OptimusPi Project

Hardware

PCB CAD Files

The PCB was designed in EAGLE, and as such is defined in the OptimusPi.sch and OptimusPi.brd files. A professional license is required to modify these designs due to their complexity, but this is easily obtained through various means.

Board Structure

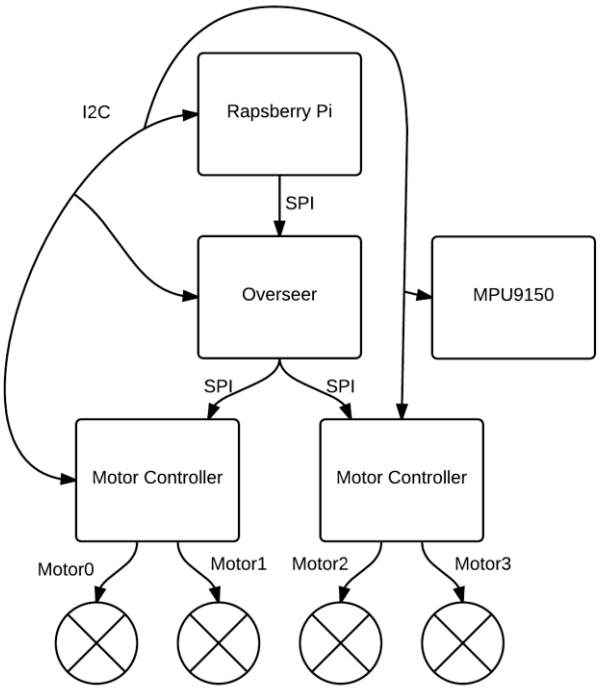
The board is comprised of six layers, with the top layer containing the SMPS, microprocessors, and MOSFET drivers. Layer 2 is used as a supply plane, split into three sections; 12V, 5V, and 3.3V. The 3.3V section runs under the microprocessors and inertial sensor, the 5v runs under the Raspberry Pi interface header and the GPIO headers, and the 12V runs under the remainder of the board. Layers 3 and 4 are used as signal layers, with one containing horizontal traces and the other vertical. Layer 5 is a ground plane, with a break running under the boundary between the 12V supply plane and the 3.3V and 5V planes, separating the digital ground return currents from the power electronic ground return currents. Layer 6, the bottom side of the board, is mostly just used for the inverter MOSFETs.

Components

All required components are listed in the bill of materials (BoM).

Organisation

The board is designed around three MCUs. Two are used to control two four phase motor channels each, for a total of four controlled motors, while the third acts as a controller parent to the two motor controllers, as well as providing a GPIO interface and communicating with the Raspberry Pi.



Three SPI interfaces exist; one between the RPi and overseer, with the RPi as the master, and two between the overseer and motor controllers, with the overseer as the master. A global I2C interface then links all three MCUs, the RPi, and the MPU9150 inertial sensor.

Software

The software is split across three projects; two Code Composer Studio projects containing the overseer and motor controller firmware, and one eclipse project for the OptimusPi Linux static library. Also included is a custom version of the Tivaware driver library, as a number of bugs exist in the current version that had to be fixed.

Motor Controller Firmware

The two motor controller MCUs are programmed with identical firmware. This firmware consists of an overall motor controller object for each of the two motor channels, which can instantiate specific types of motor controller when requested (although currently the only developed controller is the BLDC controller). Communication with the overseer MCU is achieved via an instance of the SPI slave class, which is also shared with the overseer for RPi->overseer communication.

The main loop of the overseer simply consists of checking the SPI message queue and keeping the SPI emulated registers updated.

Overseer Firmware

The overseer firmware is designed in a very similar manner to the motor controller firmware, with the main loop only consisting of populating emulated read registers and checking the SPI slave message queue. Functionality is then provided by four motor controller interfaces, which handle the

passing of messages to the child motor controllers, and a number of pin interface classes, one per GPIO channel. These function in a similar manner to the motor controller classes in the motor controller firmware, instantiating different classes of pin controller depending on how the pin is configured. This allows any pin to be configured as a GPIO input or output, as well as an input capture pin or analogue-in pin depending on the exact pin type.

MCU Firmware Organisation

All peripherals in both sets of firmware are configured by abstracting the device register addresses and accessors, meaning all accesses to the peripheral control registers are achieved using the same line of code regardless of which peripheral is being used (i.e. ADC0 or ADC1). These abstractor variables are populated by calling either the initAsADC0() or initAsADC1() functions, for example. This approach means that all functions relating to a class of peripheral only need to be defined once, as opposed to having separate classes for each instance of the same peripheral type.

SPI Message System

The SPI interface between the RPi and the overseer functions identically to the interface between the overseer and motor controllers. Transactions are based on messages comprising of an 8 bit command followed by a number of 32 bit parameters. Message commands are defined in SPICommands.h, which is shared across both sets of MCU firmware as well as the Linux library.

Transmission of a write command involves sending the 8 bit message command followed by 4*numberOfParameters bytes. At the slave end this is received by first examining the command byte, using the getNumberOfParams function of the messageParserClass class to determine how many parameter bytes are to be expected. This function works by accessing a lookup table stored in the messageParserClass class containing the number of parameters and read/write direction for every command, along with a 32bit read buffer. This read buffer is what is used as a response to a read command, and thus must be kept updated with the latest available values.

A read command also starts by sending a command byte, after which the master constantly sends zeros until an acknowledge byte (0x55) is received from the slave. All subsequent bytes are then the requested read parameters. The slave recognises the command as a read command using the messageParser's getDirection() function, and then grabs the pre-stored response using the getReadResponse() function before chopping it up into bytes and pushing it onto the TX FIFO.

JTAG Programming

During this development programming is achieved using a Blackhawk XDS100v2 JTAG programmer. On the old PCB version the cable on the JTAG connector must lie over the nearby capacitor for the overseer JTAG connector (the connector nearest to the board edge). Moving along the longest axis of the board the next connector is MCU controlling motor channels 0 and 1, and must be connected with the JTAG cable pointing the opposite direction to the overseer JTAG connection. The final connector programs the controller for motors 2 and 3, and must be connected with the cable lying over the inductor.

On the new PCB this has been modified so that all connections are in the same orientation, with the JTAG cable lying over the side of the board with the connections for motors 1 2 and 3.

Once connected, programming is achieved using the built-in CCS debugger. This should already be configured for the two included projects, or can be configured by creating a new CCS project from file>new>project>c/c++>ccs project, selecting TM4C123AH6PM as the device, and Texas Instruments XDS100v2 emulator and the connection. This will create a project with the required startup code and main.cpp if selected.

OptimusPi Linux Library

The OptimusPi library provides an easy method of accessing both the Overseer MCU and the MPU9150 sensor IC. In order to use this library in a project, include the Overseer/OptimusPiInterface.h header and add the libOptimusPi.a to the linker include option, before instantiating a single OptimusPiInterfaceClass object. The innards of the board are then accessed via the member objects of the OptimusPiInterfaceClass class, for example OptimusPiInteface.motor0.start(), OptimusPiInterface.IC0.config(*InputCapture*) etc.

The MPU9150 interface class is used by including the MPU9150/MPU9150.h header and the same linker archive as above. Instantiating an MPU9150Class object creates the MPU9150 inteface object along with the underlying I2C interface. An example of how to configured the MPU is included in the OptimusPiCopter project. Note, the compiler and linker include paths may need to be updated when moving the source folder around as some references may be absolute. An error in these will manifest as an unresolved include for a compiler include path error, or an unresolved symbol error for a linker include error.