OptimusPi_r1 Software Functional Level Hierarchy Specification (on Bare-metal)

System Level

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.

Sending a mode switching request

StandaloneCopter

First system run-mode: StandaloneCopter mode. This is the default system run-mode. 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.

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.

Computational Level

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 Raspberry 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.

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.

Check if the switching-mode SPI command is received.

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.

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

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.

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)

Sensored data of gyroscope, accelerator, magnetometer, and temperature from MPU9150 data

Peripheral Driver Level

RF receiver six channels signal duty cycles

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 PiControlledCopter mode)

SPISlave

SPI slave driver receives transactions to any SPI master

Physical Connection Level

Wire — **RF Reciver Six Channel Pins**

Wire — **Unconnected I/Os**

I2C — **MPU9150 Sensor**

SPI — **BLDC Controller MCU**

SPI — **BLDC Controller MCU**

SPI — **Raspberry Pi (If it is present)**

BLDC controllers control and drive four motors (not in this Overseer specification scope)

motor0 motor1 motor2 motor3

Wire — **RF Reciver Six Channel Pins**

Wire — **Unconnected I/Os**

I2C — **MPU9150 Sensor**

SPI — **BLDC Controller MCU**

SPI — **BLDC Controller MCU**

SPI — **Raspberry Pi**

BLDC controllers control and drive four motors (not in this Overseer specification scope)

motor0 motor1 motor2 motor3