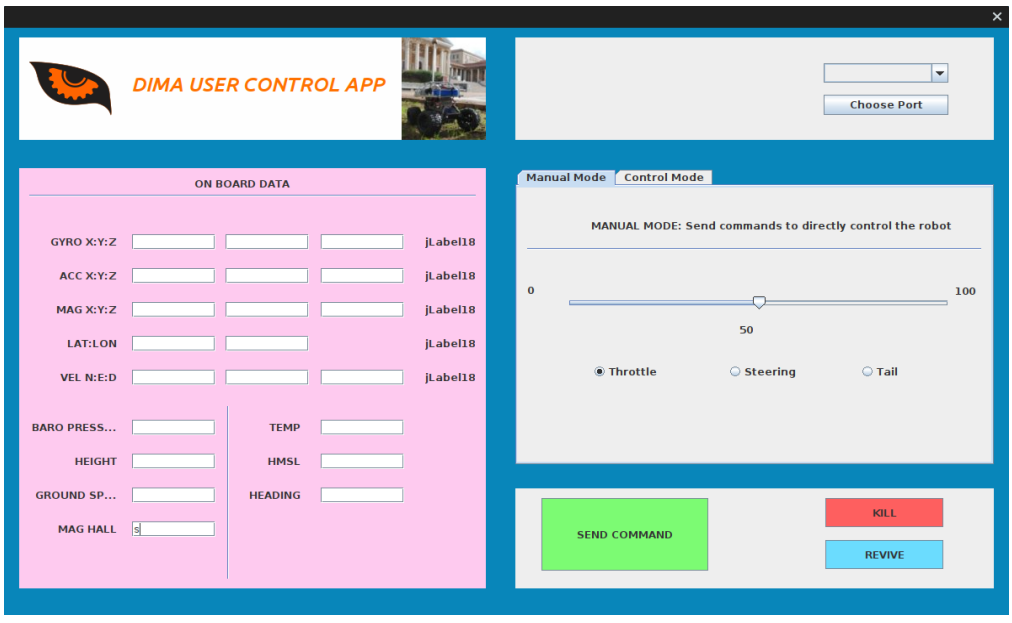


Figure 12: Graphical User Interface for the remote user Java application

5. Testing & Results

5.1 Testing Realtime Performance.

5.1.1 Testing Latency

In order to evaluate the latency of the system, an open source program called cyclictest[9] was run. A priority of 99 was chosen, with scheduler policy SCHED_FIFO used, and 1000000 iterations performed. The results are illustrated in Figure 13 .

```
nano@nano-desktop:~/cyclictest/rt-tests$ sudo ./cyclictest --smp -p99 -m -l1000000
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 0.01 0.02 0.00 1/639 18377

T: 0 (17112) P:99 I:1000 C:1000000 Min:      4 Act:    6 Avg:    7 Max:      157
T: 1 (17113) P:99 I:1500 C: 666665 Min:      4 Act:    8 Avg:    7 Max:      160
T: 2 (17114) P:99 I:2000 C: 499995 Min:      5 Act:    7 Avg:    7 Max:      147
T: 3 (17115) P:99 I:2500 C: 399993 Min:      4 Act:    7 Avg:    7 Max:      121
```

Figure 13: latency results from cyclictest, running with priority 99 on SCHED_FIFO policy

As is apparent, the system has an average latency of 7μs, and a maximum latency out of all the cores is 160μs. This test hence satisfies specification (1.1) .

5.1.2 Testing Jitter

A jitter test was performed by toggling a GPIO pin at 250Hz. The jitter is measured as the maximum time difference between the waveform edges. This was measured using a DSO3062A Agilent Digital Oscilliscope, with infinite persistence setting enabled. Figure 14 illustrates the results.

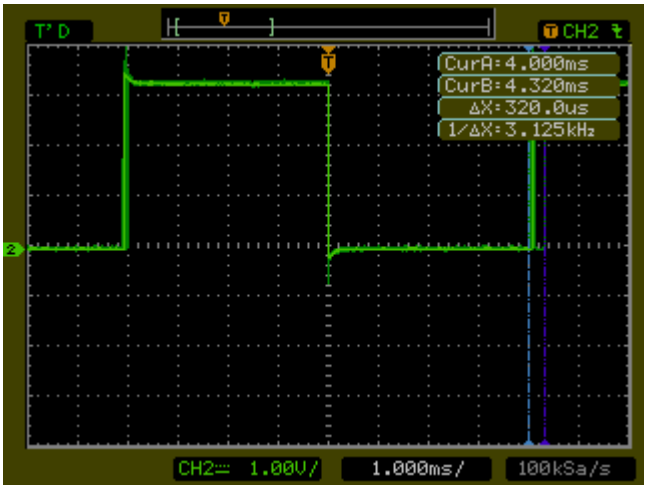


Figure 14: Oscilliscope capture of square pulse train generated by Jetson Nano

9 git://git.kernel.org/pub/scm/linux/kernel/git/clrkwllms/rt-tests.git

As measured between the two cursors, there is a maximum variation of 320μs. The jitter is therefore 8% of our desired realtime control frequency of 250Hz (4ms period). This test thus satisfies specification (1.2) .

5.2 Performance of Communication with Sensor Board

In order to test if the sensor data is being properly received, and at the desired speed, a GPIO pin was toggled from the Jetson to request a single packet of data. The checksum was then calculated and verified to determine the validity of the data. This was then done every 4ms (250Hz), and then number of dropped packets (invalid data) were counted.
The result of this test were as follows, where 'dropped' indicates the amount of invalid data packets, 'success' indicates the amount of valid packets, and 'time' indicates the total time that the test took.

```
dropped: 151
success: 209690
time: 839364 ms
```

The total requested data packets is 209690. Calculating the percentage of 'dropped' packets:
dropped_ratio = dropped/(success+dropped)
= 151/(209690+151) = 7.196e-4 = 0.0007196 = 0.072%

This is well within our desired ratio of 1% and hence specification (2.2) is satisfied. Since the data was successfully received at 250Hz, this also satisfies specification (2.1) .

5.3 Logging Data and Verifying GPS Data

Since the GPS data is automatically sent from the NEO-M8T module and saved in the STM32F4 memory before it is sampled, the data is not synchronised with the Jetson read requests. So in order to determine if the GPS Data is being updated every 10Hz, was was necessary to first log the data so that it can be manually reviewed after the program has terminated. A small C program was written to read the logged data from the logfile. It also used the gnuplot library[10] to plot a visual graph, and save it to PNG format.

By viewing only where the GPS data has changed, along with a timestamp from the Jetson, the update frequency of the GPS data can be determined. It was decided to view the latitude, longitude, and UTC data. A small subset of the analysed data sample is presented below: the timestamp is in milliseconds, as recorded by the Jetson; the UTC data is in the format (hh:mm:ss:nnnnnnnnn) ; the latitude and longitude are in degrees.

10 http://ndevilla.free.fr/gnuplot/gnuplot_i/index.html

```
timestamp||      UTC                      latitude:longitude
16      ||       17:30:55:300071777      -34.088501:18.475053
116     ||       17:30:55:400071759      -34.088502:18.475054
220     ||       17:30:55:500071741      -34.088502:18.475054
316     ||       17:30:55:600071722      -34.088502:18.475054
416     ||       17:30:55:700071704      -34.088502:18.475054
516     ||       17:30:55:800071686      -34.088502:18.475054
616     ||       17:30:55:900071667      -34.088502:18.475054
716     ||       17:30:56:    71649      -34.088502:18.475054
816     ||       17:30:56:100071631      -34.088502:18.475054
916     ||       17:30:56:200071612      -34.088502:18.475054
1016    ||       17:30:56:300071594      -34.088502:18.475055
1112    ||       17:30:56:400071576      -34.088502:18.475055
1216    ||       17:30:56:500071558      -34.088502:18.475055
1316    ||       17:30:56:600071539      -34.088502:18.475055
1416    ||       17:30:56:700071521      -34.088502:18.475055
1516    ||       17:30:56:800071503      -34.088502:18.475055
1616    ||       17:30:56:900071485      -34.088502:18.475055
1712    ||       17:30:57:    71466      -34.088502:18.475055
```

By measuring the difference between the timestamps, we get:
update_period = (100 ± 4) ms. This means that the frequency of GPS data acquisition is between 10.4 and 9.6 Hz.

Considering that the GPS data will not be used in the real time control algorithm, this result is acceptable. Thus specification (3.1) is met.

This experiment also tested that the data was being logged to the file system, so specification (4.1) was also satisfied.

5.4 Testing Wireless/Remote Operation

-spec 5.1 and 5.2
In order to evaluate the reliability of the wireless communication between the Jetson and the remote GUI application, the amount of consecutive dropped packets were measured. A variable was used to count the number of dropped packets from the Jetson side, while moving the remote laptop further away until a line of sight distance of 50m. At this distance there was a maximum of 14 consecutive lost packets. Since the period between each poll is 50ms, this corresponds to a 0.7s period of no communication. This thus satisfies specification (5.2) and (5.1).

5.5 Testing Motor Operation

In order to determine if whether the desired range of servo operation is achievable, the output of the Pololu Maestro (servo driver) was viewed on an oscilloscope (DSO3062A Agilent Digital Oscilliscope).

The available range of pulse lengths were tested with a PWM frequency of 250Hz.

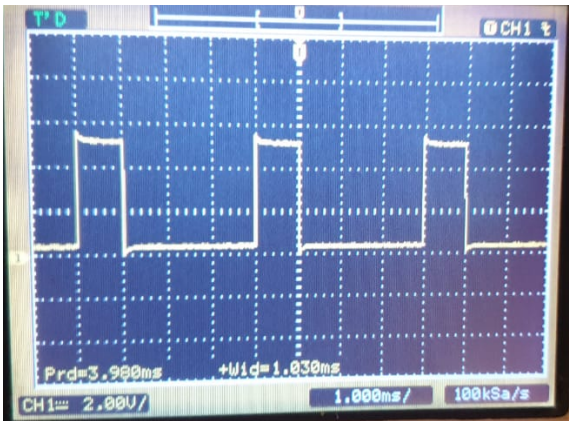


Figure 16: servo driver minimum pulse length for frequency of 250Hz

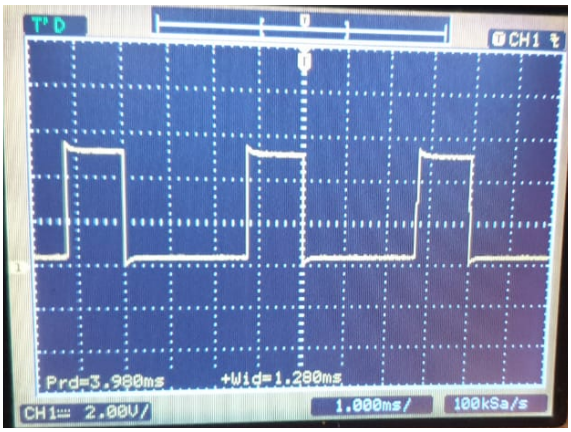


Figure 15: servo driver maximum pulse length for frequency of 250Hz

The minimum pulse length of 1 ms does satisfy the lower bound of the desired range, however, the maximum pulse length that could be produced at this frequency was 1.28 ms which does not meet the required length of 2 ms for the upper bound. This is unfortunately a physical limitation of the Pololu Micro Maestro operating at this frequency.

The test was repeated for a frequency of 50Hz, and the minimum and maximum pulse lengths were measured.

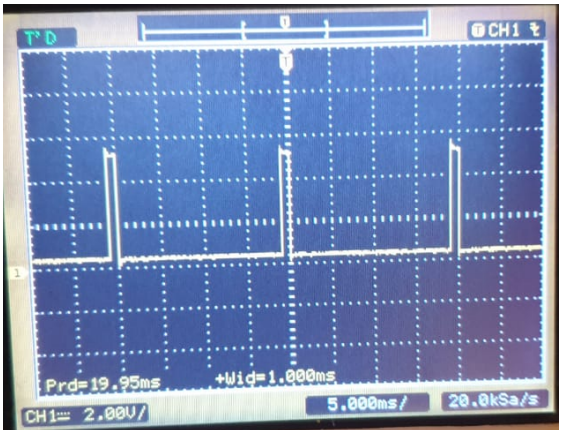


Figure 17: servo driver minimum pulse length for frequency of 50Hz

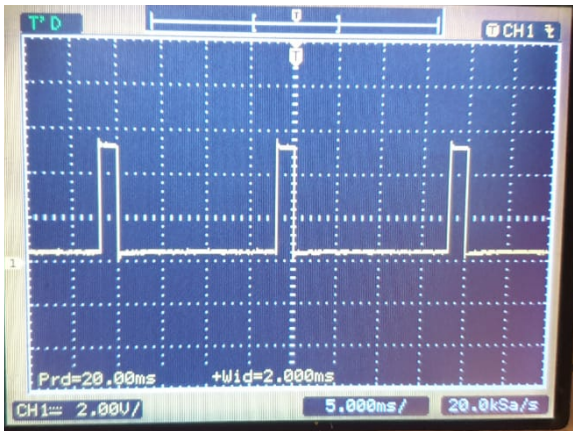


Figure 18: servo driver maximum pulse length for frequency of 50Hz

For a PWM frequency of 50Hz, the minimum pulse length satisfies the lower bound of 1ms, and the maximum pulse length satisfies the upper bound of 2ms. Thus the full range of pulse lengths is achieved, satisfying specification (6.1). The specification(6.2) however, was not met, because the physical limitation of the hardware used meant that both the pulse length and frequency specifications could not be concurrently achieved.

5.6 Demo & Code

A short video demo of the robot being controlled from the remote GUI can be can be found here:
https://youtu.be/2_5sEA1Z2CY

Source code for the sensor board firmware:
https://github.com/vikylenaidoo/EEE4022S_FinalYearProject_SensorFirmware/tree/master/DIMA_SensorBoard_Firmware

Source code for Jetson software:
https://github.com/vikylenaidoo/EEE4022S_FinalYearProject_JetsonControl

Source code for remote GUI application:
https://github.com/vikylenaidoo/EEE4022S_FinalYearProject_SensorFirmware/tree/master/DIMA_UI

6. Discussion of Results

On of the most important aspirations was to achieve realtime performance of the control loop. This meant that the control action must be allowed to trigger within a guaranteed and deterministic period of time. The tests pertaining to realtime performance confirmed the ability of the system to act in a deterministic and timeous manner. The latency of 160μs and the jitter of 320μs hardly impacts the desired 4ms period, and the testing proved that no cycles of the realtime loop will be skipped. The approach that was taken to implement the realtime capabilities, PREEMPT_RT patch, performed satisfactorily for this application, considering we only required a realtime frequency of 250Hz, which is not extremely fast. Based on the literature review, implementing the realtime capabilities using the Xenomai dual kernel approach, would in fact offer better performance, but would be overkill for this application.

The performance of data acquisition process was evaluated by the speed and reliability of the communication with the sensor board (which provided the sensor data). The rate of data reception was sufficient, but due to the inherently unreliable nature of the UART connection, especially at faster baud rates, there was a 0.072% loss of data packets. This may not seem significant at first glance, but considering the hard real time requirements of the system, it meant that the control loop would not be able to calculate the next action and consequentially not perform deterministically. This was anticipated, when writing the system specifications, and to reduce the impact of this problem, the controller will perform the action of the previous loop, thus securing the guaranteed periodic update of motors. This would not negatively affect the performance if a proper feedback controller is designed.

The acquisition of GPS data was not exactly at 10Hz, but there was a 4ms jitter. This was actually due to a synchronisation clash between the Jetson requesting data, and the NEO-M8T module updating its data. Because the NEO-M8T automatically sent an updated data message every 10Hz, to the STM32F4, this data stored in the STM32F4 memory, and was not updated at the request of the Jetson. So when a data request was made to the sensorboard while the GPS data was also being updated, the old GPS data was sent. This is why the jitter of GPS data was 4ms, corresponding to one realtime period.

Remote operation was possible at a distance of 50m, with a maximum period of 0.7s without communication. It should be noted that the test was performed in an urban area, where various sources of interference could have influenced this result. The system was designed to automatically kill the motors if a period of more than 1s occurred without communication. It can be revived

from the GUI once connection is re-established. Although it is possible to operate from a distance further than 50m, reliable communication can not be guaranteed, since that is beyond the specifications of this system.

Although the motor driver (Pololu Maestro) was able to able to achieve the full pulse width range necessary to control the servo motors, it was not possible to achieve a fast enough refresh rate (PWM frequency). In order to update the position of the motors fast enough, the PWM output of the servo driver would need to have a frequency of at least 250hz (the control frequency). Unfortunately, due to the hardware limitation of the servo driver, a trade off needed to be made between the range of possible pulse widths, and the frequency. At a 250Hz frequency, the servo driver was not able to produce the full range of pulse widths, so it was decided to limit the frequency to 50Hz in preference for achieving full servo motor range. Another limitation encountered was that the electronic speed controller (ESC) which interfaces the throttle motor with the PWM signal from the servo driver, was limited to a 50Hz servo signal (which is the standard RC/hobby frequency). This was also a factor that motivated the decision to use a 50Hz refresh rate for the servo driver.

This effectively meant that the real time control loop frequency also had to be limited to 50Hz, because the servo driver could not receive commands faster than its refresh rate. The modification did not negatively affect the realtime performance. A global variable was used to hold the realtime period, so that it can be easily modified in the future if a better servo driver is used.

7. Conclusions & Recommendations

7.1 Conclusion

A framework was developed for implementing a realtime control system for the DIMA robot. A literature review was conducted to investigate some key questions posed earlier:

- *what are existing/commonly used control frameworks for robotic control systems?*
- *what is meant by real time requirements in robotic control?*
- *what is the best method of implementing a realtime OS?*

A few previous control frameworks were explored to gain an insight into common methods of implementing such a system. The nature and requirements for realtime control were investigated, and the most importamt conclusion drawn was that realtime performance was a factor of the predictability of a system and its reliability to accurately perform periodic tasks within a predetermined period, rather than the absolute speed of the control loop. Jitter and latency were the two metrics needed to evaluate the performance of a realtime system. It was decided that the PREEMPT_RT patch was the most appropriate method of achieving realtime performance for this system.

This project entailed the modular design of three subsystems. A firmware was developed for a pre-existing sensorboard, based on a STM32F4 microcontroller, with various sensors, to better suit it to this application. The realtime control capabilities were implemented on a NVIDIA Jetson Nano, and realtime performance was made possible with the PREEMPT_RT patch for the Linux kernel. A remote GUI application was developed for the user to be able wirelessly control the robot, and view onboard data from their pc/laptop. Finally this system was interfaced with the existing hardware of the DIMA robot.

The system specification was tested by conducting experiments and benchmarking according to the acceptance test procedure outlined in section 3.3. Most of the specifications were met, except for specification (6.2) which required the PWM frequency of the motor driver to be 250Hz. This posed a limitation on the entire system and, as a result, the control loop frequency had to be reduced to 50Hz.

7.2 Recommendations

Based on the results obtained from this project, with the aspiration to use this framework to control the DIMA robot in the future, the following recommendations are proposed:

7.2.1 Feedback Controller

The development of this system did not encompass the design of a feedback controller, but rather provided a framework with real time capabilities with which to implement a controller for the motion of DIMA. A feedback controller should be designed, that will take the input provided by the sensors, and the setpoint provided by the user, to update the motor commands in order to control the motion of the robot.

7.2.2 Better Servo Driver

The main contributor to the failure of specification (6.2), (which was the requirement of 250Hz control frequency), was the limitation of the Pololu Micro Maestro in providing a servo refresh rate/PWM frequency of 250Hz. In order to realise the desired control frequency of 250Hz, a better servo driver should be installed, which will allow the motors to be updated faster.

7.2.3 Xenomai Dual Kernel

If in the future it is desired to create a faster controller (perhaps in the kHz range), it is recommended to modify the existing implementation PREEMPT_RT patch, with a Xenomai Dual kernel approach of achieving real time capabilities. The reason for this is that the Xenomai performance is undoubtedly better than PREEMPT_RT, as discussed in the literature review, and would be better suited to a faster application. This however could mean the code for the Jetson may have to be rewritten, since Xenomai uses its own APIs. Fortunately the Xenomai API also offers a POSIX skin[11], which means the current framework could be upgraded with minimal modification.

11 https://xenomai.org/documentation/xenomai-2.6/html/api/group__posix.html

8. List of References

[1] D. Y. Ku, ‘Kinematic State Estimation using Multiple DGPS/MEMS-IMU Sensors’, University of Cape Town, 2020.

[2] J. A. Grimes and J. W. Hurst, ‘The design of atrias 1.0 a unique monopod, hopping robot’, presented at the International Conference on Climbing and Walking Robots, 2012.

[3] A. Peekema and D. R. J. Hurst, ‘OPEN-SOURCE REAL-TIME ROBOT OPERATION AND CONTROL SYSTEM FOR HIGHLY DYNAMIC, MODULAR MACHINES’, presented at the ASME 2013 International Design Engineering Technical Conferences & International Conference on Multibody Systems, Nonlinear Dynamics, and Control, Portland, Oregonm USA, Aug. 2013.

[4] R. Delgado, B.-J. You, and B. W. Choi, ‘Real-time control architecture based on Xenomai using ROS packages for a service robot’, *The Journal of Systems and Software*, vol. 151, pp. 8–19, Sep. 2018, doi: 10.1016.

[5] H. Lee, Y.-H. Kim, K. Lee, D.-K. Yoon, and B.-J. You, ‘Designing the Appearance of a Telepresence Robot, M4K: A Case Study’, in *Cultural Robotics*, vol. 9549, J. T. K. V. Koh, B. J. Dunstan, D. Silvera-Tawil, and M. Velonaki, Eds. Cham: Springer International Publishing, 2016, pp. 33–43.

[6] W. E. Hong, J. S. Lee, L. Rai, and S. J. Kang, ‘RT-Linux based Hard Real-Time Software Architecture for Unmanned Autonomous Helicopters’, presented at the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Kyungpook National University, 2005.

[7] T. Knutsson, ‘Performance Evaluation of GNU/Linux for Real-Time Applications’, Uppsala Universitet, 2008.

[8] Linux Foundation, ‘Priority inversion - priority inheritance’, Jun. 23, 2016. https://wiki.linuxfoundation.org/realtime/documentation/technical_basics/pi (accessed Sep. 20, 2020).

[9] J. H. Brown and B. Martin, ‘How fast is fast enough? Choosing between Xenomai and Linux forreal-time applications’, Rep Invariant Systems, Inc., Cambridge, MA, USA.

[10] P. Gerum, ‘Xenomai Wiki’, 2019. https://gitlab.denx.de/Xenomai/xenomai/-/wikis/Start_Here (accessed Sep. 20, 2020).

[11] P. Deutsch, ‘GZIP file format specification version 4.3’. Aladdin Enterprises, May 1996, Accessed: Oct. 10, 2020. [Online]. Available: https://tools.ietf.org/html/rfc1952#section-8.

[12] ajcalderont, ‘PREEMPT-RT patches for Jetson Nano’, *NVIDIA Developer*, Dec. 22, 2019. https://forums.developer.nvidia.com/t/preempt-rt-patches-for-jetson-nano/72941/25 (accessed Oct. 09, 2020).

9. Appendices

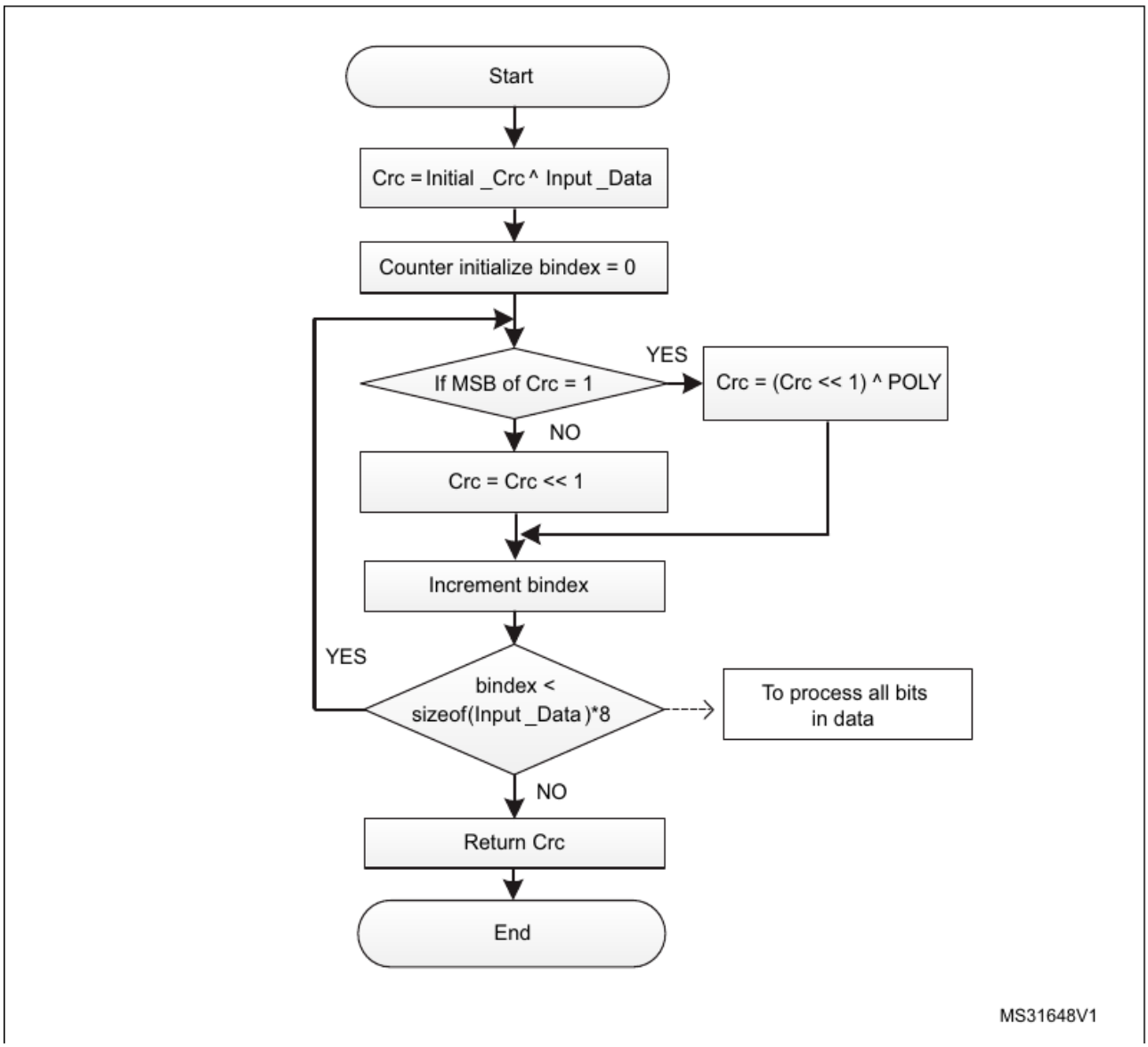
9.1 APPENDIX A: CODE AND ALGORITHMS

9.1.1 UBX checksum algorithm

```
static bool gnss_ubx_checksum(){
    uint8_t CK_A = 0;
    uint8_t CK_B = 0;

    uint8_t ck_a = GNSS_RX_BUFFER[98];
    uint8_t ck_b = GNSS_RX_BUFFER[99];

    for(int i=2; i<GNSS_BUFFER_SIZE-2; i++){
        CK_A = CK_A + GNSS_RX_BUFFER[i];
        CK_B = CK_B + CK_A;
    }

    if(CK_A != ck_a)
        return false;

    if(CK_B != ck_b)
        return false;

    return true;

}
```

9.1.2 CRC32 Algorithm (From STM32F4) [12]

Uses CRC-32 (Ethernet) polynomial: 0x4C11DB7



12 See stm32f4_crc32_app_note_an4187

9.1.3 CRC32 Algorithm (RFC 1952) [13]

Reference: [11]

```
The following sample code represents a practical implementation of
   the CRC (Cyclic Redundancy Check). (See also ISO 3309 and ITU-T V.42
   for a formal specification.)

      /* Table of CRCs of all 8-bit messages. */
      unsigned long crc_table[256];

      /* Flag: has the table been computed? Initially false. */
      int crc_table_computed = 0;

      /* Make the table for a fast CRC. */
      void make_crc_table(void)
      {
        unsigned long c;
        int n, k;

        for (n = 0; n < 256; n++) {
          c = (unsigned long) n;
          for (k = 0; k < 8; k++) {
            if (c & 1) {
              c = 0xedb88320L ^ (c >> 1);
            } else {
              c = c >> 1;
            }
          }
          crc_table[n] = c;
        }
        crc_table_computed = 1;
      }
```

13 Verbatim from: https://tools.ietf.org/html/rfc1952#section-8

```
/*
         Update a running crc with the bytes buf[0..len-1] and return
       the updated crc. The crc should be initialized to zero. Pre- and
       post-conditioning (one's complement) is performed within this
       function so it shouldn't be done by the caller
      */
      unsigned long update_crc(unsigned long crc,
                      unsigned char *buf, int len)
      {
        unsigned long c = crc ^ 0xffffffffL;
        int n;

        if (!crc_table_computed)
          make_crc_table();
        for (n = 0; n < len; n++) {
          c = crc_table[(c ^ buf[n]) & 0xff] ^ (c >> 8);
        }
        return c ^ 0xffffffffL;
      }

      /* Return the CRC of the bytes buf[0..len-1]. */
      unsigned long crc(unsigned char *buf, int len)
      {
        return update_crc(0L, buf, len);
      }
```

9.2 APPENDIX B: INFO FROM DATASHEET

9.2.1 UBX-NAV-PVT Message structure

Message	**UBX-NAV-PVT**					
Description	**Navigation position velocity time solution**					
Firmware	Supported on: u-blox 8 / u-blox M8 protocol versions 15, 15.01, 16, 17, 18, 19, 19.1, 19.2, 20, 20.01, 20.1, 20.2, 20.3, 22, 22.01, 23 and 23.01					
Type	Periodic/Polled					
Comment	This message combines position, velocity and time solution, including accuracy figures. Note that during a leap second there may be more or less than 60 seconds in a minute. See the description of leap seconds for details.					
	Header	Class	ID	Length (Bytes)	Payload	Checksum
Message Structure	0xB5 0x62	0x01	0x07	92	see below	CK_A CK_B

Payload Contents:

Byte Offset	Number Format	Scaling	Name	Unit	Description
0	U4	-	iTOW	ms	GPS time of week of the navigation epoch. See the description of iTOW for details.
4	U2	-	year	y	Year (UTC)
6	U1	-	month	month	Month, range 1..12 (UTC)
7	U1	-	day	d	Day of month, range 1..31 (UTC)
8	U1	-	hour	h	Hour of day, range 0..23 (UTC)
9	U1	-	min	min	Minute of hour, range 0..59 (UTC)
10	U1	-	sec	s	Seconds of minute, range 0..60 (UTC)
11	X1	-	valid	-	Validity flags (see graphic below)
12	U4	-	tAcc	ns	Time accuracy estimate (UTC)
16	I4	-	nano	ns	Fraction of second, range -1e9 .. 1e9 (UTC)
20	U1	-	fixType	-	GNSSfix Type: 0: no fix 1: dead reckoning only 2: 2D-fix 3: 3D-fix 4: GNSS + dead reckoning combined 5: time only fix
21	X1	-	flags	-	Fix status flags (see graphic below)
22	X1	-	flags2	-	Additional flags (see graphic below)
23	U1	-	numSV	-	Number of satellites used in Nav Solution
24	I4	1e-7	lon	deg	Longitude
28	I4	1e-7	lat	deg	Latitude
32	I4	-	height	mm	Height above ellipsoid
36	I4	-	hMSL	mm	Height above mean sea level
40	U4	-	hAcc	mm	Horizontal accuracy estimate
44	U4	-	vAcc	mm	Vertical accuracy estimate
48	I4	-	velN	mm/s	NED north velocity
52	I4	-	velE	mm/s	NED east velocity
56	I4	-	velD	mm/s	NED down velocity
60	I4	-	gSpeed	mm/s	Ground Speed (2-D)
64	I4	1e-5	headMot	deg	Heading of motion (2-D)
68	U4	-	sAcc	mm/s	Speed accuracy estimate
72	U4	1e-5	headAcc	deg	Heading accuracy estimate (both motion and vehicle)
76	U2	0.01	pDOP	-	Position DOP
78	X1	-	flags3	-	Additional flags (see graphic below)
79	U1[5]	-	reserved1	-	Reserved
84	I4	1e-5	headVeh	deg	Heading of vehicle (2-D), this is only valid when headVehValid is set, otherwise the output is set to the heading of motion
88	I2	1e-2	magDec	deg	Magnetic declination. Only supported in ADR 4.10 and later.
90	U2	1e-2	magAcc	deg	Magnetic declination accuracy. Only supported in ADR 4.10 and later.

9.3 Appendix C: Miscellaneous

9.3.1 Procedure for building real time patched kernel

This was done by following the the instructions provided by ajcalderont in the NVIDIA forum [12] .

Required packages needed to be installed first:

```
sudo apt-get update
sudo apt-get install libncurses5-dev
sudo apt-get install build-essential bc
sudo apt-get install lbzip2
sudo apt-get install qemu-user-static
```

Build folder was created:

```
mkdir $HOME/jetson_nano
cd $HOME/jetson_nano
```

Files were extracted to the folder

```
sudo tar xpf Tegra210_Linux_R32.3.1_aarch64.tbz2
cd Linux_for_Tegra/rootfs/
sudo tar xpf ../../Tegra_Linux_Sample-Root-Filesystem_R32.3.1_aarch64.tbz2
cd ../../
tar -xvf gcc-linaro-7.3.1-2018.05-x86_64_aarch64-linux-gnu.tar.xz
sudo tar -xjf public_sources.tbz2
tar -xjf Linux_for_Tegra/source/public/kernel_src.tbz2
```

PREEMPT_RT patches were installed:

```
cd kernel/kernel-4.9/
./scripts/rt-patch.sh apply-patches
```

The kernel was compiled as follows:

```
TEGRA_KERNEL_OUT=jetson_nano_kernel
mkdir $TEGRA_KERNEL_OUT
export CROSS_COMPILE=$HOME/jetson_nano/gcc-linaro-7.3.1-2018.05-x86_64_aarch64-linux-gnu/bin/
aarch64-linux-gnu-
make ARCH=arm64 O=$TEGRA_KERNEL_OUT tegra_defconfig
make ARCH=arm64 O=$TEGRA_KERNEL_OUT menuconfig
```

The following options were chosen under kernel features:

Preemption Model: Fully Preemptible Kernel (RT)
Timer frequency: 1000 HZ

```
make ARCH=arm64 O=$TEGRA_KERNEL_OUT -j4

sudo cp jetson_nano_kernel/arch/arm64/boot/Image $HOME/jetson_nano/Linux_for_Tegra/kernel/Image
sudo cp -r jetson_nano_kernel/arch/arm64/boot/dts/* $HOME/jetson_nano/Linux_for_Tegra/kernel/dtb/
sudo make ARCH=arm64 O=$TEGRA_KERNEL_OUT modules_install
INSTALL_MOD_PATH=$HOME/jetson_nano/Linux_for_Tegra/rootfs/
```

```
cd $HOME/jetson_nano/Linux_for_Tegra/rootfs/
sudo tar --owner root --group root -cjf kernel_supplements.tbz2 lib/modules
sudo mv kernel_supplements.tbz2 ../kernel/

cd ..
sudo ./apply_binaries.sh
```

Finally the Jetson Nano Image could be created

```
cd tools
sudo ./jetson-disk-image-creator.sh -o jetson_nano.img -s 14G -b jetson-nano -r 200
```

The image was then flashed to the Jetson’s micro SD card.