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.

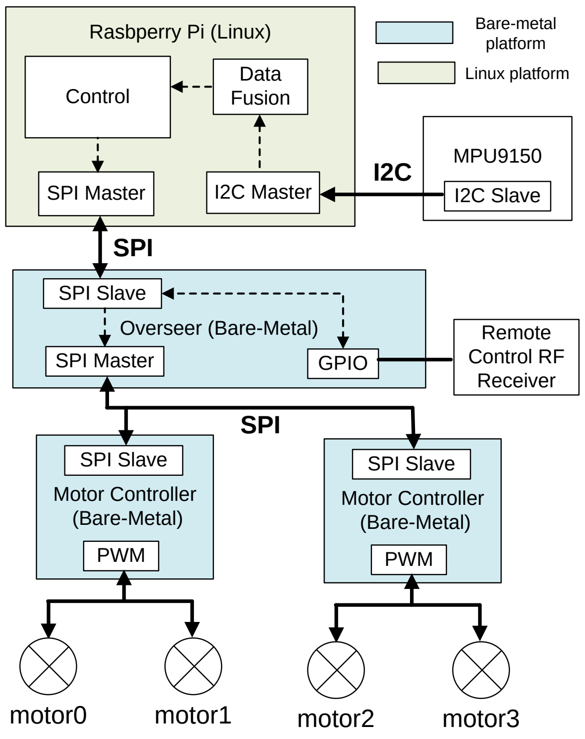

Figure 1: Previous Quadacopter Platform Software Hierarchy

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.

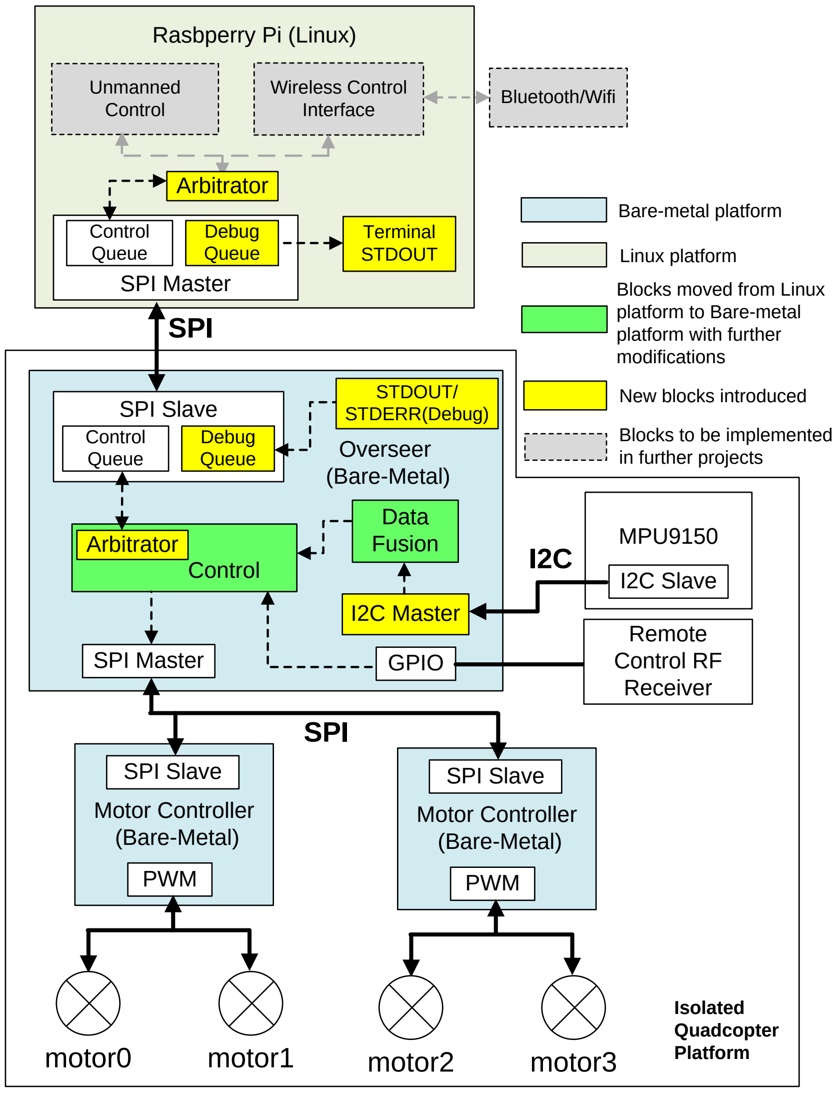

Figure 2: Current Quadacopter Platform Software Hierarchy

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