The Design and Implementation of Supports for a Customized Quadrotor Platform

The University of Sheffield

BEng Electronics Engineering

3rd year design project

Name: Tai Li

Supervisor: Dr Luke Seed

Second marker: Dr Kristian Groom

Word count: 6374

1. Abstract

A quadcopter is a small four-armed robot, which has a motor and a propeller on each arm to support flying stably. This project is a continuous project. The main objective of this continuous project was to produce a Raspberry Pi shield for students to easily implement a design of a quadcopter on a Raspberry Pi and its shied.

In this project, the main objective was to provide a user-friendly library on a Raspberry Pi shield created in the previous project. The library is for students to speed up the implementation stage of their own quadcopter by modularizing each block within a software design for a quadcopter. This was done by replicating the functionality of an existing quadcopter project from a Raspberry Pi platform to a bare-metal platform (TI MCUs). It included porting generic C++ computational module crossing platforms, replication of the functionality of low-level peripheral drivers, as well as a system level development. In the meantime, not only this support provided for the Raspberry Pi shield but also a reference system was built up which helped support students to use this shield.

Contents

2. Terms and Abbreviations

This report uses the following terms and abbreviations:

Terms	**Meaning**
AHRS	Attitude and heading reference system
API	Application program interface
C++	A general-purpose programming language
GCC	GNU Compiler Collection
I/O	Input and output
IDE	Integrated development environment
JTAG	Joint Test Action Group. It specified Standard Test Access Port and Boundary-Scan Architecture
LAN	Local area network
LHS	Left-hand-side
Motor Controllers	Two of three TI MCUs on the OptimusPi board, mainly controlling motors and communicating with Overseer
MPU9150	An on-board 9-asix sensor on OptimusPi board
OptimusPi	A project of designing a Raspberry Pi shield for students to easily implement a design of a quadrotor
OptimusPi board	Customized motor control board with three TI MCUs and sensor MPU9150 on it
OptimusPi_r0	Quadcopter project Revision 0 carried out by Matthew Watson, using Raspberry Pi and OptimusPi board to implement design
OptimusPi_r1	Quadcopter project Revision 1 carried out by Tai Li, mainly using OptimusPi board only to implement design
Overseer	One of three TI MCUs on the OptimusPi board, mainly handling GPIOs, MPU9150 sensor, UART, SPI, I2C, and other peripherals
PID control	Proportional-integral-derivative control
PWM	Pulse width modulation
Raspberry Pi	A credit card-sized single-board computer, running on a Linux operating system
RF	Radio Frequency
RF-controlled quadcopter	A standalone quadcopter system, controlled by a RF remote controller
RHS	Right-hand-side
RPi	Raspberry Pi
RPi-controlled quadcopter	A quadcopter controlled by a Raspberry Pi, which is either controlled remotely via Wifi/Bluetooth, or auto-pilot
SMBus	System Management Bus

SoC	System-on-Chip
SPI	Serial Peripheral Interface
SSH	Secure shell
Standalone quadcopter	See RF-controlled quadcopter
TI MCU	Texas Instruments™ microcontroller
TivaWare	An extensive suite of software tools designed to simplify and speed development of Tiva C Series-based MCU applications
toolchain	A set of programming tools to help produce a program, normally including a compiler, a linker, libraries, and a debugger
UART	Universal asynchronous receiver/transmitter
Ubuntu	A PC Linux operating system
VNC	Virtual Network Computing

3. Introduction

3.1. Background

A quadrotor is a small four-armed robot, which has a motor and a propeller in each arm to support flying stably. The amount of quadrotors manufactured has increased dramatically recently. They have been used in both military as well as commercial applications. [1]

3.2. Context

This project continued from projects carried out by two previous students Matthew Watson and Alexander Stevens. The main objective of this continuous project was to produce a Raspberry Pi shield for students to easily implement a design of a quadrotor on a Raspberry Pi and its shied. [1] In order to achieve this objective, the following tasks has to be completed:

1) A design of the Raspberry Pi shield powerful enough to control at least four motors for a quadrotor project as well as other peripherals. The hardware design of this shield has been completed by Matthew Watson. This motor control board is also known as OptimusPi board. [2]
2) Providing an easy way to communicate between Raspberry Pi and OptimusPi board. This included the communication during run-time and before the program starts, i.e. providing a method of loading motor control board program from Raspberry Pi. This program could be cross-compiled somewhere else but loaded via Raspberry Pi. A SPI communication system in program run-time between a Raspberry Pi and three TI MCUs in OptimusPi board has been completed by Matthew Watson. [2] And a method of loading program from Raspberry Pi to OptimusPi board has been completed by Alexander Stevens. [1]
3) Providing a user-friendly library on OptimusPi board for students to use, modify and download to OptimusPi board in order to implement a quadrotor project easily, mainly easy to control motors and to communicate with peripherals. [1] This was done by building a quadcopter project on a Raspberry Pi and this OptimusPi board. However, when Matthew Watson carried out this quadcopter project, he wrote most of computational-level code in the Raspberry Pi as well as a sensor driver class for a RF-controlled quadcopter project [2], as known as OptimusPi_r0, which was not ideal for fully supporting this OptimusPi board.
4) Providing an user-friendly library on Raspberry Pi as well for students to use, modify in order to implement a quadcopter project easily, mainly high-level

supports such as Bluetooth, Wifi, GPS, GUI and auto-pilot algorithm. This has not been started yet.

4. Project Specifications

4.1. Current project objectives

In order to provide more supports on OptimusPi board, Matthew Watson built a RF-controlled quadcopter on this board with a Raspberry Pi [2], as known as OptimusPi_r0. But it was not ideal to write any code in a Raspberry Pi for a RF-controlled quadcopter project because this piece of software could have been written in OptimusPi board to provide more user cases for OptimusPi board. Therefore, this project was to port most of code from Raspberry Pi down to OptimusPi board to maintain the same functionality, and meanwhile, to provide more use cases for OptimusPi board and organize them into a library for OptimusPi board.

4.2. OptimusPi board

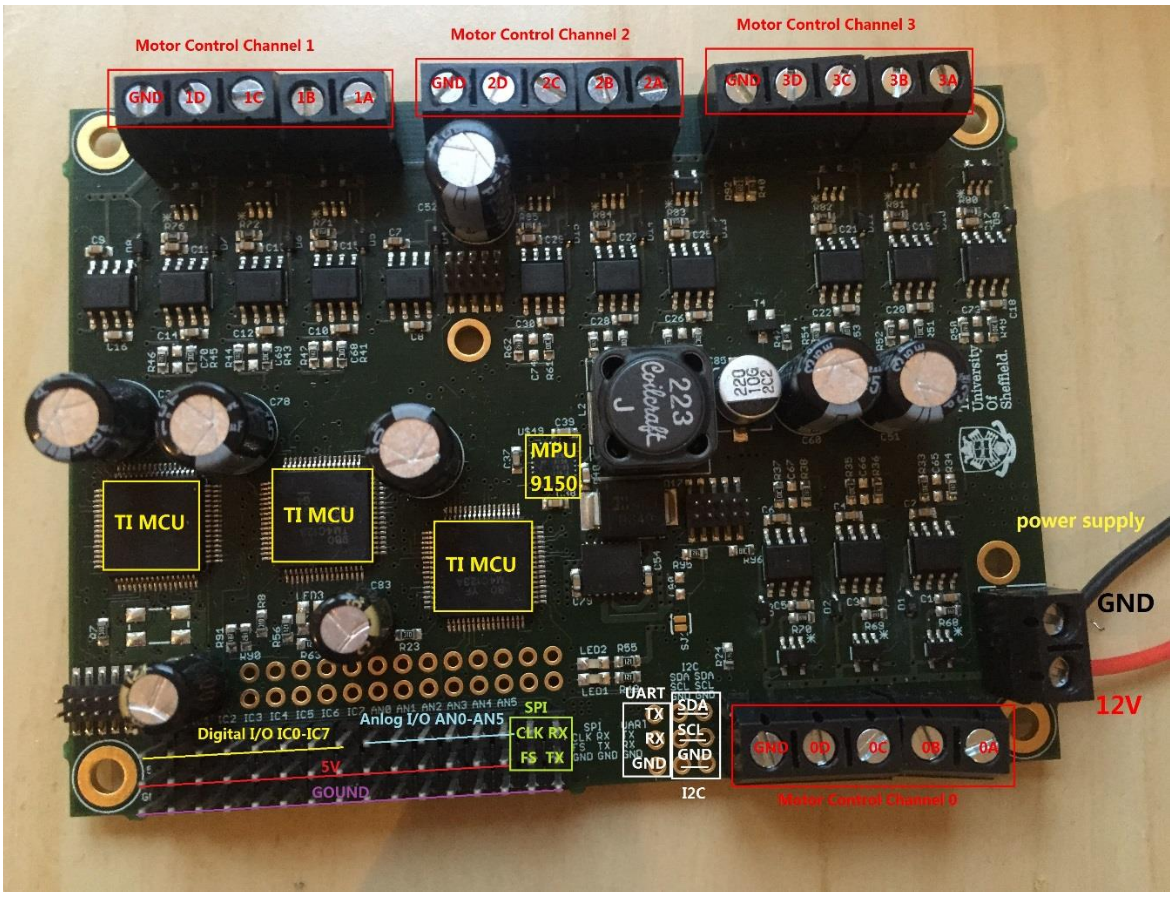


Figure 1 OptimusPi_board I/O support

OptimusPi board was designed by Matthew Watson to provide a Raspberry Pi shield powerful enough to drive at least four motors, to collect real time sensor information, to communicate with a Raspberry Pi and to support other peripherals.

There are three TI MCUs on board. The TI MCU is a System-on-Chip (SoC) called TM4C123AH6PM based on ARM-M4 core, designed to be capable of motion control. [3] Two of TI MCUs are denoted as Motor Controllers, physically connecting to half bridges of motor channels and internal SPI bus with another TI MCU. The other TI MCU is denoted as Overseer, physically connecting to lots of on-board peripherals, such as I/Os, I2C, UART, external SPI bus and internal SPI bus with Motor Controllers.

There are four motor channels present on board where each channel supports up to four pulse width modulation (PWM) channels. In other words, it supports at least four motors and up to sixteen motors. Every two motor channels are controlled by a Motor Controller, which receives and sends PWM signals to control half bridges on board in order to control motors.

MPU9150 is an on-board 9-asix sensor including a gyroscope, an accelerometer and a magnetometer. TI MCUs collect real-time data from MPU9150 via an on-board I2C bus.

OptimusPi board supports eight digital I/Os and six analog I/Os. Currently, each digital I/O can be configured as not only an input/output but also an input capture pin to capture how much duty cycle of PWM signals received on the input pin.

The board needs to be powered by a 12V power supply and it supports 5V power on the board to provide power for Raspberry Pi and other peripherals. (Noted, the TI MCUs are powered on 3.3V but no user has access to the 3.3V power rail)

4.3. OpitmusPi_r0 specification

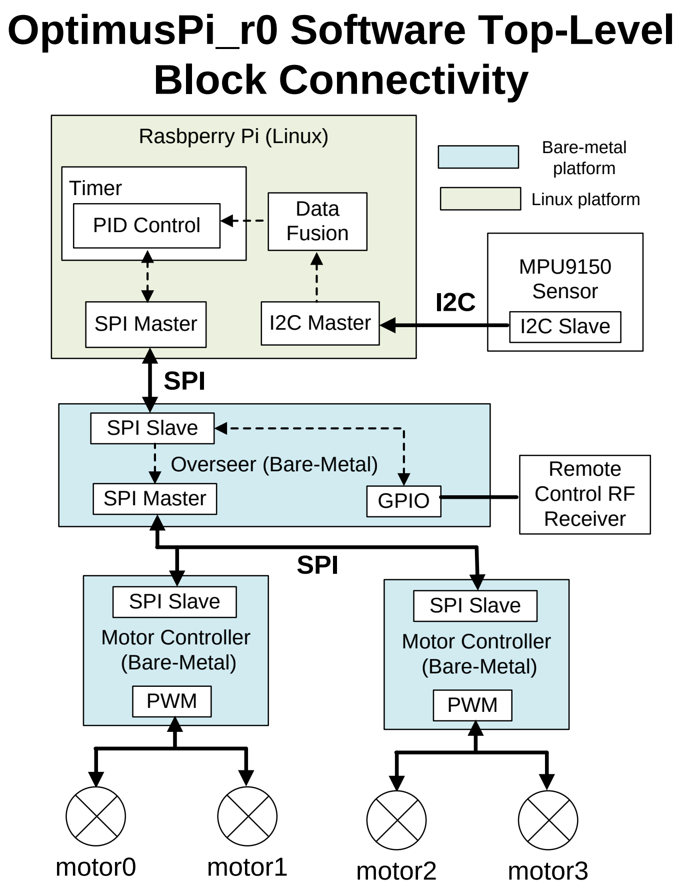


Figure 2 OptimusPi_r0 software top-level block connectivity diagram

OptimusPi_r0 was a software design project of a RF-controlled quadcopter controlled by a Raspberry Pi and three TI MCUs on OptimusPi board. The figure above shows a top-level of the software structure.

The Raspberry Pi collects real-time sensor data from MPU9150 sensor and collects remote control information from Overseer which connects to a remote control RF receiver. Then, these two pieces of information are fed into a sets of Proportional-Integral-Derivative (PID) control to calculate modification on rate of each motor to achieve such a control. Finally, these modification parameters are passed down from Raspberry Pi down to Motor Controllers in order to update motors status.

In this software project, all the code were written in C++. Code in TI MCUs were compiled by a TI toolchain (TI compiler + TI linker) while code in Raspberry Pi were compiled by a GCC (GNU Compiler Collection) toolchain (GCC compiler + GCC linker). Since some code in Raspberry Pi are Linux-dependent, such as I2C master driver, the code in Raspberry Pi was not regarded as board-independent and hence unable to be ported to Overseer directly.

4.4. OptimusPi_r1 specification

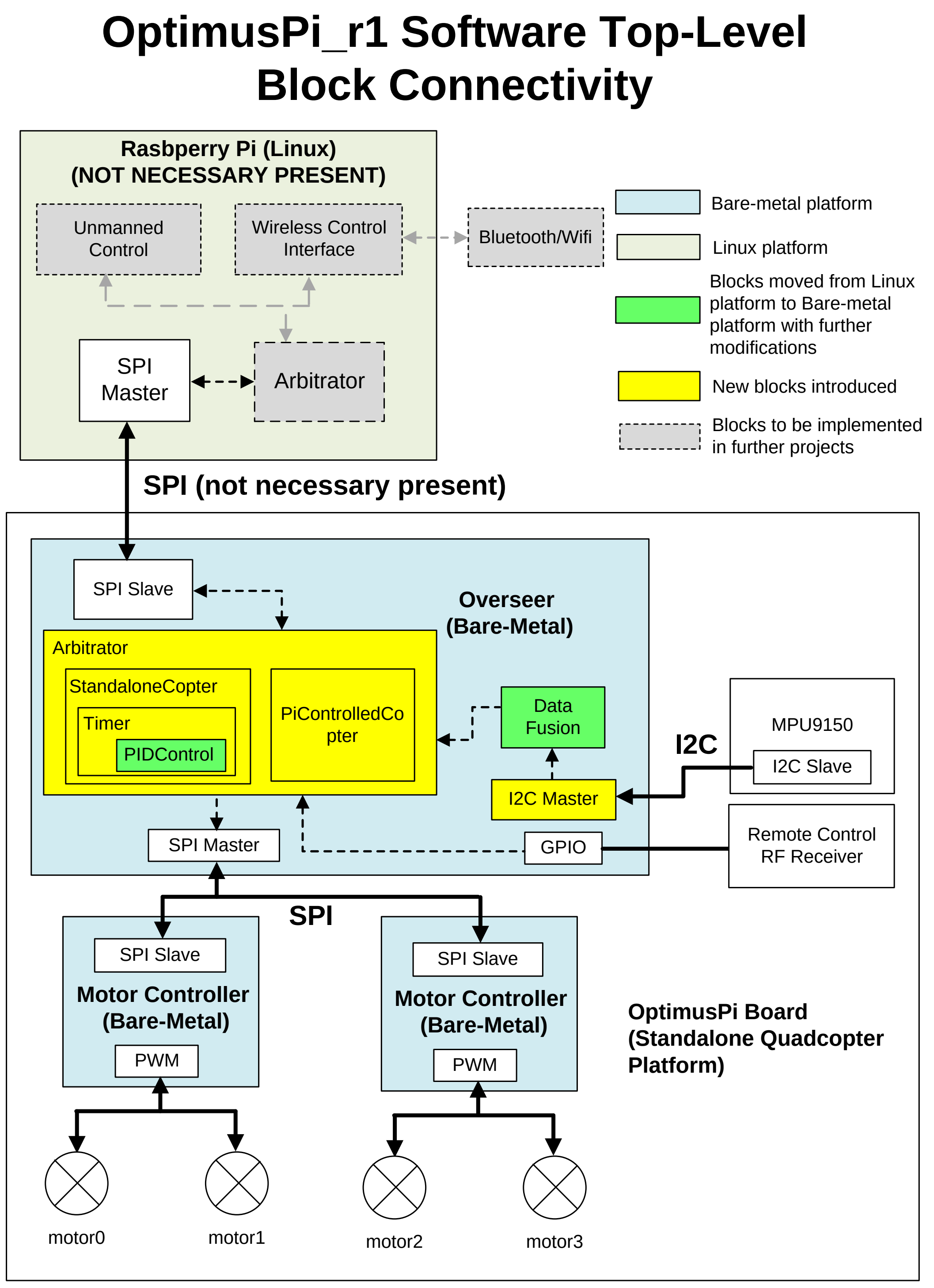

OptimusPi_r1 Software Top-Level Block Connectivity
Rasbperry Pi (Linux) (NOT NECESSARY PRESENT)
Unmanned Control
Wireless Control Interface
Bluetooth/Wifi
SPI Master
Arbitrator
Bare-metal platform
Linux platform
Blocks moved from Linux platform to Bare-metal platform with further modifications
New blocks introduced
Blocks to be implemented in further projects
SPI (not necessary present)
SPI Slave
Overseer (Bare-Metal)
Arbitrator
StandaloneCopter
Timer
PIDControl
PiControlledCopter
Data Fusion
I2C Master
I2C
MPU9150
I2C Slave
Remote Control RF Receiver
GPIO
SPI Master
SPI
SPI Slave
Motor Controller (Bare-Metal)
PWM
SPI Slave
Motor Controller (Bare-Metal)
PWM
OptimusPi Board (Standalone Quadcopter Platform)
motor0
motor1
motor2
motor3

Figure 3 OptimusPi_r1 Software Top-level Block Connectivity

OptimusPi_r1 was the second version of Optimus_r0 project. OptimusPi_r1 was a software project including

(1) Porting the code originally in Raspberry Pi down to Overseer while maintaining the same functionality including replacing Linux-dependent drivers as well as verifying the

code ported down functioning well. The top-level view of software structure is shown above.
(2) Development of an arbitrator switching quadcopter between a standalone flying mode, controlled by a RF remote control and a RPi-controlled flying mode, controlled by a Raspberry Pi attached to it. In the latter mode, the OptimusPi board acts as a hub of its peripheral drivers including motors under control of a Raspberry Pi.
(3) Packaging the use cases of software into a user library, such as peripheral drivers, high-level board-independent utilities, and properly documenting them in order to contribute supports for OptimusPi board.
(4) Documenting this quadcopter project well as a reference system for board users to develop a quadcopter easier on this board in order to contribute supports for OptimusPi board. The detail of this quadcopter software structure can be found in **9.9 OptimusPi_r1 Software Functional Level Hierarchy Specification**.

4.5. Design & Verification Plan

Since this project was mainly a respin of the previous project, it was planned out using a V model to guide the design and verification plan. The following were tasks broken down in the plan:

1) Peripheral driver level (board-dependent module): Writing the most bottom-level peripheral drivers for Overseer including timer and I2C master driver to replace the Linux-dependent peripheral driver originally in Raspberry Pi.
2) Computational level (board-independent module): Porting computational modules from Raspberry Pi down to Overseer and verifying them. Since in such modules some code had Linux dependencies, removing/replacing these dependencies might lead to malfunction of the code. Furthermore, Overseer and Raspberry Pi had different compilation toolchain, the programs generated even from the same piece of source code might perform differently due to a variation of the toolchain used in different platforms.
3) System level: Completion of a standalone quadcopter system, controlled by a RF remote control, as well as a RPi-controlled quadcopter system, controlled by a Raspberry Pi. Furthermore, providing a switching mechanism between these two system and therefore a Raspberry Pi can request Overseer to switch systems if it is present.
4) Library development for Overseer: Packaging module into user-friendly library for supporting OptimusPi board and rearrange class hierarchy if necessary.

5. Design & Verification

5.1. I2C master driver

In the Optimus_r0 project, I2C master driver was written as Linux-based driver for Raspberry Pi to support communication with MPU9150 sensor. (Details can be found in **9.4 OptimusPi_r0 Software Classes Hierarchy Specification**) In order to be compatible with higher-level module that uses this I2C driver, for example, MPU9150 driver class, the header file of this I2C master driver should retain the same while keeping the same functionality as well.

In order to write an I2C driver on Overseer, the following needs to be completed:
1) Understanding of I2C, and what supports available within TI MCU, i.e. TivaWare.
2) Understanding of how I2C driver worked in Raspberry Pi, what functionality it achieved and the I2C protocol that MPU9150 supports.
3) Write an I2C driver using TivaWare peripheral drivers to configure I2C hardware module to achieve this functionality.

It turned out TivaWare, an extensive suite of APIs of Tiva C Series MCU applications, only supported the most fundamental I2C protocol, not including other uses of I2C-bus communication protocols, such as System Management Bus (SMBus), CBUS, PMBus. [4] However, the MPU9150 sensor communicated with Raspberry Pi using a SMBus protocol on an I2C bus. So a SMBus driver had to been written at the presence of an I2C bus.

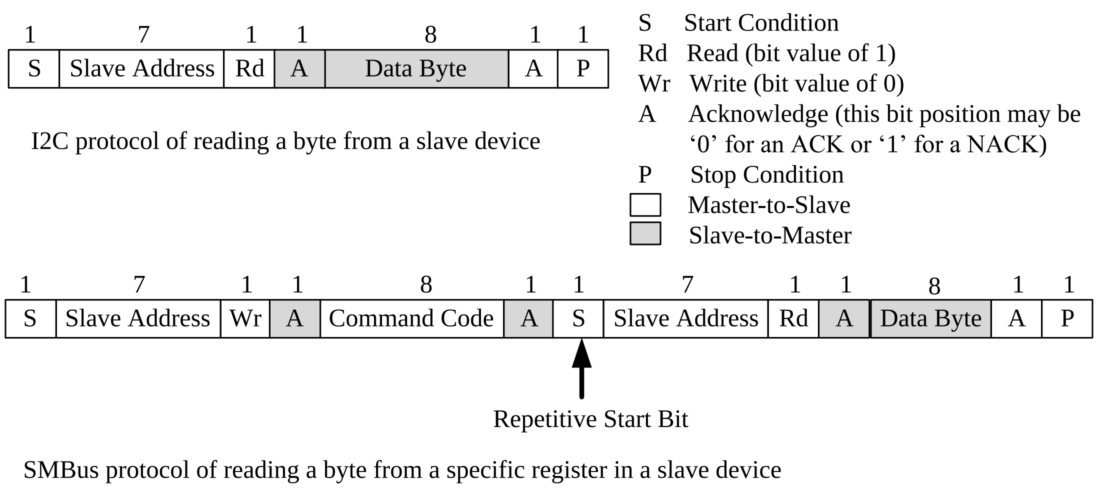

1 7 1 1 8 1 1
S | Slave Address | Rd | A | Data Byte | A | P
I2C protocol of reading a byte from a slave device
S Start Condition
Rd Read (bit value of 1)
Wr Write (bit value of 0)
A Acknowledge (this bit position may be '0' for an ACK or '1' for a NACK)
P Stop Condition
Master-to-Slave
Slave-to-Master
1 7 1 1 8 1 1 7 1 1 8 1 1
S | Slave Address | Wr | A | Command Code | A | S | Slave Address | Rd | A | Data Byte | A | P
Repetitive Start Bit
SMBus protocol of reading a byte from a specific register in a slave device

Figure 4 Reading a byte in I2C protocol and SMBus protocol [5]

For a common simple I2C protocol, a master can only define what slave I2C address is and the type of operation (Read/Write). For instance, in the above diagram, an I2C read-a-byte transaction starts from a start bit, followed by a slave address, and then the slave device sends a byte back to the master, finally ends with a stop bit. It is easy to realize that such a protocol can be easily used as communication between two active devices but not efficient to be used to access a passive device such as a sensor.

SMBus, one of whose class is with the same physical layer and data link layer but different network layer with I2C, [5] can be easily present on an I2C bus. In the above diagram, a SMBus read-a-byte transaction starts from a start bit, followed by a slave address, a commend code, normally a register address on the slave device, and then followed by another start bit (repetitive start bit), then followed by a common I2C read a byte protocol. The data sent from the slave device will be the value in the register address specified before. Such a protocol can support not only common I2C protocol feature but also support an easy access to registers in the passive slave device by specifying the register address within a transaction.

5.1.1. SMBus driver design

TivaWare supported transaction-level APIs only for the simple I2C protocol driver and this API considered a transaction to start from a start bit and to end with a stop bit. It was not allowed to have a repetitive start bit within a transaction, i.e. SMBus transaction was not permitted in such an API.

Therefore, in order to generate the repetitive start bit to form a SMBus driver, it was necessary to understand how to use TivaWare low-level I2C APIs to manipulate I2C hardware modules within TI MCU. If a simple I2C protocol can be broken down into pieces of operations and they could be then used to integrate into a SMBus protocol. However, there was little information in the manual of TivaWare I2C APIs because these APIs were just reading/writing hardware registers inside the I2C hardware modules. [6] At last, by reading through hardware register descriptions for the I2C hardware module [7], several hardware register configurations were interpreted as pieces of I2C operations within a transaction, and these pieces of I2C operations were then be integrated into a SMBus transaction. Finally this method was achieved by rearranging TivaWare low-level I2C APIs to integrate pieces of I2C operations into a SMBus transaction. The source code is shown below:

```
//read a byte of data from a specified register on slave device
int I2CClass::SMBusReadASingleByte(uint8_t slaveAddress, uint8_t registerAddress, uint8_t* buf)
```

```
{

    //specify the address of the slave device that we are writing to
    I2CMasterSlaveAddrSet(I2C0_BASE, slaveAddress, false);

    //specify register to be read
    I2CMasterDataPut(I2C0_BASE, registerAddress);

    //send control byte and register address byte to slave device
    //This generates a start bit and sends the register address only
    I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_BURST_SEND_START);

    //wait for MCU to finish transaction
    while(I2CMasterBusy(I2C0_BASE));

    //specify that we are going to read from slave device
    I2CMasterSlaveAddrSet(I2C0_BASE, slaveAddress, true);

    //send control byte and read from the register specified
    //This generates a repetitive start bit and read the data of the register
      followed by a stop bit, SMBus transaction finishes here.
    I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_SINGLE_RECEIVE);

    //wait for MCU to finish transaction
    while(I2CMasterBusy(I2C0_BASE));

    //return data pulled from the specified register
    *buf = I2CMasterDataGet(I2C0_BASE);

    return 0;
}
```

In the above source code, the two values of I2C control register did not form a simple I2C transaction. It started with `I2C_MASTER_CMD_BURST_SEND_START`, supposed to start a common I2C multiple-byte write operation while sending the first byte, and then it followed by `I2C_MASTER_CMD_SINGLE_RECEIVE`, supposed to start a common I2C single-byte read operation. The control register with the configuration `I2C_MASTER_CMD_BURST_SEND_START` forced the I2C master not to generate a stop bit after sending the first byte, and therefore the start bit generated from the control register with the configuration `I2C_MASTER_CMD_SINGLE_RECEIVE` was then regarded as the repetitive start bit. Only such an arrangement of the I2C configurations can form a protocol for the SMBus single-byte read operation (shown in **Figure 4 Reading a byte in I2C protocol and SMBus protocol**).

A SMBus write-a-byte method was implemented in a similar way as well. In order to reduce project time, other multiple-byte SMBus operations were all written by calling existing single-byte SMBus operations instead of using hardware I2C module to implement multiple-byte SMBus protocol. And it was verified that the performance of I2C driver in terms of communication with MPU9150 was sufficient by managing to receive a package of MPU9150 sensor data at 2 kHz.

5.1.2. SMBus driver verification

The verification of this driver was done by accessing registers on MPU9150. For verifying SMBus read-bytes method, a Read-Only ID register on MPU9150 has been used as a test source to produce a constant value for every test. By reading the value of this ID register via I2C, the functionality of the SMBus read-bytes method was verified as long as it returned the right value of this test register. The test code is shown below:

```
    //1.1.1 test readReg8 single byte functionality
    I2C_UUT.readRegisters8(MPU9150_ADDRESS_AD0_LOW,     MPU9150_RA_WHO_AM_I, 
&TP_Rreg8_signle, 1);
    while(TP_Rreg8_signle != 0x68){}
    //pass
```

For verifying SMBus write-bytes method, a Read-Write register on MPU9150 were used as a test source to produce a test point. This Read-Write register is really selective, it has to be (1) a non-critical register respect to I2C configuration in MPU9150 (2) a Read-Write register that does not act like a Write-Only register, i.e. the register can be written a value and this value can be read back later. By writing a value into this register and read it back, the SMBus write-bytes method were verified.

Such tests eventually have been added into a debug environment for boot-up time self-tests. As long as the debug switch is turned on, the self-test will be run before the user program starts. It can be used for board testing to test the connection of MPU9150 and each TI MCU on the OptimusPi board.

5.1.3. MPU9150 sensor data reading

After completion of the SMBus driver, a test case for reading sensor data has been written and run. MPU9150 support therefore was verified. In the later phase, when printf support was added into this environment, a debug switch was added to control if the sensor data were printed into the console, which supports future users for dumping the debug information of MPU9150 sensor data. The following are test output examples.

```
[CORTEX_M4_0] 3. Start Continuous MPU Tests.
MPU9150 connection verified
    rawData.x is 0.
    rawData.y is 0.
    rawData.z is 0.
    rawData.temp is 80.
    rawData.p is 63A0.
    rawData.q is 1.
    rawData.r is BBC.
    rawData.magx is 0.
    rawData.magy is 0.
    rawData.magz is 0.

    scaledSensorData.x is 0.021729.
    scaledSensorData.y is 0.004395.
    scaledSensorData.z is 0.980225.
    scaledSensorData.temp is 30.205883.
    scaledSensorData.p is -1.236641.
    scaledSensorData.q is -0.992366.
    scaledSensorData.r is 0.625954.
    scaledSensorData.magx is 0.000000.
    scaledSensorData.magy is 0.000000.
    scaledSensorData.magz is -8.015640.
```

5.2. Timer

Due to the limitation of the length of this report, the configuration of a periodic timer is only briefly illustrated.

The timer configuration has been completed by using TivaWare timer API [6] to configure hardware timer module inside the TI MCU. In order to maintain the same functionality of the whole system, it is critical to identify which hardware timer module in Overseer were used by other modules, and to set the right priority of this timer to keep the system performing as expected. It turned out that four 16-bit timer and four 32-bit wide timers have been used for decoding PWM signals for a RF receiver. After the right timer selected and right priority set, the timer was run as expected. (There was actually a bug found within the latest version of TI MCU API on reading system clock frequency during setting the load value of a timer. This bug has been confirmed from TI [8], more details can be found in the timer source code [9])

5.3. Sensor fusion

The sensor fusion module, as known as attitude and heading reference system (AHRS), fuses data produced from multiple sensors to filter out the noise in order to provide an estimation of orientation of the board in space. [10] This sensor fusion module is a purely computational module. It was implemented based on a mathematical matrix library called Eigen [11], which was originally designed for running on an operating system because it had lots of Linux dependencies. Disabling Linux assertions by turning on a global switch NDEBUG could not completely remove all Linux dependencies. Therefore, extra removal of Linux dependencies has to be done in order to compile in a bare-metal TI MCU. Such a library was regarded as not enough confident to run on a bare-metal system. As expected, although the sensor fusion module managed to be compiled in the Overseer, it did not perform well when filtering noise from the data measured in the MPU9150 sensor in the first place.

Due to the complexity of the math library Eigen, it was not worth to test every matrix operation used in the source code by writing another piece of test code. This might take much longer project time than using the sensor fusion code itself. Therefore, the verification was carried out by running a same module in both Raspberry Pi and Overseer using a same set of test inputs. By comparing the test outputs produced, the functionality of this module can be considered as verified if the test outputs from them are the same. And if the test outputs are not the same, insertions of lots of checkpoints by using printf would help debug where the variations are produced.

5.3.1. Printf support

The current project did not support printf-like support in the first place. The debug ability provided was mainly a combination of setting breakpoints and accesses to registers. However, to efficiently test a computational module like a sensor fusion module, it was necessary to provide a printf-like support in order to dump a large group of data for comparisons between an expected output and actual test output.

As the OptimusPi board was regarded as a shield of a Raspberry Pi, the initial printf implementation was considered to use SPI communication system that Matthew Watson created [2] to forward the printf formatted strings from the Overseer up to the Raspberry Pi. But it turned out the bandwidth that SPI communication system was definitely not sufficient enough to provide a reasonably fast printf function.

Then an URAT solution was considered to provide printf support. But at that time, this solution was considered as too costly in terms of project time.

Finally, a JTAG solution to provide printf support was completed. This was the most intuitive solution since there was already a JTAG port for loading program as well as debugging. But at the first time, there were not many supports online to provide a JTAG solution for TI IDE to support printf via JTAG. Fortunately, a TI wiki page about supporting printf via JTAG in another processor was found. [12] And this support happened to work in this TI MCU as well. (It was reasonable that printf via JTAG was a board-independent support. Because as long as the MCUs has a JTAG interface and the same MCU architecture, the high-level printf support should work on such MCUs) When printf via JTAG was enabled, the debugger would set a breakpoint to where the printf functions were, and pull the formatted strings in printf out via JTAG and display them into console.

5.3.2. Test environment set-up and test results

The test stimulus of a sensor fusion module was a set of sensor data generated from MPU9150 on board. A test stimulus could be automatically generated in the console by using printf support to write a script. An example is shown below:

```
//This is MPU9150 scaledSensoredData in AHRS before AHRS.update().fuse(dt)
//function. dt=0.05 <=> 200Hz.
//AHRS.update()              only              run              getSensors();
//MerayoCalibClass::apply(&scaledSensorData.x,         &scaledSensorData.y,
//&scaledSensorData.z, &accCalibData).


//This is TestData set 0.
TestData.at(0).X = 0.031491;
TestData.at(0).Y = -0.642912;
TestData.at(0).Z = 0.885281;
TestData.at(0).P = -39.465649;
TestData.at(0).Q = -19.389313;
TestData.at(0).R = 10.854961;
TestData.at(0).Temp = 27.597059;
```

After generation of a set of test stimulus, a Raspberry Pi compilation on the same piece of module had to be completed. A Raspberry Pi program could be cross-compiled using a C++ eclipse in a Linux operating system on a PC, like Ubuntu. [13] An USB Wifi module provided the Raspberry Pi internet access and therefore the program could be loaded into a Raspberry Pi within A local area network (LAN) via a secure shell (SSH).

[13] A SSH provided a secure communication between a Raspberry Pi and a host PC. [1] Then, a Raspberry Pi development environment was set up by using such a setting. And the test stimulus generated were ready to be imported into the same computational module in both the Raspberry Pi and the Overseer. By comparing the test point values (printf directed values to the console) inside the computational module, a bug in the module source code managed to be located. The following was a piece of an example computational module with test points and followed by an example output generated by such a code:

```
// Source code with test points
    x = x + K * y;

    #ifdef DEBUG_EXTENDEDKALMAN
    std::cout << "Matrix x is " << std::endl << x(0) << std::endl << x(1) <<
std::endl << x(2) << std::endl << x(3) << std::endl;
    #endif
```

```
// Test results generated by such a code
Matrix x is
0.953422
0.301277
0.0147571
0
```

5.3.3. Bug introduced due to TI compiler

It turned out the malfunction of the sensor fusion module was not due to the incompatibility of the complex mathematical matrix library Eigen. It is nevertheless a bug introduced due to the variation of the compiler. This bug did not occur when the code was compiled in the Raspberry Pi (using GCC toolchain) but occurred when compiled in the Overseer (using TI toolchain). The following code was the source code with a bug in it:

```
// return the quaternion object into a local instance
quaternion = EKF.update(x,y,z,dt);
```

On the right hand side (RHS), *EKF.update()* method returned an instance of *QuaternionClass* class while on the left hand side (LHS), *quaternion* was an instance of *QuaternionClass* class as well. And *QuaternionClass* was a user-defined class.

GCC compiler generated two copy constructors for every class (including user-defined class and predefined class) in default. One of them was overloading assignment operator, i.e. *operator=*, into a copy constructor. Every time an instance was assigned into another instance of the same class, e.g. *Obj_A=Obj_B*, the LHS instance called its copy constructor and copies the contents of the RHS instance into itself. Therefore, in the above example, *quaternion* contained the data that *EFK.update()* returned, which was as expected to simplify the operation of copying an object into the operation of copying an element like an integer type.

However, TI compiler was not intelligent enough to automatically overload the assignment operator into a copy constructor. It interpreted the assignment as copying the address of the RHS instance into the reference of the LHS instance. Thus, at the time the assignment completed, both *quaternion* and the returned object from *EFK.update()* were pointing to the same object. But since the returned object were locally created within the method, it was generated in the stack. After this assignment finished, the processor destroyed the returned object by freeing up the memory used to store the returned object. As long as another stack-used operation happened, it likely overwrote this memory and the RHS instance then pointed into some random data. This caused *quaternion* did not contain the right contents from *EFK.update()*.

In the C++98 standard, it was supposed to be responsibility of developers to ensure that what *assignment operator=* did to a user-defined class when the compiler handled this class with an assignment operator. Nevertheless, TI compiler did not generate any warnings or errors about the compilation of this piece of code and it was considered as a terrible deficiency since it dramatically increased the difficulty of porting code from a common GCC toolchain into a TI toolchain and forced the programmer to gain the awareness of the differences between these two toolchains in order to debug the program.

At last, after this bug fixed, the sensor fusion module in the Overseer performed the same as the module in the Raspberry Pi.

5.4. System development

Due to the limited length of this report, this section is briefly described. More details can be derived from the source code [9] and documentations in the section **9. Appendix**.

In order to produce more reference systems for use cases of this board on supporting quadcopter design project for future users, it was desirable to not only complete two existing systems but also a mechanism switching between these two systems during program run-time. These two system were a standalone quadcopter, running on its own and controlled by a RF remote control, and a RaspberryPi-controlled quadcopter, running on the control of a Raspberry Pi. The system specification can be found in section **9.9 OptimusPi_r1 Software Functional Level Hierarchy Specification**

5.4.1. StandaloneCopter system

This system was completed by porting the code from the Raspberry Pi in project OptimusPi_r0 into the Overseer, including low-level driver replacements and computational modules verification. However, due to shortage of the project time, the system verification has not been carried out on a quadcopter hardware prototype yet. The progress made on it could be evaluated as the changes done, shown in section **9.7 OptimusPi_r1 Software Classes Hierarchy Specification with Change Highlighted**.

5.4.2. RPiControlledCopter system

This system had relatively less changes done. It was basically a remaining system from OptimusPi_r0 added an uploading process on sensor data collected from MPU9150. The progress made on it could be evaluated as the changes done, shown in section **9.7 OptimusPi_r1 Software Classes Hierarchy Specification with Change Highlighted**. This system has not been verified because the updated SPI communication system has not been verified. The reason is explained below.

5.4.3. SPI communication system update

There were quite a few SPI commands updated in order to support sensor data fetching and system run-mode switching. However, because the software in the Raspberry Pi has not been updated yet due to limited project time (this was really costly in project time since there were lots of set-up on Raspberry Pi to do in order to have the source code compiled and the program properly run. But none of these set-up have been documented in OptimusPi_r0. Finding out the set-up on Raspberry Pi to run a previous software was not regarded as a high priority task in this project context), the updated SPI communication system was not verified yet.

5.4.4. System-mode arbitrator

The system is running one of two modes in Overseer system: StandaloneCopter or RPiControlledCopter. The system-mode arbitrator was switching the system between these two run modes. When running in any of these modes, the system would dynamically create its mode class. And only one of them could exist in this system at a time. Every time the system was switching system mode, the instance of the first mode have to be properly deinitialised by running its destructor, and the memory allocated would be deleted. Then, the system-mode arbitrator created the instance of next mode as well as initialized the system by running its constructor. Such a switching mechanism would reduce the instability of the system and have the system with better memory management.

The default mode was set as StandaloneCopter mode. If a Raspberry Pi is present, as long as SPISlave in the Overseer receives a specific RPiControlledCopter mode SPI command, the Overseer will switch to the RPiControlledCopter mode. Similarly, If the copter is running RPiControlledCopter mode, as long as SPISlave in the Overseer receives a specific StandaloneCopter mode SPI command,the Overseer will switch to the StandaloneCopter mode.

During the verification of this arbitrator module, there was a run-time bug: the dynamical allocation of memory of system mode instances did not well perform. By stepping through the low-level dynamical allocation TI support, it was deduced as insufficient memory left to create such a large instance. However, a run-time storage SRAM with 32KB [14] in TI MCU should have been large enough for such a program. Eventually, it turned out this was due to the intelligence of TI linker. The size of the heap and the stack had to be set manually via TI linker settings. By setting the heap size into 16KB, the run-time bug was fixed.

In an all, none of the above all four were fully tested because most of them required either a hardware prototype or a Raspberry Pi with a compiled updated program to complete the test. (Both of them are costly to carry out in terms of project time)

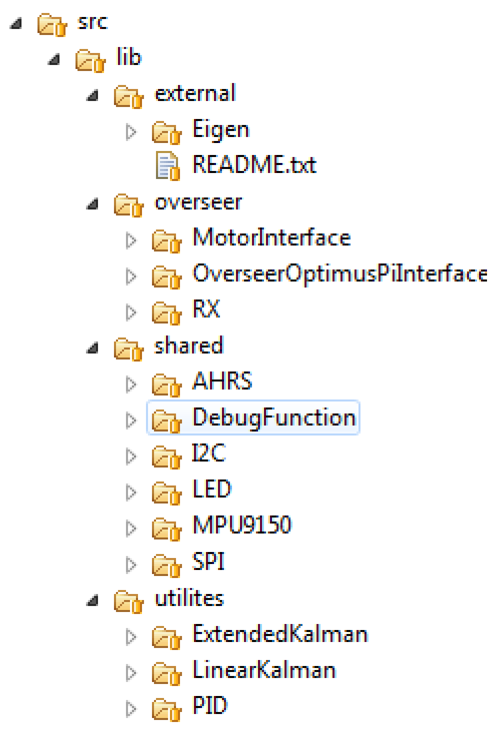
5.5. Others

There were other modules that have been tested as well, such as GPIO, LEDs, RX interface for RF remote controller, but due to the limit length of this report, these progresses were skipped within this report. More details can be found in the source code [9].

6. OptimusPi Library

Due to the limited length of this report, this section is briefly described. More details can be derived from the source code [9].

The library were divided into four categories: external, shared, overseer, and utilities.



```
src
  lib
    external
      Eigen
      README.txt
    overseer
      MotorInterface
      OverseerOptimusPiInterface
      RX
    shared
      AHRS
      DebugFunction
      I2C
      LED
      MPU9150
      SPI
    utilites
      ExtendedKalman
      LinearKalman
      PID
```

Figure 5 OptimusPi Library directory structure

(1) The external library contains the libraries not created by this project, in this case, a mathematical matrix library Eigen.

(2) The shared library contains the libraries that could be used in both the Overseer TI MCU and the Motor Controllers TI MCUs, such as LEDs, I2C drivers, SPI drivers.

(3) The overseer library contains the libraries that could only be used in the Overseer TI MCU. This is mainly because the Overseer connects lots of peripheral including GPIOs. And these libraries are the peripheral drivers as well as the high-level driver as a wrapper of these low-level drivers, such as the remote control receiver interface.

(4) The utilities library contains the board-independent computational libraries for using a quadcopter design, such as Kalman filter, and PID controllers.

7. Conclusion

The original objectives were to build up a functioning quadcopter by doing the following. Firstly porting the code from the Raspberry Pi in the previous project into the TI MCU. Secondly implementing more support for the OptimusPi board. However only the latter has been achieved. A fully functioning quadcopter was not completed due to the limited project time as a system-level verification was not possible.

The evaluation of every level in design and verification is shown below:

1) In the physical driver level, a few self-tests have been written to ensure the replacement driver managed to replicate the functionality of the previous project in another platform, i.e. Raspberry Pi. All tests were successful.
2) At the computational module level, a test was run to ensure that the computational module had the same functionality when it was originally on the Raspberry Pi platform.
3) At the system level, due to the limited project time, only the implementation of the design was completed. However verification was not carried out.
4) In terms of creating support of the OptimusPi board, a user-friendly library was created to support future users of the OptimusPi board. This would make it easier to build up the quadcopter platform. Furthermore, lots of documentations was to provide guidance and references for future use of the OptimusPi board.

In section **9.1.1 Completion** details on the progress of the project can be found, which is in section **9.1 OptimusPi_r1 Software Release Notes**. Nevertheless, there are still lots of further improvements that could be completed in future projects on the OptimusPi board. The details has been summarized in section **9.1.2 Further work in the future** from **9.1 OptimusPi_r1 Software Release Notes.**

8. References

[1] A. Stevens, “The Design and Fabrication of a Quadrotor Platform,” University of Sheffield, Sheffield, 2014.

[2] M. Watson, “OptimusPi Project,” University of Sheffield, Sheffield, 2015.

[3] Texas Instrument, “1.1 Tiva™ C Series Overview,” in *Tiva TM4C123AH6PM Microcontroller Data Sheet*, Austin, Texas Instrument Incorporated, 2014, p. 40.

[4] N. Semiconductors, I2C-bus specification and user manual, Eindhoven: NXP Semiconductors. N.V., 2014.

[5] S. I. Forum, System Management Bus (SMBus) Specification Version 2.0, Texas: SBS Implementers Forum, 2000.

[6] T. Instrument, TivaWare™ Peripheral Driver Library, Texas: Texas Instrument, 2014.

[7] T. Instrument, “Register 2: I2C Master Control/Status (I2CMCS), offset 0x004,” in *Tiva TM4C123AH6PM Microcontroller Data Sheet*, Austin, Texas Instrument Incorporated, 2014, p. 966.

[8] A. Ashara, "set Tiva C series 80MHZ," Texas Instrument, 29 April 2015. [Online]. Available: http://e2e.ti.com/support/microcontrollers/tiva_arm/f/908/t/343830. [Accessed 29 April 2015].

[9] T. Li, “relliky/OptimusPi_r1,” University of Sheffield, 29 April 2015. [Online]. Available: https://github.com/relliky/OptimusPi_r1. [Accessed 29 April 2015].

[10] M. Watson, “The Design and Implementation of a Robust AHRS for Integration into a Quadrotor Platform,” University of Sheffield, Sheffield, 2013.

[11] Tuxfamily, “Eigen,” Tuxfamily, 29 April 2015. [Online]. Available: http://eigen.tuxfamily.org/index.php?title=Main_Page. [Accessed 29 April 2015].

[12] T. Intrument, “Printf support for MSP430 CCSTUDIO compiler,” Texas Intrument, 29 April 2015. [Online]. Available: http://processors.wiki.ti.com/index.php/Printf_support_for_MSP430_CCSTUDIO_compiler. [Accessed 29 April 2015].

[13] H. Al-Hertani, “Development Environment for the Raspberry Pi using a Cross Compiling Toolchain and Eclipse,” Hertaville, 7 September 2014. [Online]. Available: http://hertaville.com/2012/09/28/development-environment-raspberry-pi-cross-compiler/. [Accessed 30 April 2015].

[14] T. Instrument, “Table 2-6. SRAM Memory Bit-Banding Regions,” in *Tiva TM4C123AH6PM Microcontroller Data Sheet*, Austin, Texas Instrument Incorporated, 2014, pp. 89-90.

[15] T. Li, "Platform Crossing Development on Previous Quadcopter Project from Linux to Bare-metal System," University of Sheffield, Sheffield, 2015.

9. Appendix

9.1. OptimusPi_r1 Software Release Notes

9.1.1. Completion

Moved the code from Raspberry Pi down to Overseer, including:

(1) Added MPU9150 support, I2C/SMBus support on Overseer, and SPI commands for passing sensor data.
(2) Added general-purpose timer on Overseer
(3) Added printf support via JTAG
(4) Added test environment for self-checking for peripheral drivers as well as debugging other codes
(5) Added Arbitrator to switch fly mode between StandaloneCopter (controlled by RF controller) or PiControlledCopter (controlled by RasbperryPi)
(6) Fixed run-time bugs when code was moved from Raspberry Pi down to Overseer
(7) Created remote control TX documentation. (kill switch documentation)
(8) Created lots of other docs and supports

9.1.2. Further work in the future

The following are tasks with high priority:

(1) To build up a quadcopter hardware prototype

(i) Most of components can be found in the component list, which seats in the OptimusPi_r1 directory

OptimusPi_r1/Documentations/Optimuspi_projects_documents/OptimusPi_r0_documents/Created_By_Matthew/Quadcopter Component List.xlsx

Using the components provided above can assemble a hardware prototype

(ii) To be noticed, when testing motors, each motor has power 100W with voltage 12V, i.e. rating current is 8.3A. Just four motors in the full power require about 33A. However, the maximum current of power supply in the lab currently is 18A, which is definitely not sufficient enough to drive all the four motors.

(2) StandaloneCopterClass in src/TopLevel/StandaloneCopter/StandaloneCopter.cpp

(i) To verify the quadcopter hardware

(ii) RF controller sw2 is the kill switch of the StandaloneCopter system.

(3) RPiControlledCopterClass in src/TopLevel/RPiControlledCopter/ RPiControlledCopter.cpp

(i) To verify SPI mode-switching mechanism.

(ii) To build a new update mechanism. Currently RPiControlledCopterClass uses an infinity loop to update system, and the Overseer passes the raw data of sensor MPU9150 up to Raspberry Pi and Raspberry Pi is filtering noise itself. In the future project, instead of using an infinity loop to update system, using a timer to call an ISR to update system would give a finite system update rate, which would enable Overseer to filter sensor noise itself by using the Extended Kalman Filter in the AHRS Class. (N.B. this mechanism has been implemented in the StandaloneCopterMode class, which would be a good reference to build this mechanism in RPiControlledCopter Class) To be noticed, the update rate required for AHRS class is normally between 100Hz and 400Hz (quoted from Matthew Watson), and therefore, the SPI updating system in RPiControlledCopter needs to meet this time constrain. It is necessary to verify that the longest updating cycle of RPiControlledCopter, i.e. RPiControlledCopterClass::running() method, is sorter than the fastest AHRS update rate: 1/400Hz = 2.5ms.

The following are tasks with medium priority:

(1) More driver supports ported from Sensorhub TI example project. Documentations can be found under OptimusPi_r1 directory OptimusPi_r1\Documentations\TivaWare_C_Seires_Documents\ TivaWare_Sensorhub_Example_Project_on_Tiva_TM4C1294NCPDT.pdf

The following are tasks with low priority:

(1) To free up SRAM.

(i) only 32KB SRAM for run-time memory for the processor, and the processor currently uses 16KB for the heap(necessary for printf support), 8KB for the stack(only less than 4KB of the stack size is currently required, just in case for future projects, the stack size is set as 8KB). It is necessary to simplify some functional drivers to reduce the size of memory used.

(2) I2C class in src/shared/I2C/I2C.cpp

(i) Complete both SMBus and I2C support in I2C driver.

(3) AHRSClass in src/shared/AHRS/AHRS.cpp

(i) To build a simple In AHRS class, the Extended Kalman Filter uses a math matrix library Eigen. This math library is really costly in terms of CPU time. Reduction on the time running AHRS class per system update cycle can provide more bandwidth for other code to run. Furthermore, there is only 32KB SRAM for run-time memory in the processor.

(ii) AHRS Class inherits from its parent. Usage of inheritance is not common in this project, which should be consistent throughout the project.

(4) RX documentation update

(i) More detail illustration of sw1 function which switch attitude control and rate control

(ii) To add sw1/sw2 location and function description in rx.cpp

(5) Global variable definitions

(i) Currently all the static variables (inside .cpp scope) as static points

(most of them serves a ISR) were moved into a global variable header. and used extern to import them back. Although it will make these pointers globally, not static in a cpp scope only. But it helps to manage these non-local static variables to avoid linker problems as well as less error-prone when the code is ported outside of this project. TO-DO: These interrupt driver should be written in C instead of C++ in the future so as long as proper C drivers are included, instantiating a class would give rise to a new interrupt module instead of adding another static non-local variables like now.

9.2. OptimusPi_r1 HobbyKing Tx Illustrations

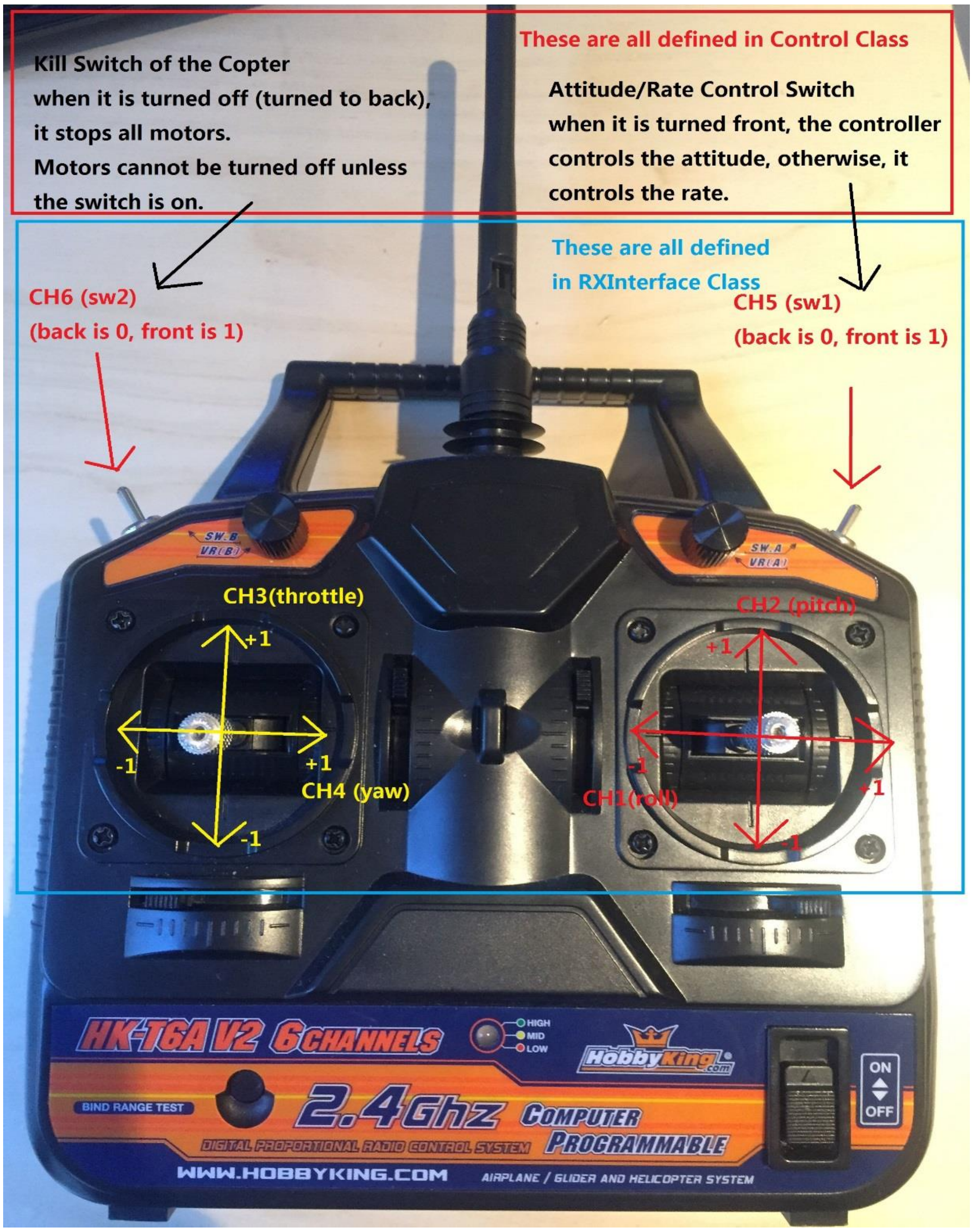
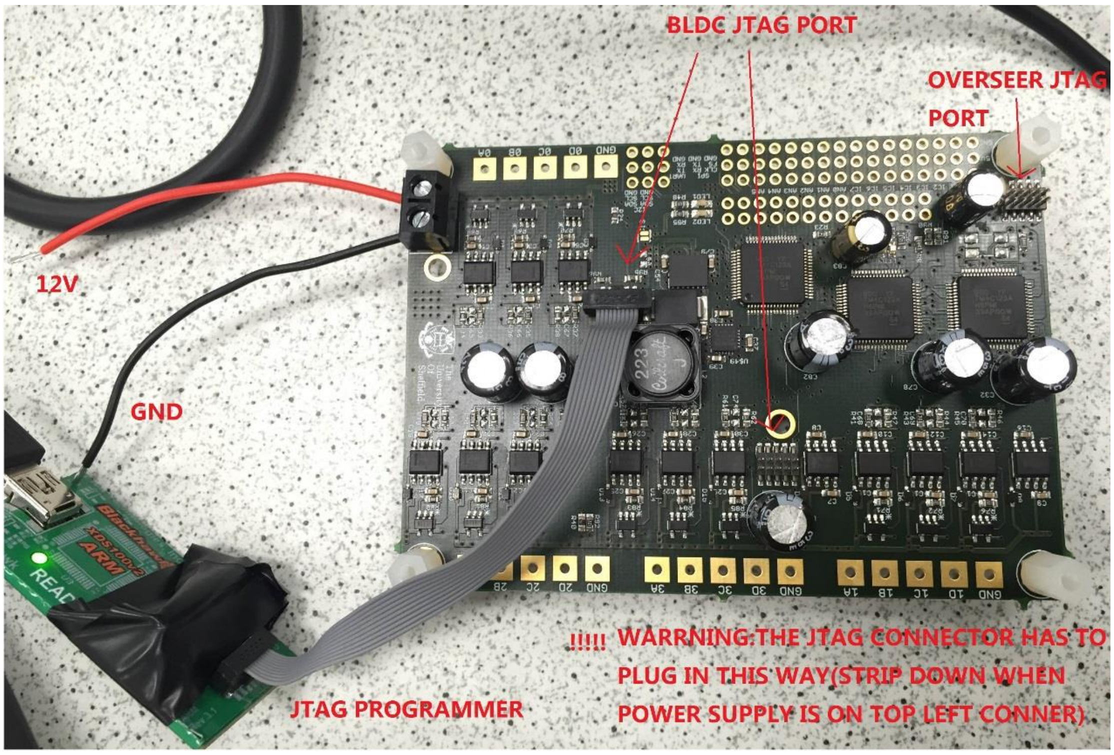

Kill Switch of the Copter
when it is turned off (turned to back),
it stops all motors.
Motors cannot be turned off unless
the switch is on.
These are all defined in Control Class
Attitude/Rate Control Switch
when it is turned front, the controller
controls the attitude, otherwise, it
controls the rate.
These are all defined
in RXInterface Class
CH6 (sw2)
(back is 0, front is 1)
CH5 (sw1)
(back is 0, front is 1)
CH3(throttle)
+1
-1
+1
-1
CH4 (yaw)
CH2 (pitch)
+1
-1
-1
+1
CH1(roll)
HK-T6A V2 6CHANNELS
2.4Ghz COMPUTER PROGRAMMABLE
HobbyKing.com
WWW.HOBBYKING.COM

Figure 6 OptimusPi_r1 HobbyKing Tx Illustrations

9.3. Programming the board

i) Select the right JTAG port

The board has 3 JTAG ports, two for bottom-level motor controllers, and one for top-level controllers, i.e. Overseer. To program any of controllers, the JTAG cable has to be plugged in with strip down, and the USB cable should not be plugged in until the JTAG cable connected board. More detail is shown in the following diagram.


BLDC JTAG PORT
OVERSEER JTAG PORT
12V
GND
JTAG PROGRAMMER
!!!!! WARRNING:THE JTAG CONNECTOR HAS TO PLUG IN THIS WAY(STRIP DOWN WHEN POWER SUPPLY IS ON TOP LEFT CONNER)

ii) CCS set-up

1) Download the CCS from ti.com and load a copy of Tai's CCS project code as the workspace.

2) (There used to be path setting for including TivaWare library. But now the including path uses relatively path by using environment variables of CCS projects. So there is no need to reset path for TivaWare as long as TivaWare stays where it relatively is in the CCS projects) No extra setting needs to be done for CCS projects as long as the OptimusPi_r1 project downloaded from https://github.com/relliky/OptimusPi_r1.git

3) Build project "Overseer [Debug - Active]" (Clean the project first before to build it first time)

9.4. OptimusPi_r0 Software Classes Hierarchy Specification

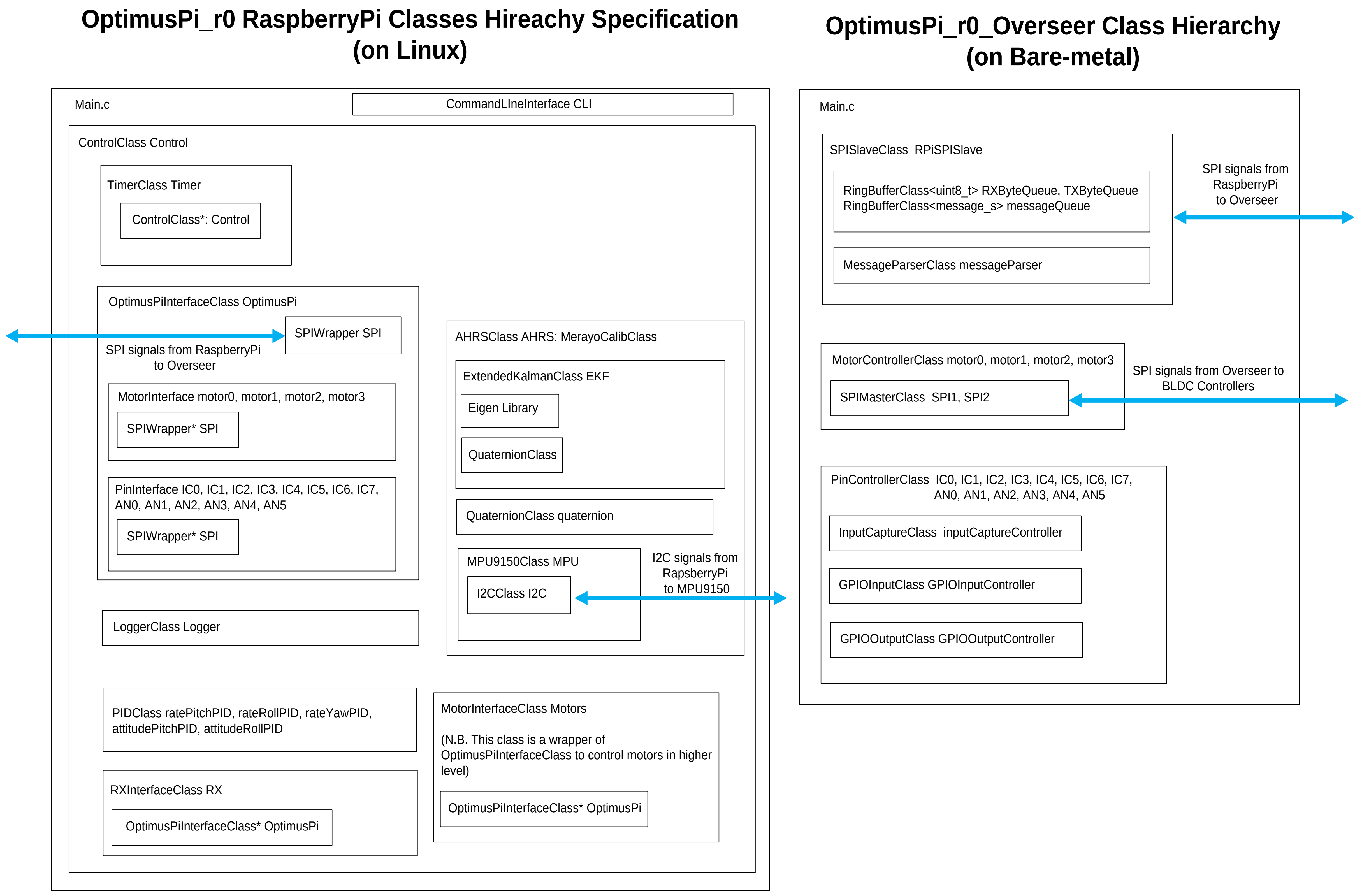
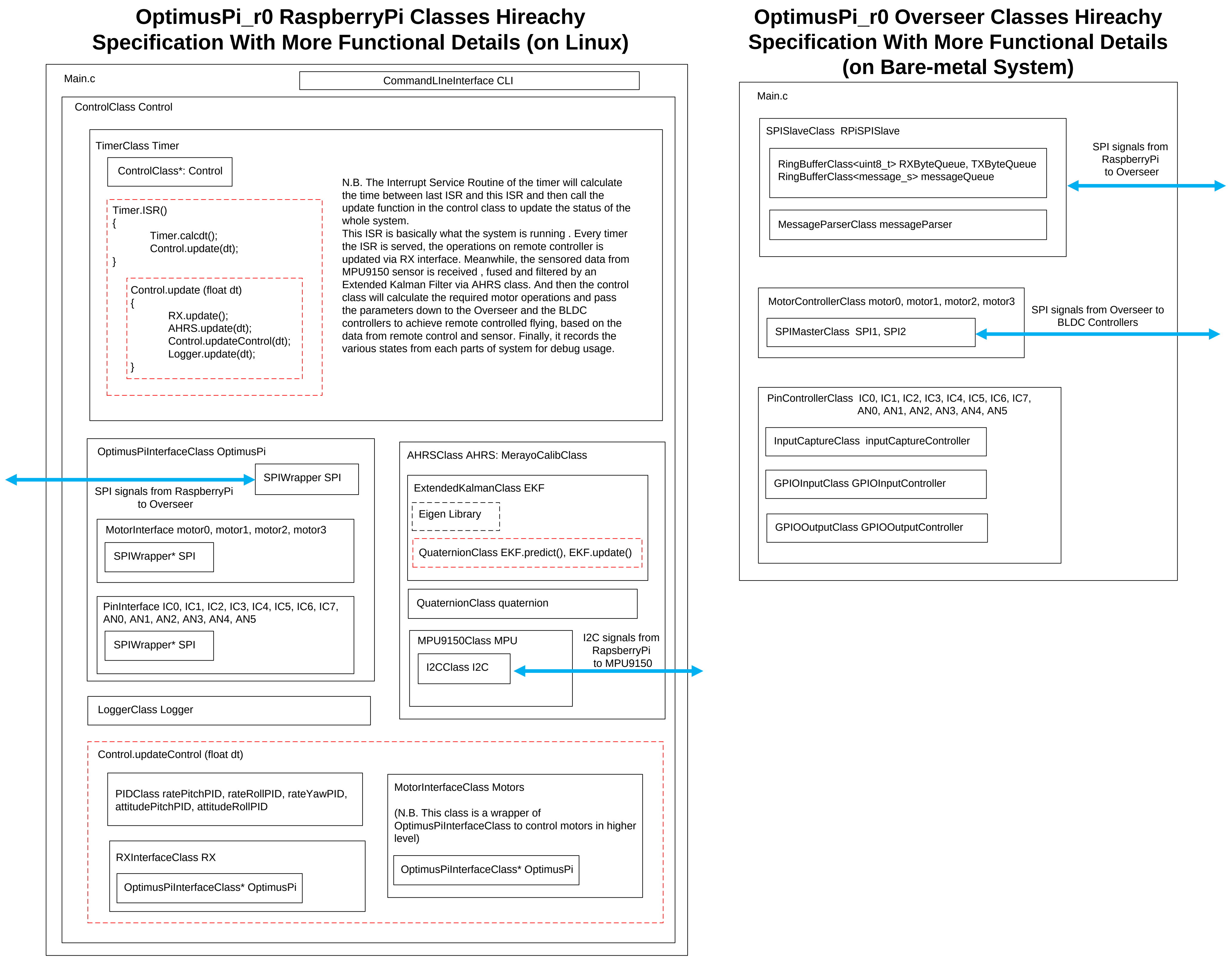

OptimusPi_r0 RaspberryPi Classes Hireachy Specification (on Linux)
Main.c
CommandLIneInterface CLI
ControlClass Control
TimerClass Timer
ControlClass*: Control
OptimusPiInterfaceClass OptimusPi
SPIWrapper SPI
SPI signals from RaspberryPi to Overseer
MotorInterface motor0, motor1, motor2, motor3
SPIWrapper* SPI
PinInterface IC0, IC1, IC2, IC3, IC4, IC5, IC6, IC7, AN0, AN1, AN2, AN3, AN4, AN5
SPIWrapper* SPI
LoggerClass Logger
AHRSClass AHRS: MerayoCalibClass
ExtendedKalmanClass EKF
Eigen Library
QuaternionClass
QuaternionClass quaternion
MPU9150Class MPU
I2CClass I2C
I2C signals from RapsberryPi to MPU9150
PIDClass ratePitchPID, rateRollPID, rateYawPID, attitudePitchPID, attitudeRollPID
RXInterfaceClass RX
OptimusPiInterfaceClass* OptimusPi
MotorInterfaceClass Motors
(N.B. This class is a wrapper of OptimusPiInterfaceClass to control motors in higher level)
OptimusPiInterfaceClass* OptimusPi
OptimusPi_r0_Overseer Class Hierarchy (on Bare-metal)
Main.c
SPISlaveClass RPiSPISlave
RingBufferClass<uint8_t> RXByteQueue, TXByteQueue
RingBufferClass<message_s> messageQueue
MessageParserClass messageParser
SPI signals from RaspberryPi to Overseer
MotorControllerClass motor0, motor1, motor2, motor3
SPIMasterClass SPI1, SPI2
SPI signals from Overseer to BLDC Controllers
PinControllerClass IC0, IC1, IC2, IC3, IC4, IC5, IC6, IC7, AN0, AN1, AN2, AN3, AN4, AN5
InputCaptureClass inputCaptureController
GPIOInputClass GPIOInputController
GPIOOutputClass GPIOOutputController

Figure 7 OptimusPi_r0 Software Classes Hierarchy Specification

9.5. OptimusPi_r0 Software Classes Hierarchy Specification with More Details


OptimusPi_r0 RaspberryPi Classes Hireachy Specification With More Functional Details (on Linux)
Main.c
CommandLIneInterface CLI
ControlClass Control
TimerClass Timer
ControlClass*: Control
OptimusPiInterfaceClass OptimusPi
SPIWrapper SPI
SPI signals from RaspberryPi to Overseer
MotorInterface motor0, motor1, motor2, motor3
SPIWrapper* SPI
PinInterface IC0, IC1, IC2, IC3, IC4, IC5, IC6, IC7, AN0, AN1, AN2, AN3, AN4, AN5
SPIWrapper* SPI
LoggerClass Logger
AHRSClass AHRS: MerayoCalibClass
ExtendedKalmanClass EKF
Eigen Library
QuaternionClass EKF.predict(), EKF.update()
QuaternionClass quaternion
MPU9150Class MPU
I2CClass I2C
I2C signals from RapsberryPi to MPU9150
Control.updateControl (float dt)
PIDClass ratePitchPID, rateRollPID, rateYawPID, attitudePitchPID, attitudeRollPID
RXInterfaceClass RX
OptimusPiInterfaceClass* OptimusPi
MotorInterfaceClass Motors
OptimusPiInterfaceClass* OptimusPi
OptimusPi_r0 Overseer Classes Hireachy Specification With More Functional Details (on Bare-metal System)
Main.c
SPISlaveClass RPiSPISlave
RingBufferClass<uint8_t> RXByteQueue, TXByteQueue
RingBufferClass<message_s> messageQueue
MessageParserClass messageParser
SPI signals from RaspberryPi to Overseer
MotorControllerClass motor0, motor1, motor2, motor3
SPIMasterClass SPI1, SPI2
SPI signals from Overseer to BLDC Controllers
PinControllerClass IC0, IC1, IC2, IC3, IC4, IC5, IC6, IC7, AN0, AN1, AN2, AN3, AN4, AN5
InputCaptureClass inputCaptureController
GPIOInputClass GPIOInputController
GPIOOutputClass GPIOOutputController

```
Timer.ISR()
{
        Timer.calcdt();
        Control.update(dt);
}

Control.update (float dt)
{
        RX.update();
        AHRS.update(dt);
        Control.updateControl(dt);
        Logger.update(dt);
}
```

N.B. The Interrupt Service Routine of the timer will calculate the time between last ISR and this ISR and then call the update function in the control class to update the status of the whole system.
This ISR is basically what the system is running . Every timer the ISR is served, the operations on remote controller is updated via RX interface. Meanwhile, the sensored data from MPU9150 sensor is received , fused and filtered by an Extended Kalman Filter via AHRS class. And then the control class will calculate the required motor operations and pass the parameters down to the Overseer and the BLDC controllers to achieve remote controlled flying, based on the data from remote control and sensor. Finally, it records the various states from each parts of system for debug usage.

(N.B. This class is a wrapper of OptimusPiInterfaceClass to control motors in higher level)

Figure 8 OptimusPi_r0 Software Classes Hierarchy Specification with More Details

9.6. OptimusPi_r1 Software Classes Hierarchy Specification

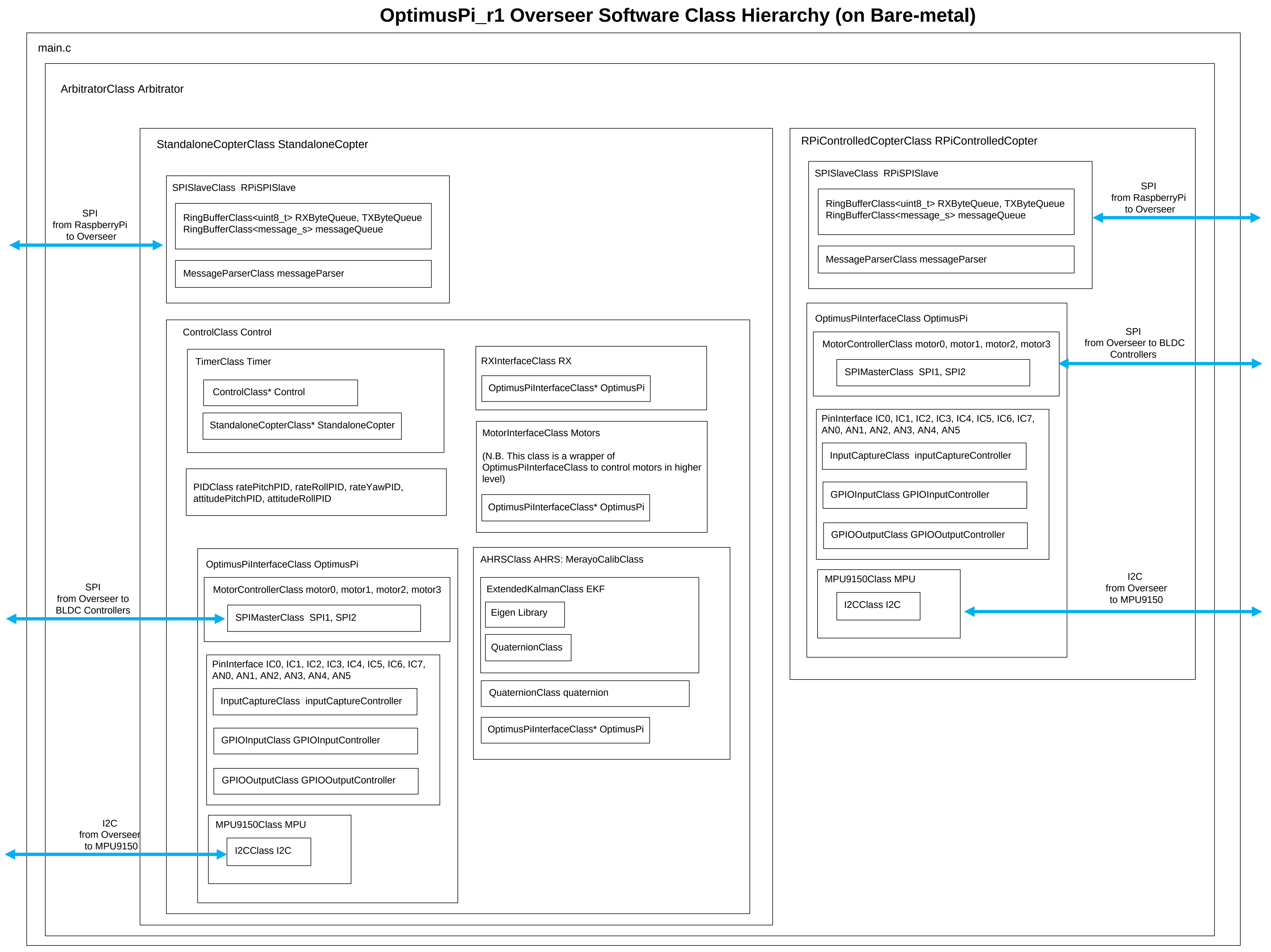

OptimusPi_r1 Overseer Software Class Hierarchy (on Bare-metal)
main.c
ArbitratorClass Arbitrator
StandaloneCopterClass StandaloneCopter
SPISlaveClass RPiSPISlave
RingBufferClass<uint8_t> RXByteQueue, TXByteQueue
RingBufferClass<message_s> messageQueue
MessageParserClass messageParser
SPI from RaspberryPi to Overseer
ControlClass Control
TimerClass Timer
ControlClass* Control
StandaloneCopterClass* StandaloneCopter
PIDClass ratePitchPID, rateRollPID, rateYawPID, attitudePitchPID, attitudeRollPID
RXInterfaceClass RX
OptimusPiInterfaceClass* OptimusPi
MotorInterfaceClass Motors
(N.B. This class is a wrapper of OptimusPiInterfaceClass to control motors in higher level)
OptimusPiInterfaceClass* OptimusPi
OptimusPiInterfaceClass OptimusPi
MotorControllerClass motor0, motor1, motor2, motor3
SPIMasterClass SPI1, SPI2
SPI from Overseer to BLDC Controllers
PinInterface IC0, IC1, IC2, IC3, IC4, IC5, IC6, IC7, AN0, AN1, AN2, AN3, AN4, AN5
InputCaptureClass inputCaptureController
GPIOInputClass GPIOInputController
GPIOOutputClass GPIOOutputController
MPU9150Class MPU
I2CClass I2C
I2C from Overseer to MPU9150
AHRSClass AHRS: MerayoCalibClass
ExtendedKalmanClass EKF
Eigen Library
QuaternionClass
QuaternionClass quaternion
OptimusPiInterfaceClass* OptimusPi
RPiControlledCopterClass RPiControlledCopter
SPISlaveClass RPiSPISlave
RingBufferClass<uint8_t> RXByteQueue, TXByteQueue
RingBufferClass<message_s> messageQueue
MessageParserClass messageParser
SPI from RaspberryPi to Overseer
OptimusPiInterfaceClass OptimusPi
MotorControllerClass motor0, motor1, motor2, motor3
SPIMasterClass SPI1, SPI2
SPI from Overseer to BLDC Controllers
PinInterface IC0, IC1, IC2, IC3, IC4, IC5, IC6, IC7, AN0, AN1, AN2, AN3, AN4, AN5
InputCaptureClass inputCaptureController
GPIOInputClass GPIOInputController
GPIOOutputClass GPIOOutputController
MPU9150Class MPU
I2CClass I2C
I2C from Overseer to MPU9150

Figure 9 OptimusPi_r1 Software Classes Hierarchy Specification

9.7. OptimusPi_r1 Software Classes Hierarchy Specification with Change Highlighted

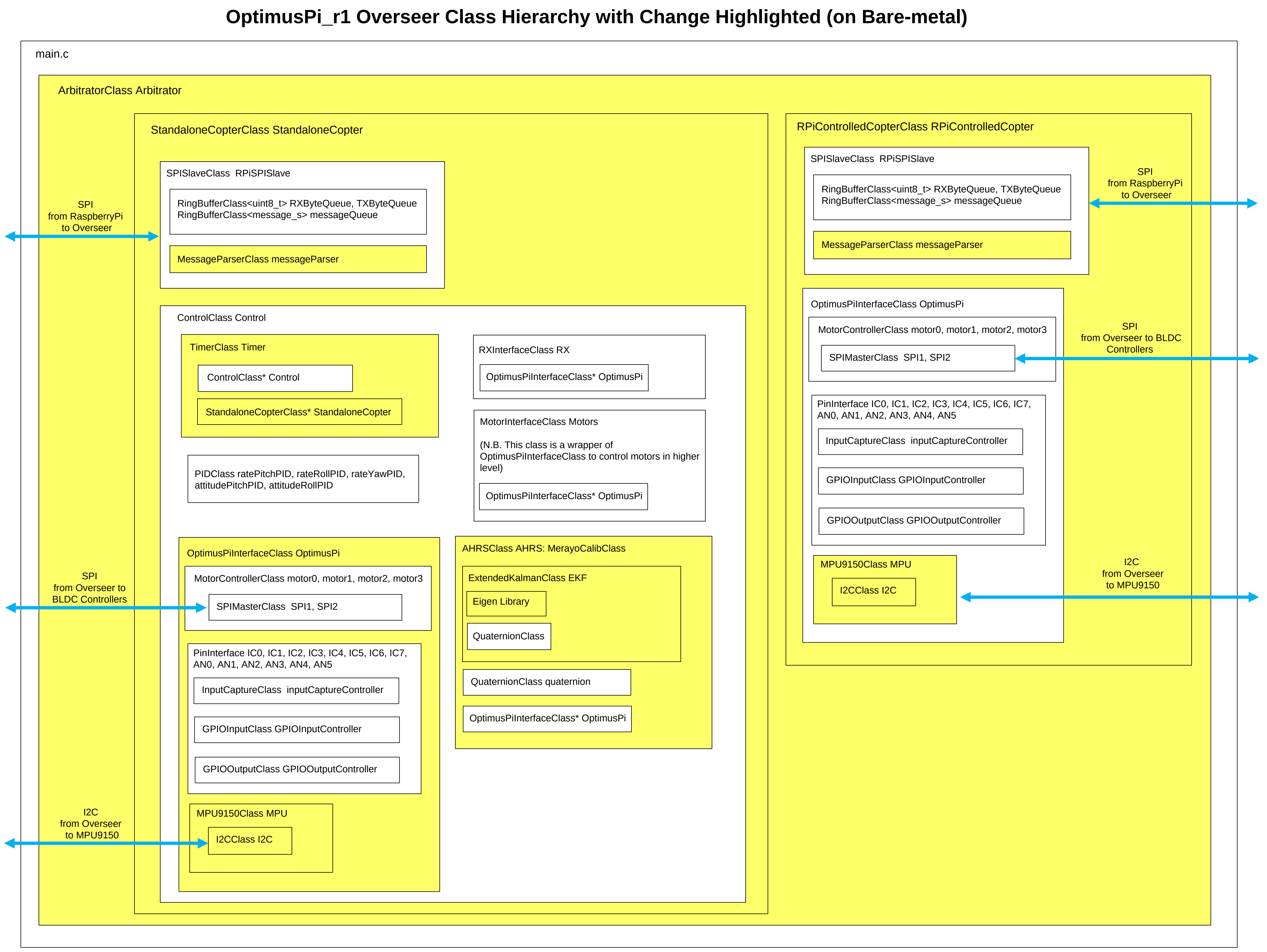
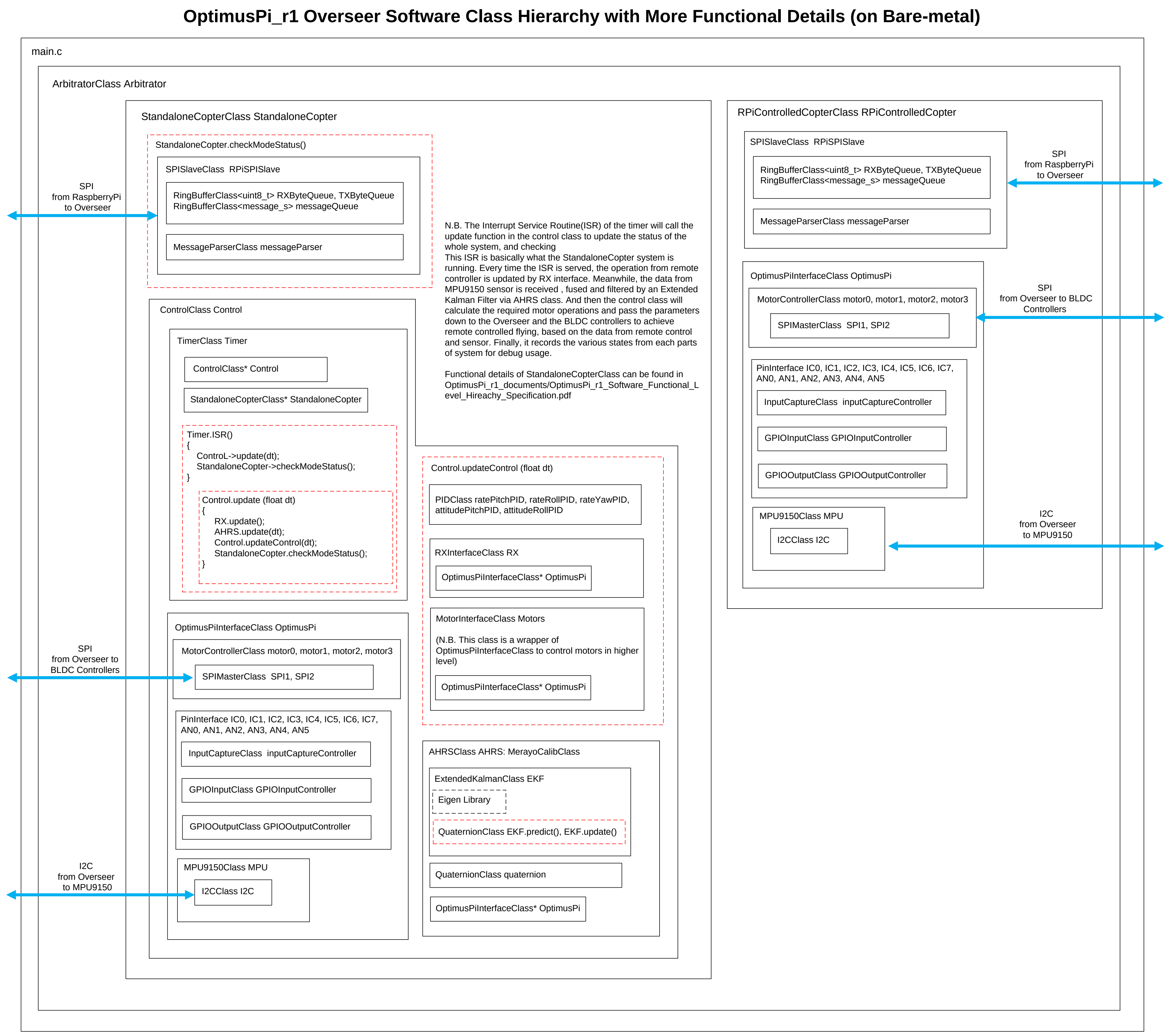

OptimusPi_r1 Overseer Class Hierarchy with Change Highlighted (on Bare-metal)
main.c
ArbitratorClass Arbitrator
StandaloneCopterClass StandaloneCopter
SPISlaveClass RPiSPISlave
RingBufferClass<uint8_t> RXByteQueue, TXByteQueue
RingBufferClass<message_s> messageQueue
MessageParserClass messageParser
SPI from RaspberryPi to Overseer
ControlClass Control
TimerClass Timer
ControlClass* Control
StandaloneCopterClass* StandaloneCopter
PIDClass ratePitchPID, rateRollPID, rateYawPID, attitudePitchPID, attitudeRollPID
RXInterfaceClass RX
OptimusPiInterfaceClass* OptimusPi
MotorInterfaceClass Motors
(N.B. This class is a wrapper of OptimusPiInterfaceClass to control motors in higher level)
OptimusPiInterfaceClass* OptimusPi
OptimusPiInterfaceClass OptimusPi
MotorControllerClass motor0, motor1, motor2, motor3
SPIMasterClass SPI1, SPI2
SPI from Overseer to BLDC Controllers
PinInterface IC0, IC1, IC2, IC3, IC4, IC5, IC6, IC7, AN0, AN1, AN2, AN3, AN4, AN5
InputCaptureClass inputCaptureController
GPIOInputClass GPIOInputController
GPIOOutputClass GPIOOutputController
MPU9150Class MPU
I2CClass I2C
I2C from Overseer to MPU9150
AHRSClass AHRS: MerayoCalibClass
ExtendedKalmanClass EKF
Eigen Library
QuaternionClass
QuaternionClass quaternion
OptimusPiInterfaceClass* OptimusPi
RPiControlledCopterClass RPiControlledCopter
SPISlaveClass RPiSPISlave
RingBufferClass<uint8_t> RXByteQueue, TXByteQueue
RingBufferClass<message_s> messageQueue
MessageParserClass messageParser
SPI from RaspberryPi to Overseer
OptimusPiInterfaceClass OptimusPi
MotorControllerClass motor0, motor1, motor2, motor3
SPIMasterClass SPI1, SPI2
SPI from Overseer to BLDC Controllers
PinInterface IC0, IC1, IC2, IC3, IC4, IC5, IC6, IC7, AN0, AN1, AN2, AN3, AN4, AN5
InputCaptureClass inputCaptureController
GPIOInputClass GPIOInputController
GPIOOutputClass GPIOOutputController
MPU9150Class MPU
I2CClass I2C
I2C from Overseer to MPU9150

Figure 10 OptimusPi_r1 Software Classes Hierarchy Specification with Change Highlighted

9.8. OptimusPi_r1 Software Classes Hierarchy Specification with More Functional Details

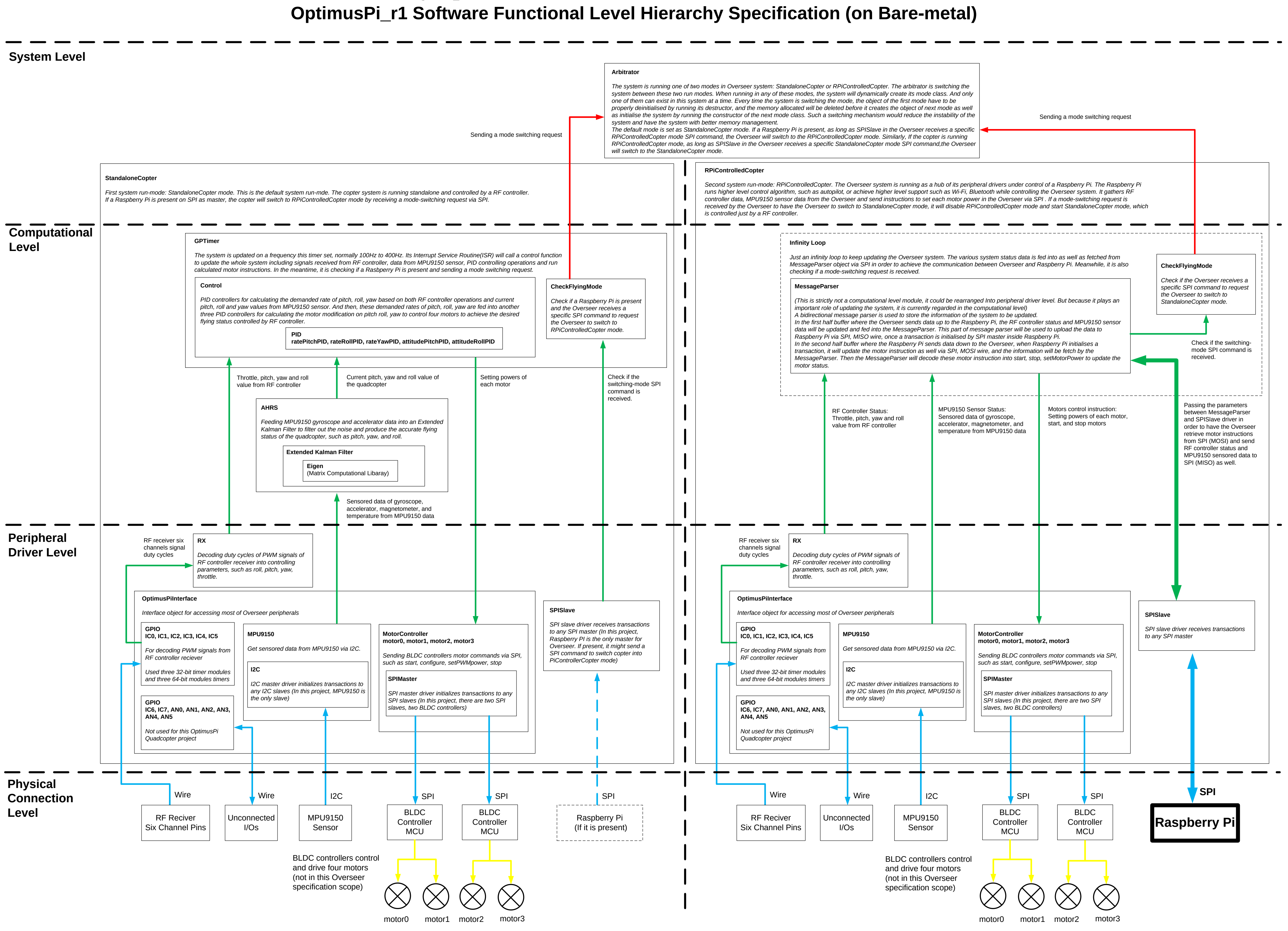

OptimusPi_r1 Overseer Software Class Hierarchy with More Functional Details (on Bare-metal)
main.c
ArbitratorClass Arbitrator
StandaloneCopterClass StandaloneCopter
StandaloneCopter.checkModeStatus()
SPISlaveClass RPiSPISlave
RingBufferClass<uint8_t> RXByteQueue, TXByteQueue
RingBufferClass<message_s> messageQueue
MessageParserClass messageParser
SPI
from RaspberryPi
to Overseer
ControlClass Control
TimerClass Timer
ControlClass* Control
StandaloneCopterClass* StandaloneCopter
Timer.ISR()
{
ControL->update(dt);
StandaloneCopter->checkModeStatus();
}
Control.update (float dt)
{
RX.update();
AHRS.update(dt);
Control.updateControl(dt);
StandaloneCopter.checkModeStatus();
}
Control.updateControl (float dt)
PIDClass ratePitchPID, rateRollPID, rateYawPID, attitudePitchPID, attitudeRollPID
RXInterfaceClass RX
OptimusPiInterfaceClass* OptimusPi
MotorInterfaceClass Motors
(N.B. This class is a wrapper of OptimusPiInterfaceClass to control motors in higher level)
OptimusPiInterfaceClass* OptimusPi
AHRSClass AHRS: MerayoCalibClass
ExtendedKalmanClass EKF
Eigen Library
QuaternionClass EKF.predict(), EKF.update()
QuaternionClass quaternion
OptimusPiInterfaceClass* OptimusPi
OptimusPiInterfaceClass OptimusPi
MotorControllerClass motor0, motor1, motor2, motor3
SPIMasterClass SPI1, SPI2
SPI
from Overseer to
BLDC Controllers
PinInterface IC0, IC1, IC2, IC3, IC4, IC5, IC6, IC7, AN0, AN1, AN2, AN3, AN4, AN5
InputCaptureClass inputCaptureController
GPIOInputClass GPIOInputController
GPIOOutputClass GPIOOutputController
MPU9150Class MPU
I2CClass I2C
I2C
from Overseer
to MPU9150
RPiControlledCopterClass RPiControlledCopter
SPISlaveClass RPiSPISlave
RingBufferClass<uint8_t> RXByteQueue, TXByteQueue
RingBufferClass<message_s> messageQueue
MessageParserClass messageParser
SPI
from RaspberryPi
to Overseer
OptimusPiInterfaceClass OptimusPi
MotorControllerClass motor0, motor1, motor2, motor3
SPIMasterClass SPI1, SPI2
SPI
from Overseer to BLDC
Controllers
PinInterface IC0, IC1, IC2, IC3, IC4, IC5, IC6, IC7, AN0, AN1, AN2, AN3, AN4, AN5
InputCaptureClass inputCaptureController
GPIOInputClass GPIOInputController
GPIOOutputClass GPIOOutputController
MPU9150Class MPU
I2CClass I2C
I2C
from Overseer
to MPU9150

N.B. The Interrupt Service Routine(ISR) of the timer will call the update function in the control class to update the status of the whole system, and checking
This ISR is basically what the StandaloneCopter system is running. Every time the ISR is served, the operation from remote controller is updated by RX interface. Meanwhile, the data from MPU9150 sensor is received , fused and filtered by an Extended Kalman Filter via AHRS class. And then the control class will calculate the required motor operations and pass the parameters down to the Overseer and the BLDC controllers to achieve remote controlled flying, based on the data from remote control and sensor. Finally, it records the various states from each parts of system for debug usage.

Functional details of StandaloneCopterClass can be found in OptimusPi_r1_documents/OptimusPi_r1_Software_Functional_Level_Hireachy_Specification.pdf

Figure 11 OptimusPi_r1 Software Classes Hierarchy Specification with More Functional Details

9.9. OptimusPi_r1 Software Functional Level Hierarchy Specification


OptimusPi_r1 Software Functional Level Hierarchy Specification (on Bare-metal)
System Level
Computational Level
Peripheral Driver Level
Physical Connection Level
Sending a mode switching request
Sending a mode switching request
Throttle, pitch, yaw and roll value from RF controller
Current pitch, yaw and roll value of the quadcopter
Setting powers of each motor
Check if the switching-mode SPI command is received.
Sensored data of gyroscope, accelerator, magnetometer, and temperature from MPU9150 data
RF receiver six channels signal duty cycles
RF Controller Status: Throttle, pitch, yaw and roll value from RF controller
MPU9150 Sensor Status: Sensored data of gyroscope, accelerator, magnetometer, and temperature from MPU9150 data
Motors control instruction: Setting powers of each motor, start, and stop motors
Check if the switching-mode SPI command is received.
Passing the parameters between MessageParser and SPISlave driver in order to have the Overseer retrieve motor instructions from SPI (MOSI) and send RF controller status and MPU9150 sensored data to SPI (MISO) as well.
RF receiver six channels signal duty cycles
Wire
Wire
I2C
SPI
SPI
SPI
Wire
Wire
I2C
SPI
SPI
SPI
RF Reciver Six Channel Pins
Unconnected I/Os
MPU9150 Sensor
BLDC Controller MCU
BLDC Controller MCU
Raspberry Pi (If it is present)
RF Reciver Six Channel Pins
Unconnected I/Os
MPU9150 Sensor
BLDC Controller MCU
BLDC Controller MCU
Raspberry Pi
BLDC controllers control and drive four motors (not in this Overseer specification scope)
BLDC controllers control and drive four motors (not in this Overseer specification scope)
motor0
motor1
motor2
motor3
motor0
motor1
motor2
motor3

Arbitrator

The system is running one of two modes in Overseer system: StandaloneCopter or RPiControlledCopter. The arbitrator is switching the system between these two run modes. When running in any of these modes, the system will dynamically create its mode class. And only one of them can exist in this system at a time. Every time the system is switching the mode, the object of the first mode have to be properly deinitialised by running its destructor, and the memory allocated will be deleted before it creates the object of next mode as well as initialise the system by running the constructor of the next mode class. Such a switching mechanism would reduce the instability of the system and have the system with better memory management.
The default mode is set as StandaloneCopter mode. If a Raspberry Pi is present, as long as SPISlave in the Overseer receives a specific RPiControlledCopter mode SPI command, the Overseer will switch to the RPiControlledCopter mode. Similarly, If the copter is running RPiControlledCopter mode, as long as SPISlave in the Overseer receives a specific StandaloneCopter mode SPI command,the Overseer will switch to the StandaloneCopter mode.

StandaloneCopter

First system run-mode: StandaloneCopter mode. This is the default system run-mde. The copter system is running standalone and controlled by a RF controller. If a Raspberry Pi is present on SPI as master, the copter will switch to RPiControlledCopter mode by receiving a mode-switching request via SPI.

GPTimer

The system is updated on a frequency this timer set, normally 100Hz to 400Hz. Its Interrupt Service Routine(ISR) will call a control function to update the whole system including signals received from RF controller, data from MPU9150 sensor, PID controlling operations and run calculated motor instructions. In the meantime, it is checking if a Rasbperry Pi is present and sending a mode switching request.

Control

PID controllers for calculating the demanded rate of pitch, roll, yaw based on both RF controller operations and current pitch, roll and yaw values from MPU9150 sensor. And then, these demanded rates of pitch, roll, yaw are fed into another three PID controllers for calculating the motor modification on pitch roll, yaw to control four motors to achieve the desired flying status controlled by RF controller.

PID
ratePitchPID, rateRollPID, rateYawPID, attitudePitchPID, attitudeRollPID

CheckFlyingMode

Check if a Raspberry Pi is present and the Overseer receives a specific SPI command to request the Overseer to switch to RPiControlledCopter mode.

AHRS

Feeding MPU9150 gyroscope and accelerator data into an Extended Kalman Filter to filter out the noise and produce the accurate flying status of the quadcopter, such as pitch, yaw, and roll.

Extended Kalman Filter

Eigen
(Matrix Computational Libaray)

RX

Decoding duty cycles of PWM signals of RF controller receiver into controlling parameters, such as roll, pitch, yaw, throttle.

OptimusPiInterface

Interface object for accessing most of Overseer peripherals

GPIO
IC0, IC1, IC2, IC3, IC4, IC5

For decoding PWM signals from RF controller reciever

Used three 32-bit timer modules and three 64-bit modules timers

GPIO
IC6, IC7, AN0, AN1, AN2, AN3, AN4, AN5

Not used for this OptimusPi Quadcopter project

MPU9150

Get sensored data from MPU9150 via I2C.

I2C

I2C master driver initializes transactions to any I2C slaves (In this project, MPU9150 is the only slave)

MotorController
motor0, motor1, motor2, motor3

Sending BLDC controllers motor commands via SPI, such as start, configure, setPWMpower, stop

SPIMaster

SPI master driver initializes transactions to any SPI slaves (In this project, there are two SPI slaves, two BLDC controllers)

SPISlave

SPI slave driver receives transactions to any SPI master (In this project, Raspberry PI is the only master for Overseer. If present, it might send a SPI command to switch copter into PiControllerCopter mode)

RPiControlledCopter

Second system run-mode: RPiControlledCopter. The Overseer system is running as a hub of its peripheral drivers under control of a Raspberry Pi. The Raspberry Pi runs higher level control algorithm, such as autopilot, or achieve higher level support such as Wi-Fi, Bluetooth while controlling the Overseer system. It gathers RF controller data, MPU9150 sensor data from the Overseer and send instructions to set each motor power in the Overseer via SPI . If a mode-switching request is received by the Overseer to have the Overseer to switch to StandaloneCopter mode, it will disable RPiControlledCopter mode and start StandaloneCopter mode, which is controlled just by a RF controller.

Infinity Loop

Just an infinity loop to keep updating the Overseer system. The various system status data is fed into as well as fetched from MessageParser object via SPI in order to achieve the communication between Overseer and Raspberry Pi. Meanwhile, it is also checking if a mode-switching request is received.

MessageParser

(This is strictly not a computational level module, it could be rearranged Into peripheral driver level. But because it plays an important role of updating the system, it is currently regarded in the computational level)
A bidirectional message parser is used to store the information of the system to be updated.
In the first half buffer where the Overseer sends data up to the Raspberry Pi, the RF controller status and MPU9150 sensor data will be updated and fed into the MessageParser. This part of message parser will be used to upload the data to Raspberry Pi via SPI, MISO wire, once a transaction is initialised by SPI master inside Raspberry Pi.
In the second half buffer where the Raspberry Pi sends data down to the Overseer, when Raspberry Pi initialises a transaction, it will update the motor instruction as well via SPI, MOSI wire, and the information will be fetch by the MessageParser. Then the MessageParser will decode these motor instruction into start, stop, setMotorPower to update the motor status.

CheckFlyingMode

Check if the Overseer receives a specific SPI command to request the Overseer to switch to StandaloneCopter mode.

RX

Decoding duty cycles of PWM signals of RF controller receiver into controlling parameters, such as roll, pitch, yaw, throttle.

OptimusPiInterface

Interface object for accessing most of Overseer peripherals

GPIO
IC0, IC1, IC2, IC3, IC4, IC5

For decoding PWM signals from RF controller reciever

Used three 32-bit timer modules and three 64-bit modules timers

GPIO
IC6, IC7, AN0, AN1, AN2, AN3, AN4, AN5

Not used for this OptimusPi Quadcopter project

MPU9150

Get sensored data from MPU9150 via I2C.

I2C

I2C master driver initializes transactions to any I2C slaves (In this project, MPU9150 is the only slave)

MotorController
motor0, motor1, motor2, motor3

Sending BLDC controllers motor commands via SPI, such as start, configure, setPWMpower, stop

SPIMaster

SPI master driver initializes transactions to any SPI slaves (In this project, there are two SPI slaves, two BLDC controllers)

SPISlave

SPI slave driver receives transactions to any SPI master

Figure 12 OptimusPi_r1 Software Functional Level Hierarchy Specification

9.10. Interim Report

Platform Crossing Development on Previous Quadcopter Project from Linux to Bare-metal System

Introduction

This project is basically about moving existing code from a Linux ARM platform to a bare-metal ARM platform while maintaining the same functionality. The previous quadcopter platform is implemented with a Raspberry Pi and three TI microcontrollers and this platform will achieve the same functionality with only three TI microncontrollers. The freed up Raspberry Pi can be used to achieve an unmanned flight or provide complicated wireless interface, such as Bluetooth, Wifi.

Summary of Previous Platform

There is a pervious implementation of a quadcopter platform by a previous MEng student Matthew Watson. The system hierarchy is shown in figure 1. This platform is implemented in C++ with a Raspberry Pi and three TI Tiva C Series microcontrollers, TM4C123AH6PM, where each of them contains an ARM M4 core. Two of them are used for controlling two four-phase motor channels each, for a total of four controlled motors. The third one acts as a parent to these two motor controllers as well as providing a GPIO interface, such as to a Remote Control RF Receiver, and communicating with the Raspberry Pi. Although this one is called overseer in the diagram, the control module actually is implemented in the Raspberry Pi and the overseer just passes the commands from the Raspberry Pi down the the motor controller.

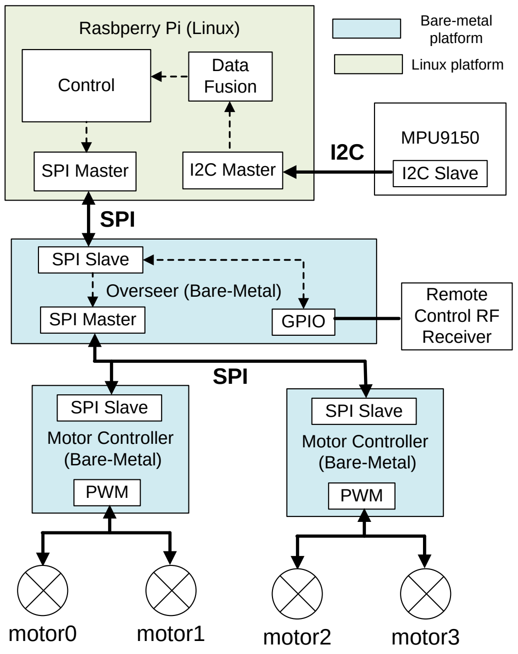


Figure 1: Previous Quadacopter Platform Software Hierarchy

The Raspberry Pi is on the top of the hierarchy. The main control code is implemented here since Linux operating system supports better debugability while using various libraries. The remote control operations is passed from the overseer up to the Raspberry Pi via a SPI bus by using a Linux SPI library. Then, the Raspberry Pi takes the data measured from MPU9150, an 9-axis sensor, via an I2C bus by using a Linux I2C library and sends back to the control module. Next, the control module processes the measurement data, fusing sensor data with a Kalman filters, and using tuned loop to calibrate the motor in order to achieve a stable flight. And it sends the calibration parameters from the Raspberry Pi down to the two motor controllers via SPI, where the overseer basically just bypass the most of the commands from Raspberry Pi down to the two motor controllers. The Raspberry pi currently is not a real supervisor to achieve unmanned fly but control the quadcopter to fly stably in order to achieve the commands from the remote control.

For the project, Matthew Watson left his previous code, a third-year thesis about implementing the control module, and a simple final documentation of this project.

Specification of Current Project

The current implementation moves all of previous Raspberry Pi code down to the overseer. The system hierarchy is shown in figure 2. The complete platform consists of three TI microcontrollers only, and the Raspberry Pi is for expansion usage such as unmanned control, wireless control as well as providing a debug interface to print out debug information on the terminal.

The overseer now implements the control module and data fusion module by removing the Linux decencies from previous Raspberry Pi code. The overseer currently takes instructions from the remote control via a GPIO interface, communicates with MPU9150 to get the sensor data via an I2C interface by using a Tivaware API, and then feed them back to the control module. If the control module is not configured as controlled by the Raspberry Pi, the control module will be set as controlled by the remote control and running as an isolated system. Then the control module will send the motor calibration parameters down to two motor controllers via an SPI interface.

Because the overseer runs on bare metal and it does not supports “printf-based” debug information but only supports breakpoints and memory/register read/write. For such a complicated system, it is necessary to forward debug information, such as STDOUT/STDERR, to certain place instead of regarding them as Linux dependencies. It is easy to implement it via an URAT to a host computer with a URAT RX/TX chip. But considering the Raspberry Pi is an essential expansion for future projects, it is done by forwarding STDOUT/STDERR into a separate SPI queue back to the Raspberry Pi. And when the Raspberry Pi is connected to the overseer, it will takes the data out of the queen and forward it to a terminal.

The Raspberry Pi is used as an expansion for future projects as well as debug terminal. When the Raspberry Pi connects to the overseer, it can configure the overseer as controlled by the Raspberry Pi and use the future development to control the quadcopter such as unmanned control, Bluetooth, and Wifi.

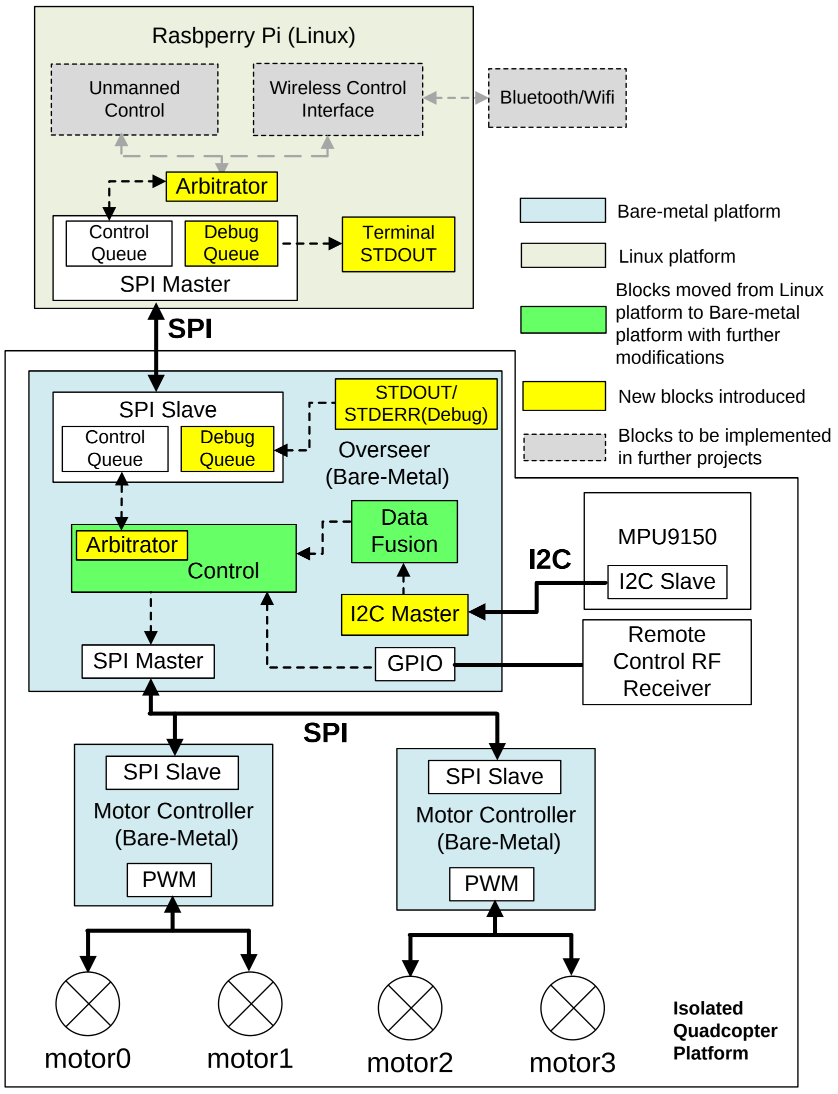


Figure 2: Current Quadacopter Platform Software Hierarchy

Progress

To the time this report written, the I2C master in the overseer has been implemented and fully tested. The debug print interface in the overseer up to the Raspberry Pi via SPI is being implemented. Once it is done, it is easy to see where the code goes wrong in the control module when the Linux dependencies are removed.

The main changllenge in this project is lacking of documentations. When implementing I2C master in the overseer, I2C master needs to follow the same bus protocol as the code in the Raspberry Pi defines. Since the previous student did not leave any documentations about where to find the Linux I2C library or any references used to implement the I2C interface. The progress was stuck for a while to seek for the right references in order to achieve the same functionality on the I2C interface. It turned out eventually the I2C protocol that MPU9150 supports is a derivation from the System Management Bus protocol, such as the timing sequence of its single read is StartBit + SlaveAddress + RegisterAddress+ StartBit + SlaveAddress + ReadData + StopBit rather than a most simple I2C protocol, i.e. StartBit + SlaveAddress + ReadData + StopBit. However, the I2C API for the overseer does not supports such a complicated timing sequence. Therefore, to these timing sequence was stitched by a few different sub timing I2C API, such as I2C_MASTER_CMD_BURST_SEND_START, I2C_MASTER_CMD_BURST_SEND_FINISH. Unfortunately, these sub timing I2C APIs are just writing to hardware control registers in the I2C hardware module. And the overseer chip specification only describes much fundamental hardware registers descriptions rather than the timing sequences it will follow. It took a while to make various assumptions to implement the I2C read/write interface. The assumptions on the timing sequence for each control command finally were verified from the oscilloscope and then the I2C interface for MPU9150 was then implemented.

Another changllege is actually about the JTAG programmer. Since Dr Dan Gladwin failed to order the right JTAG programmer, to the time this reported written, I have still been sharing a JTAG programmer with another student Matthew Fergusson who is doing a similar project. Because my project is mainly about verifying the code, it hinders my progress on this project.

The Gantt chart still kept similar to the initial report as the project kept following it.

Component	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Literature Review and get familiar with the software environment	x	x	x	x	x													
Board testing				x	X													
Design of system					x	x	x	x	x	x	x	x	x	x	x	x		
Software Implementation								x	x	x	x	x	x	x	x	x		
Testing of system								x	x	x	x	x	x	x	x	X	X	X
Documentation Writing					x	x	x	x	x	x	x	x	x	x	x	x	x	x