**System Definition Document**

Peter R Hague

|  |  |  |
| --- | --- | --- |
| Date | Updated Reference Number | change |
| 11/08/2009 | PLM-OBDH-SDD-205-3 | Clarified text based on first proof reading |
| 06/08/2009 | PLM-OBDH-SDD-205-2 | Changed to new format |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**Table of Contents**

Introduction 3

Physical specification 3

Software modules 5

Core Modules 5

Low level interfaces 5

Subsystems 6

Outline of operation 8

Modes 8

Watchdog timer 9

Mode Diagram 9

Commands 10

List of Commands 11

Main Programme Loop 13

System Diagram 14

Logging 15

System Diagram 16

Recovery/Start-up 17

System Diagram 18

I2C Interface 19

System Diagram 19

Modem Interface 20

System Diagram 21

SD Card Access 22

System Diagram 23

ADC Interface 24

System Diagram 24

Timing 25

System Diagram 26

ADCS 27

System Diagram 28

COMS 29

System Diagram 30

Camera 31

System Diagram 32

Payload 33

System Diagram 33

PSU 34

System Diagram 34

Memory configuration 35

Error correction 37

# Introduction

The On Board Data Handling (OBDH) is the embedded computer system on board PLUME, that controls the actions of all the other subsystems, stores their output, and transfers it (utilising the COMS subsystem) back to the ground station on Earth.

The OBDH subsystem consists of a flight controller, an SD card for secondary storage, and the software required to run the flight controller.

As the hardware has already been provided, the focus of the subsystems development is on software and interfacing. This document therefore only briefly covers the physical nature of the subsystem, as there is already extensive documentation for it available.[[1]](#endnote-1)

It is also worth noting that the board we are using has flown a previous cubesat mission successfully, and therefore can be considered a proven component.[[2]](#endnote-2)

The software must fit in the 50Kb flash ROM of the flight controller, as there are no plans for loading software modules from the secondary storage (the SD card).

## Physical specification

In order to speed development, we have used an off-the-shelf flight controller. A full description of its physical specifications can be found at these locations:

<http://cubesatkit.com/docs/datasheet/DS_CSK_FM430_710-00252-C.pdf>

<http://www.cubesatkit.com/docs/cubesatkitsystemchart.pdf>

The following is a brief overview based on this information:

|  |  |
| --- | --- |
| Board Model | CubesatKit FM430 Flight Module |
| Mass | 74g |
| Dimensions | 96mm x 90mm x 14.9mm |
| MCU | MSP430 F1612 16-bit RISC CPU running at 8MHz / 8MIPS with 55Kb Flash and 5Kb RAM |
| Power | No more than 20mA at 3.3V (66mW max power consumption) – low power modes can run the CPU as low as 2μA |
| Logic | 3.3V digital |
| Operating Temperature | -40°C to 85°C |
| Extra Devices | USB 2.0 port for interfacing pre-launch  SD card (memory 2Gb, mass 2g, operating voltage -2.7V to 3.6V) |

The specifics of the electronic interfaces will be detailed in the electronic interface document.

# Software modules

In order to speed development and testing, the software is divided in to self-contained components, each with specified inputs and outputs. This section provides a brief description of each module, and a longer description of each can be found under the heading of the same name.

There are thirteen modules:

## Core Modules

**Main Programme Loop**

This component is automatically run when the satellite powers up, and keeps running throughout its life. Referred to as MPL throughout the documentation. It runs in different modes depending on the status of the satellite.

**Logging**

This component writes data to the log data to the board.

**Recovery/Startup**

These are a series of procedures invoked by the MPL at the start of the mission and after any power loss.

## Low level interfaces

**I2C Interface**

Uses the I2C standard, implemented by the board, to interface with the power supply and camera

**Modem Interface**

Interface used to communicate with the satellites modem

**SD Card Access**

Procedures to read and write to the SD card. This uses raw data, foregoing the functionality of a proper filing system for the sake of simplicity.

**ADC Interface**

The 12-bit Analogue-to-Digital Converter interface. Used to gather data from two analogue sources; an internal temperature sensor mounted on the flight controller, and the ADCS subsystem.

**Timing**

Provides timing services for electronic interfaces that need to function over specific periods of time (this cannot be done by clock cycles as the speed of the MSP430 is variable dependent on power conditions). Also provides a long period timer to give an estimate of orbital position.

## Subsystems

**ADCS**

Interfaces with the ADCS subsystem, sends it commands to select the three channels, and reads the data from each channel in turn.

**COMS**

Handles communication, also implements commands sent from the ground station to the satellite.

**Camera**

Interfaces with the camera, checks the suitability of images (i.e. that they are facing a well let Earth not blackness) and compresses them on the SD card ready for transmission to the ground station.

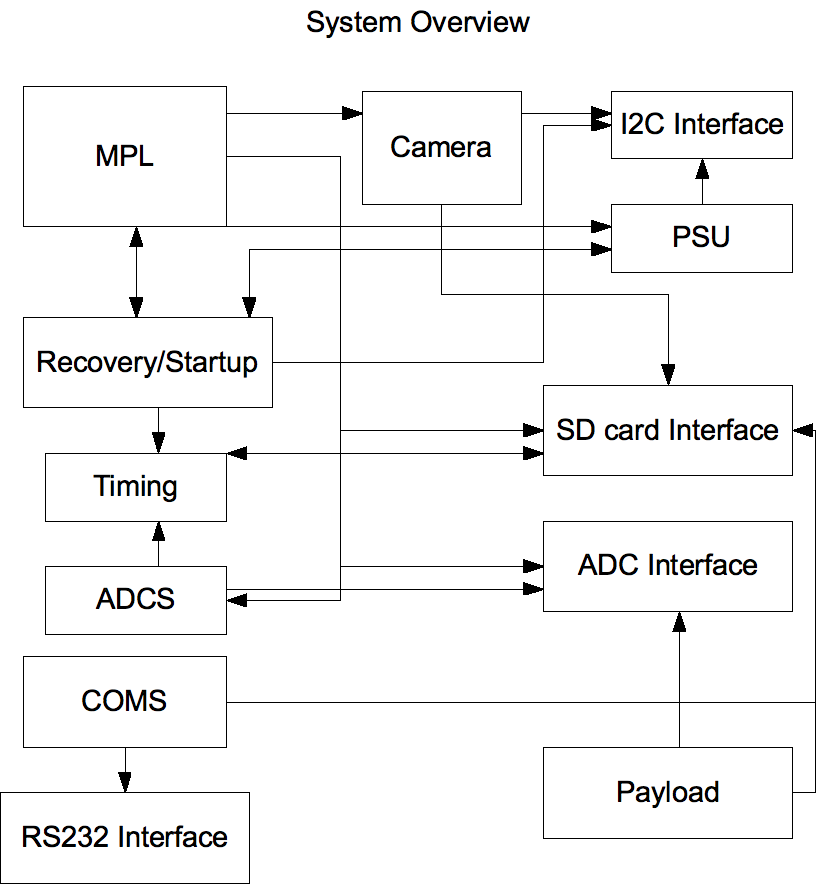
**Payload**

An interrupt-driven component that registers a payload event, gathers the corresponding data from the payload, then stores this on the SD card.

**PSU**

Interfaces with the power supply subsystem, and feeds information about current power levels to the main programme loop and recovery components.

The diagram overleaf indicates how the components connect together. Note that although components may share names with subsystems, each box is a software component not a different subsystem. Each arrow indicates a component calling another one. Each box represents a function, and box with dashed outlines are interrupts (although the particular diagram below does not currently feature any). In the discussion of each individual component, the specifics of these calls will be shown.



# Outline of operation

## Modes

The satellite operates in one of three modes at all times, each one having a distinct power requirement:

1. Recovery
2. Idle
3. Operation
4. Autonomous

The entry point to the code leads into Recovery, which powers on the COMS systems of the satellite, and then begins transmitting a beacon so that the ground station can locate the satellite. After waiting 30 minutes, the antenna will be deployed. This is required on first start up, and remains in Recovery as there is no way to sense the deployment status of the antenna. Intervals (the size of which has yet to be determined) are left in the transmission to allow a first contact command to be received. If no command is forthcoming in a predetermined amount of time (in the order of days or weeks, the exact value is yet to be decided) it is assumed the satellite is unable to respond to ground station commands, and the satellite enters Autonomous mode.

Recovery mode is entered on the following criteria; separation switch release, watchdog timer time out, long period of no ground contact when in idle or operation mode, power loss, and brownout.

Idle mode is just that; the satellite is passive and waits for further instruction from the ground station. Unlike Recovery, no signal is transmitted and the satellite will not enter Autonomous mode. It is entered after the ground station sends a command to the satellite while it is in Recovery mode. It is exited when a second command is received, and this is the entry point for Operation mode.

Operation mode is the normal mode of the satellite – one that runs some or all of the satellites subsystems based on an action list (which can be updated from the ground station during the mission, see the commands section for more details). In each loop, each item on the action list is attempted if there is enough power available. Below is a data structure containing which actions must be performed each loop through the Operation mode.

|  |  |
| --- | --- |
| **Action** | **Description** |
| X Payload normal operation | Operates the 'X' dust detector in normal mode, with safety functions |
| Y Payload normal operation | Operates the 'Y' dust detector in normal mode, with safety functions |
| X Payload unsafe operation | Operates the 'X' dust detector with NO safety functions, only to be used if it will not work otherwise |
| Y Payload unsafe operation | Operates the 'Y' dust detector with NO safety functions, only to be used if it will not work otherwise |
| ADCS operation | Collects ADCS data |
| PSU temperature check | Uses the OBDH to perform a backup check on the PSU temperature |
| Camera (smart) | Takes a photograph the next time ADIL spots a good photo |
| Camera (dumb) | Takes a photograph as soon as the action is set, and then clears action |
| SD card | Stores data to the SD card rather than RAM |

For each action, there is a boolean status (either the action will be performed or it will not) and a delay (which, if nonzero, counts down each loop and when it reaches one, flips the status and zeros the delay). As some of the actions (the unsafe payload operations) are potentially dangerous, each action status will be represented by a byte which is all zeros for no operation, and all ones for operation, thus minimising the possibility of random bit flips altering the table in a coherent way.

Autonomous mode is a special mode designed for the contingency in which the satellite cannot receive ground station commands. It runs a specialised version of the operation mode that attempts to complete the mission without human intervention. It transmits log data continuously, leaving gaps in the transmission in the same way Recovery mode does, and then after a delay (yet to be determined) power up the ADCS system, and then after another delay power up the primary payload. The camera is not used in this mode. If communication is restored, a command can be sent by the ground station to switch from Autonomous mode to Idle.

## Watchdog timer

The board is equipped with a watchdog timer. This will require the code to constantly report normal functionality at the beginning of each non-trivial subroutine (that is, each one that contains branching/decision making), and if this does not occur at a predetermined interval at least, then an interrupt will be generated that will place the satellite in recovery mode. The purpose of this is to stop any unforeseen bugs locking the OBDH subsystem.

## Mode Diagram

# Commands

The following is a list of commands that will be recognised by the satellite. When each command is received by the satellite, an acknowledgement is transmitted back to the ground station in addition to any data. If a command is sent that does not apply to the mode the satellite is currently in, a negative acknowledgement will be sent instead.

Each command (excluding data) is 4 bytes in length. This is an extremely large command space for only a small number of commands, but it will prevent third parties from being able to easily take control of our satellite.

## List of Commands

These are the commands that we intend to implement. The size in bytes includes the command code (4 bytes, always) and the parameters that accompany it (variable in size).

|  |  |  |  |
| --- | --- | --- | --- |
| **Command** | **Size (bytes)** | **Applicable Modes** | **Description** |
| Idle | 4 | Recovery, Autonomous, Operation | Puts the satellite into Idle mode |
| Reset | 4 | Autonomous, Idle, Operation | Puts the satellite into Recovery mode |
| Wake | 4 | Idle | Puts the satellite into Operation mode |
| Status | 4 | All | After acknowledgement, transmits the current mode of the satellite |
| Action | 6 | Operation | Data in command is written to status component of action table for a specified action. |
| Get Table | 4 | Operation | After acknowledgement, returns all status variables from the action table. |
| Delay | 6 | Operation | Data in command is applied to status component of action table for a specified action. |
| Download | 7 | Operation, Idle | A 512 byte block of memory (specified in command) is transmitted after acknowledgement |
| Current log | 5 | Operation, Idle | After acknowledgement, returns the current address in the log area being written to |
| Current photo | 5 | Operation, Idle | After acknowledgement, returns the current address in the photo area being written to |
| Payload Command | 6 | Operation | Sends a specific 9-bit digital signal to the payload (see payload software interface for details) |
| Payload Status | 5 | Operation | Returns, after acknowledgement, a specified analogue channel from the payload (see payload software interface for details) |
| Upload | 1031 | Idle, Operation | Uploads a 512 byte block of data (doubled up in case of error) to be stored in a specified address on the SD card. Returns negative acknowledgement if bits do not match up. |
| Reflash | 10 | Idle, Operation | Reflashes the boards memory with the data between the two specified address, and then restarts to execute the new program. |

# Main Programme Loop

The main programme loop implements the four operational modes of the satellite. It contains the entry point of the code, and also the code to interpret ground station commands supplied from the COMS subsystem.

Commands come in a bit at a time, each on generating an interrupt, and are built into complete commands in a buffer that is global in this subsystem. When the main loop detects the command is ready, it branches based on its content.

The MPL consists of the following functions:

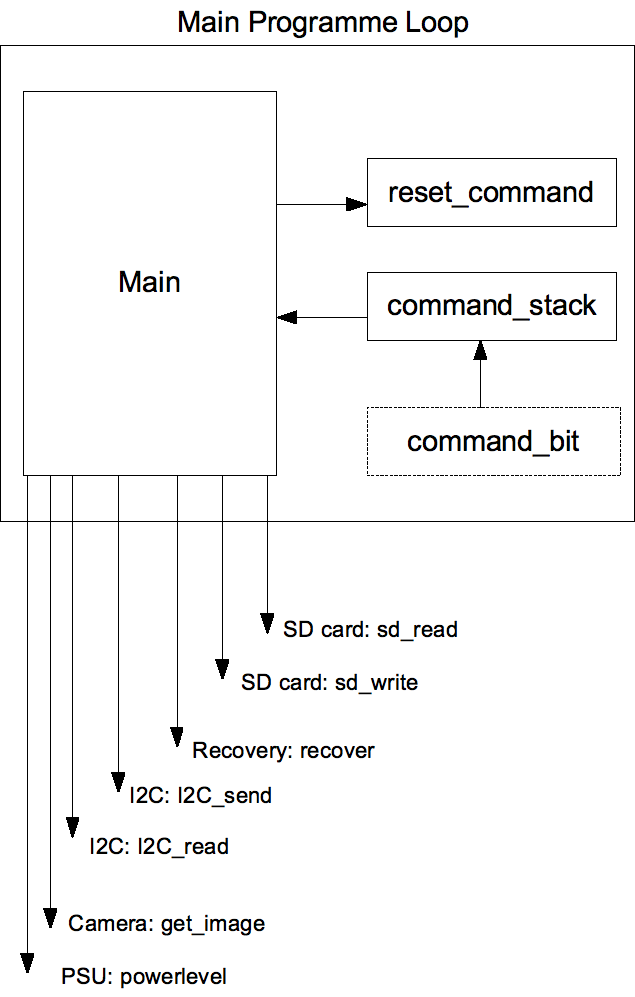
void reset\_command();

int command\_stack(char bit);

interrupt command\_bit();

int main();

## System Diagram



# Logging

The logging component sequentially writes a data structure to the SD card, via a buffer, reflecting the status of the satellite.

This buffer is 512 bytes in size, and is held in the boards RAM. When log\_entry is invoked, it checks if there is enough room in the buffer to store an entry, and if there is not the buffer is written to the SD card and the log entry is written to a new buffer.

When a payload event occurs, an event is logged regardless of if an entry is being written to the buffer at the time. For this reason, all pointers are updated before a writing event occurs, which should prevent log data being overwritten.

The data packets stored for each log entry are as follows:

|  |  |
| --- | --- |
| Byte | Contents |
| 1-2 | Time Index |
| 3 | Payload event indicator |
| 4-5 | Payload data |
| 6 | Payload error code |
| 7-8 | Internal temperature reading |
| 9-14 | ADCS reading (X, Y, Z) |
| 15-26 | Power readings from 6 solar panels, 2 bytes each |
| 27-28 | PSU temperature reading |
| 29-30 | Camera brightness reading (from ADIL) |

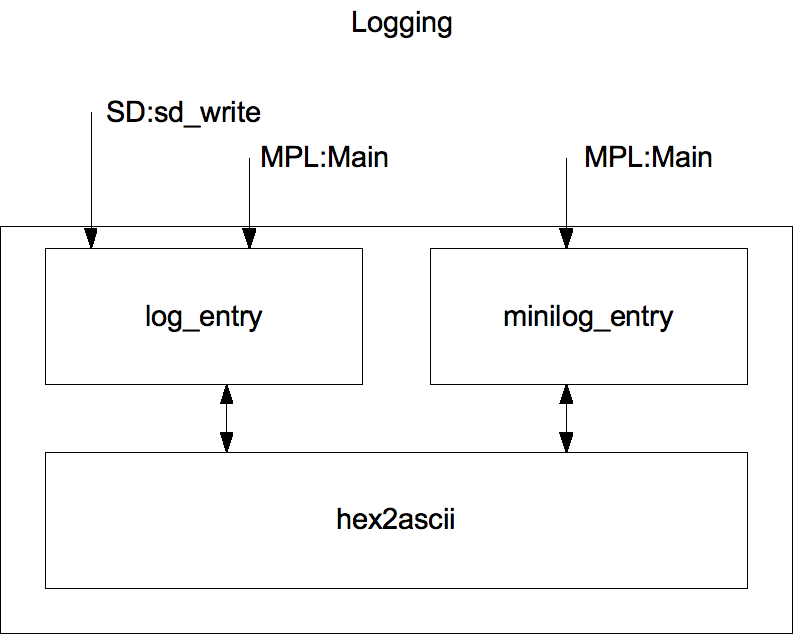
There is also an alternative logging routine (minilog) to be used if the SD card is unavailable. This stores a smaller data packet in memory instead of on the SD card. The logging subroutine consists of the following functions:

void log\_entry();

void minilog\_entry(unsigned char \*minilog);

void hex2ascii();

## System Diagram



# Recovery/Start-up

Recovery and start-up (R&S) are covered by a single module; from the perspective of the flight controller the two situations are largely the same. This module will be invoked at the beginning of the Recovery mode (implemented in MPL) One component, the payload initialisation routine, will not be executed automatically. Because of the delicate nature of the payload the flight controller will wait for a signal before attempting to power it up.

The R&S module consists of the following functions:

void recover();

int powerup\_adc();

int powerup\_sd();

int powerup\_payload();

int powerup\_camera();

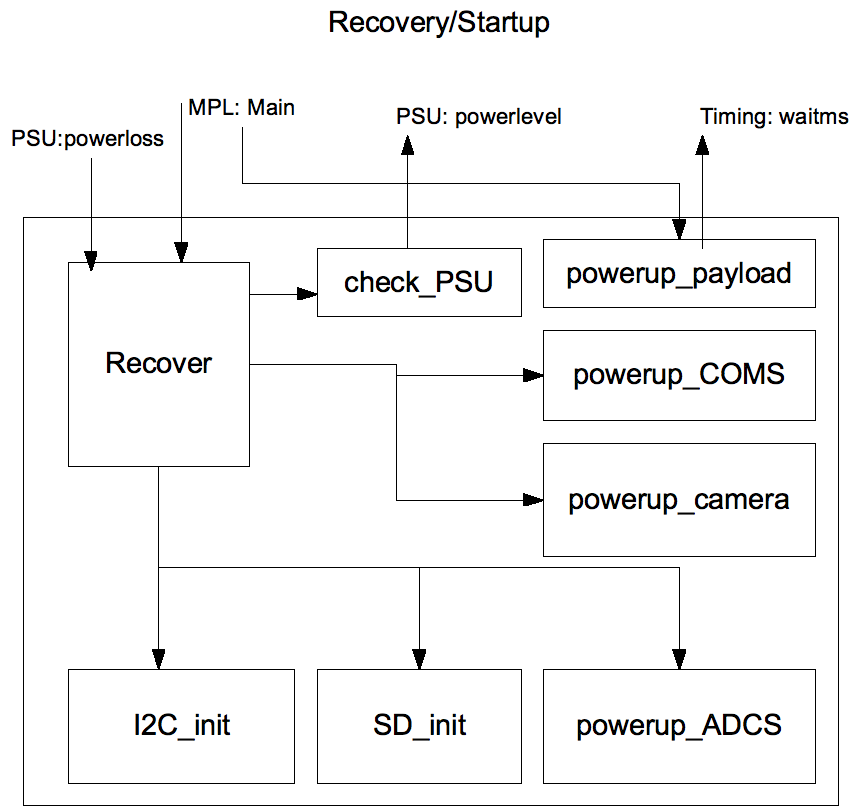
int powerup\_coms();

int i2c\_init();

int check\_psu();

int SD\_init();

## System Diagram



# I2C Interface

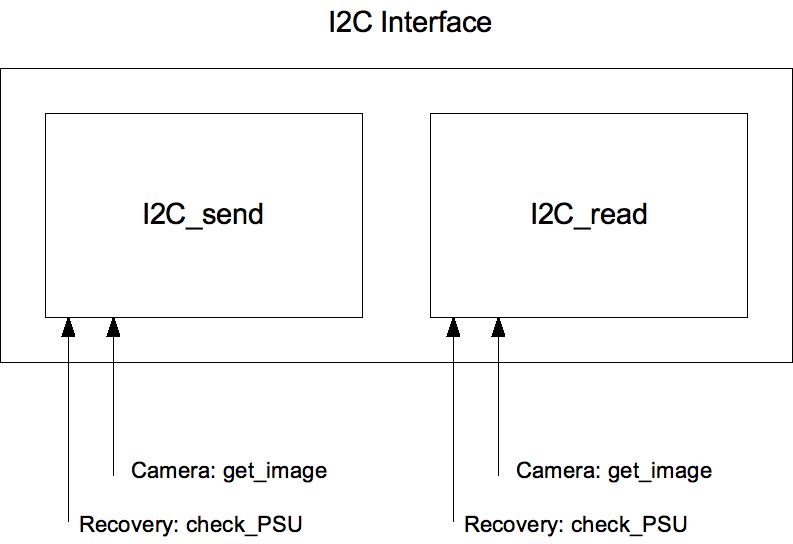
An I2C interface is used to communicate between the flight controller and the PSU, and also between the flight controller and the camera. In each case, the flight controller is set up as the master and the subsystem as the slave.

The I2C module consists of the following functions:

void I2C\_send(int address, char data);

char I2C\_read(int address);

## System Diagram



# Modem Interface

This component will contain the code for dealing with the modem at the lowest level, providing services for the COMS component. There will also be an interrupt that is to be triggered when a signal is received from the ground station. This interrupt will keep track of received data and trigger the COMS component at the appropriate point.

The modem is purely for use by the COMS component, however to make development more modular we have kept it separate. The modem component will have the following functions:

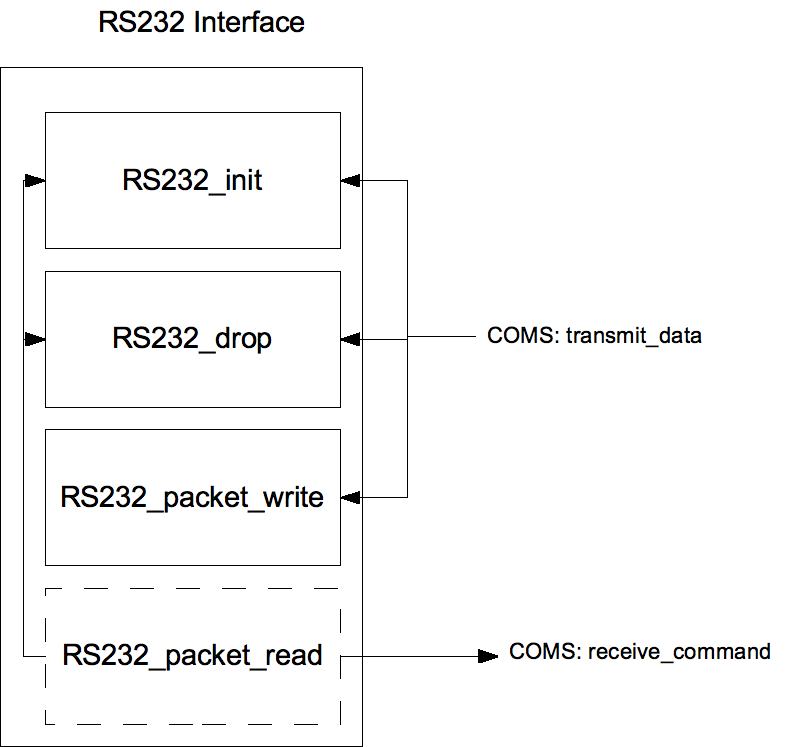
void RS232\_init();

void RS232\_drop();

void RS232\_packet\_write(char \*source);

interrupt void RS232\_packet\_read();

## System Diagram



# SD Card Access

The SD card access module allows reading and writing of raw data to the SD card. No formal file-system is included in order to simplify the module. All data is written to predefined points on the card.

In order to ensure that data is not overwritten in the event of a loss of power and a reset of the flight controller, some data pointers will be stored at the beginning of the SD card (with redundant copies in case a problem or power loss occurs during writing to one of them). It is important to note this is not a file allocation table, as it lacks the majority of the features that would make it one; there is no facility for creation of new files, movement of files, or indexing of them.

Each category of data to be stored has a space assigned for it, and at the beginning of that space a pointed indexing the location where the next element is to be stored. This is incremented *before* writing of an element commences, so that in the event of a power loss during writing a partial element may still be recovered

For the sake of efficiency, writing to the SD card is done in 512k blocks, which are buffered in the RAM of the micro-controller. This takes up approximately 10% of the total RAM, but greatly speeds up SD card access and at this point there are no memory issues.

The SD module has the following functions:

void SPISendByte(unsigned char c);

int sd\_send\_command(unsigned char cmd, unsigned char response\_type, unsigned char \*response, unsigned char \*argument);

void sd\_delay(char number);

unsigned char SPIReadByte();

int sdInit();

int sd\_set\_blocklen(unsigned long int length);

int sd\_read\_block(unsigned long int blockaddr, unsigned char \*data);

int sd\_write\_block(unsigned long int blockaddr, unsigned char \*data);

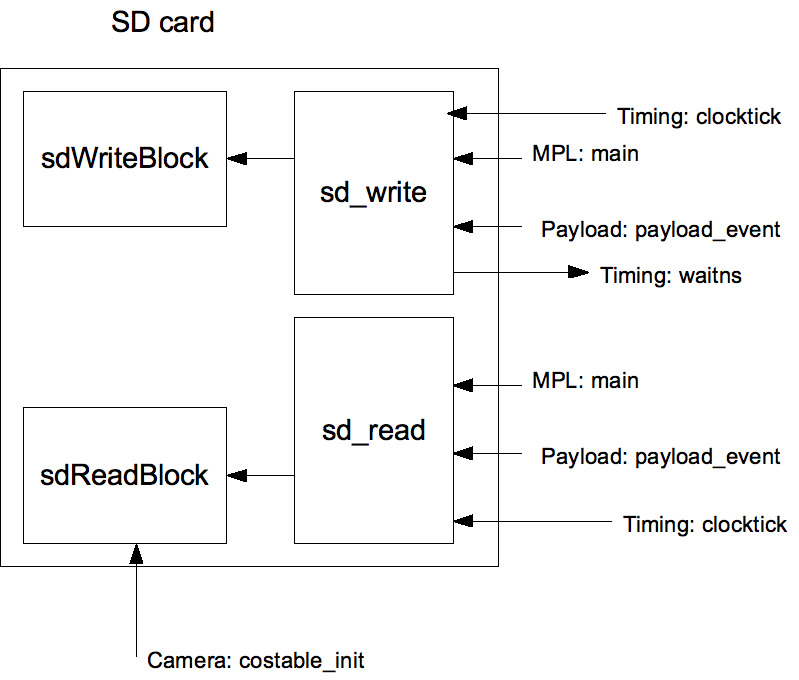
void sd\_wait\_notbusy();

void sdDisable();

void sdEnable();

int readySDForMPL();

## System Diagram



# ADC Interface

The on board ADC interface has two functions in the current design: accessing the temperature sensor that is integrated with the OBDH hardware, and converting data produced by the ADCS subsystem. The single parameter passed to this component determines which of these two sources it samples from (the individual channels of the ADCS subsystem are addressed separately.)

This component is fairly simple and consists of only a single function:

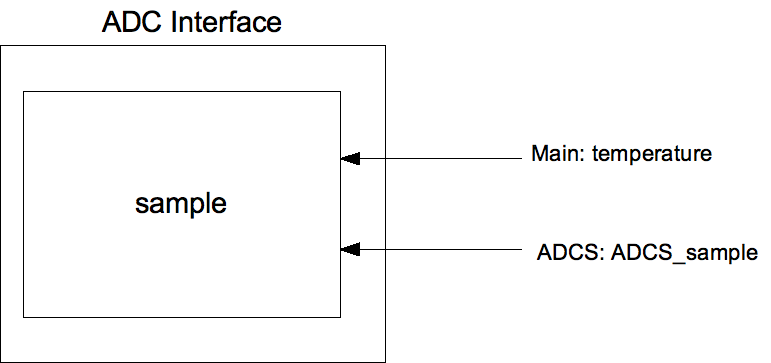
void adcInit();

void adcDisable();

void adcEnable(char ref);

void adcSample(char inchx, char srefx);

## System Diagram



# Timing

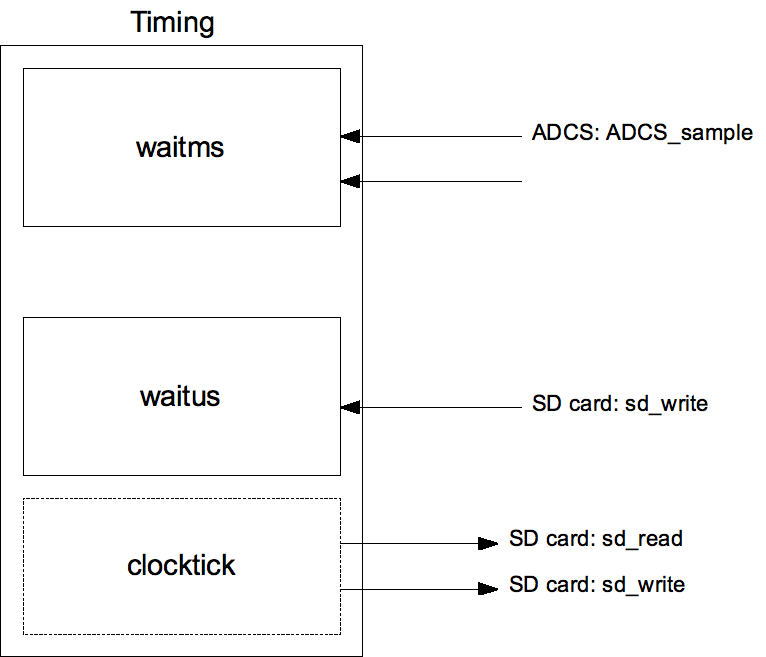
The timing module provides timekeeping services for other parts of the subsystem. These are required primarily for inserting delays measured in real time (rather than clock cycles) into code that is communicating with other subsystems' electronics.

A secondary function is to provide a long period timer that forms part of the ADCS system. It is designed to measure time in orbit by updating a counter on the SD card at predetermined intervals. It is important to note that this does not function as an independent real time clock and therefore should not be taken to be an absolute reference for time.

void waitms(int time);

WAITUS(int time);

## System Diagram



# ADCS

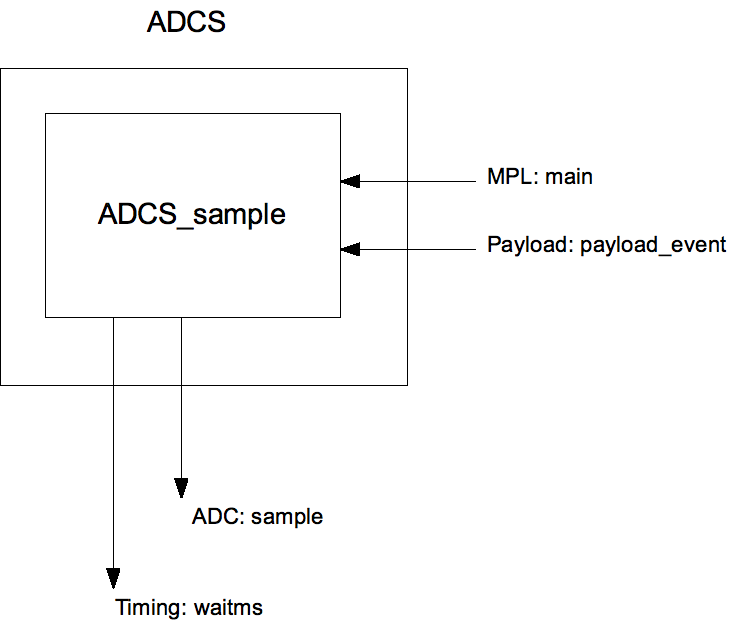
The ADCS component retrieves data from the ADCS system. It passes 4 control signals to the subsystem, and retrieves three values:

|  |  |
| --- | --- |
| 0,0 | Y-axis data |
| 1,0 | X-axis data |
| 0,0 | Y-axis data (data not used; this is required by the ADCS subsystem) |
| 0,1 | Z-axis data |

Each piece of data is a 12 bit value, stored in a 16 bit register, and returned in the sequence X, Y, Z stored in memory at the address of a pointer passed to the subroutine.

void ADCS(int \*attitude);

## System Diagram



# COMS

At this time, we do not know the details of the COMS subsystem, but we can still specify the overall structure of the software required to interface with it.

For receiving instructions from the ground station, we will define an interrupt which will be triggered by the COMS subsystem (see RS232). When a full command is received, this will then pass the command to the MPL which will then call the appropriate subroutine.

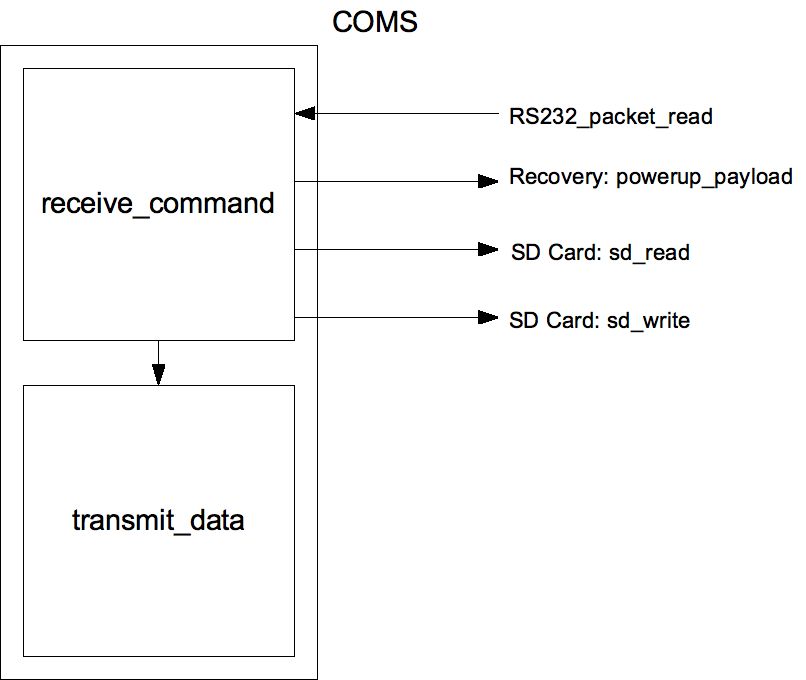
One of these commands will be to transmit information back, which will be a function called from directly by the receiving function. As the satellite will be likely receiving and transmitting during the same time period, we must use a token stored in memory to ensure that the receive interrupt aborts if it is invoked during the transmit procedure (we cannot assume the COMS system will be full duplex at this point.) Details of command can be found below.

The following functions make up the COMS component:

void receive\_command(char \*command);

void transmit\_data(char \*source, int length);

## System Diagram



# Camera

The software for the camera consists of not only the facilities to interface with the hardware, but also the subroutines to select relevant images for storage and to compress images so they are small enough to be transmitted to the ground station in a reasonable amount of time.

We are designing a piece of software called ADIL (ADIL Determines Image Lightness) that gathers information about the overall brightness of the image, on the principle that this can be used to determine whether the satellite is facing the day side of Earth – which is the only time interesting pictures could be taken with our camera in any case.

The MPL invokes the get\_image function periodically, which then invokes ADIL and based on the result invokes the remainder of this component.

Once the software has determined that a good photo can be taken, it is retrieved from the camera using the I2C bus, and then compressed to JPEG format. This compression is quite computationally intensive for this micro-controller, but it is necessary to make the images small enough. In order to facilitate this process, the discrete cosine trasform part of the JPEG algorithm will be assisted by look-up tables loaded from the SD card. This will not generate a significant overhead as the table will only have to be loaded once for each image.

The camera software uses the following functions:

int adil(char \*image);

void get\_image();

void jpeg\_compress(char \*source, char \*destination, int size, int width);

void rgb\_to\_ycc(char \*source, int size);

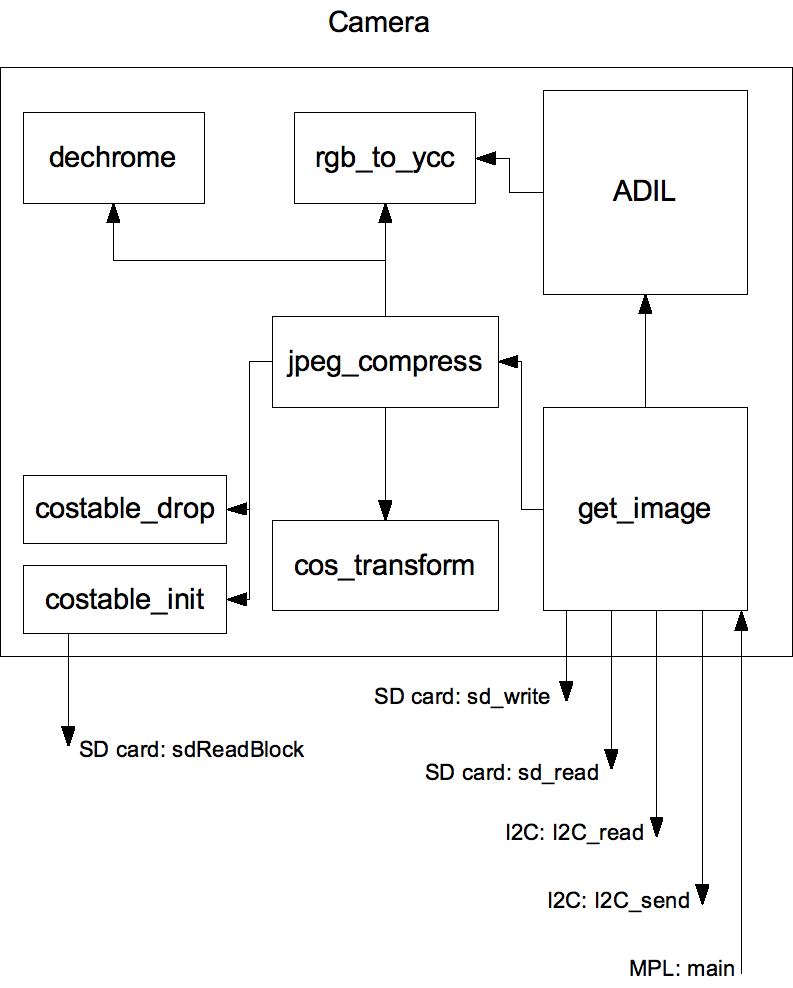
void dechrome(char \*source, int size, int width);

void cos\_transform(char \*source, char \*destination);

void costable\_init();

void costable\_drop();

## System Diagram



# Payload

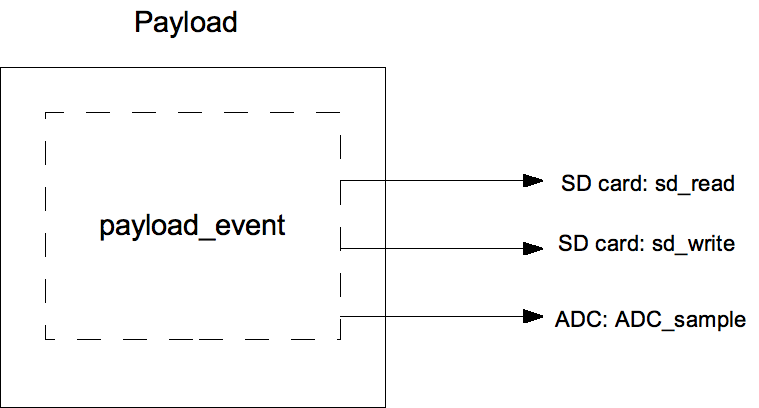
Payload data is stored as log entries as described in the MPL. The payload subroutine therefore works in much the same way as the writing portion of the MPL, but we feel it is worth duplicating a small amount of code in order to minimise the amount of co-ordination that is required between different software components.

The interrupt is triggered from hardware by the payload subsystem, and then specific data regarding the event is purposely read by the OBDH subsystem, and recorded on the SD card. Only one subroutine is required for this:

interrupt(PORT1\_VECTOR) payload\_event();

int payload\_init(int voltage, char channel);

## System Diagram



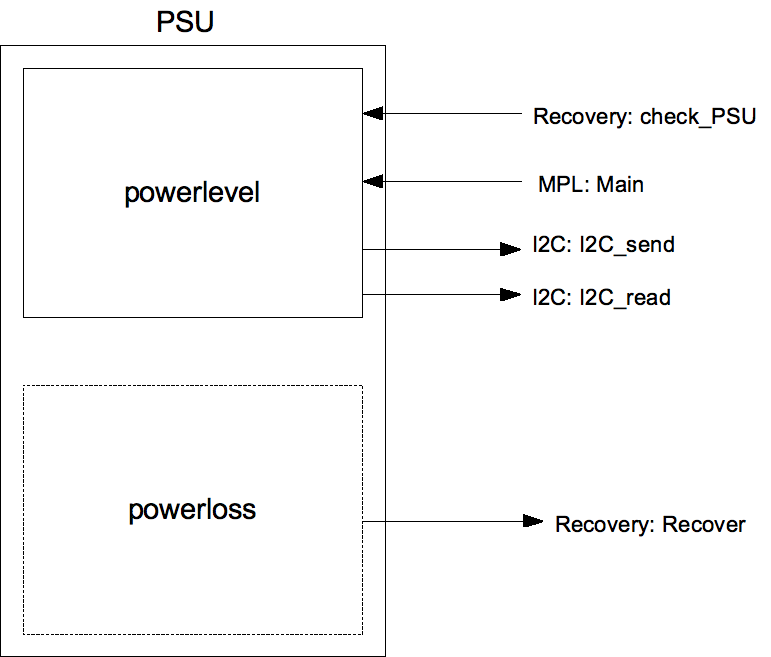
# PSU

The PSU component provides an interface with the Power Supply Unit. It not only ensures there is sufficient power available to start subsystems on recovery, it also generates interrupts during power loss and provides rough attitude information based on the relative power levels of the six solar panels located on the faces of the cubesat.

int powerlevel(int source);

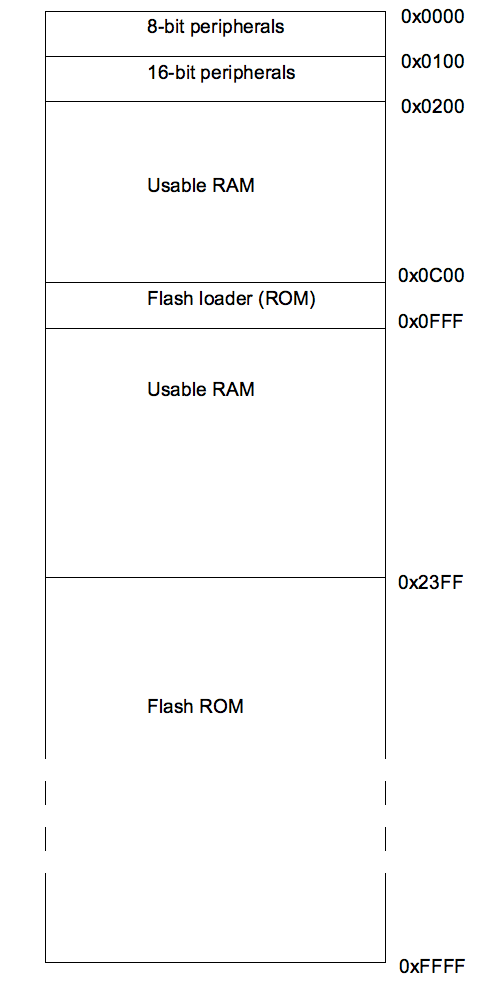
interrupt void powerloss();

## System Diagram

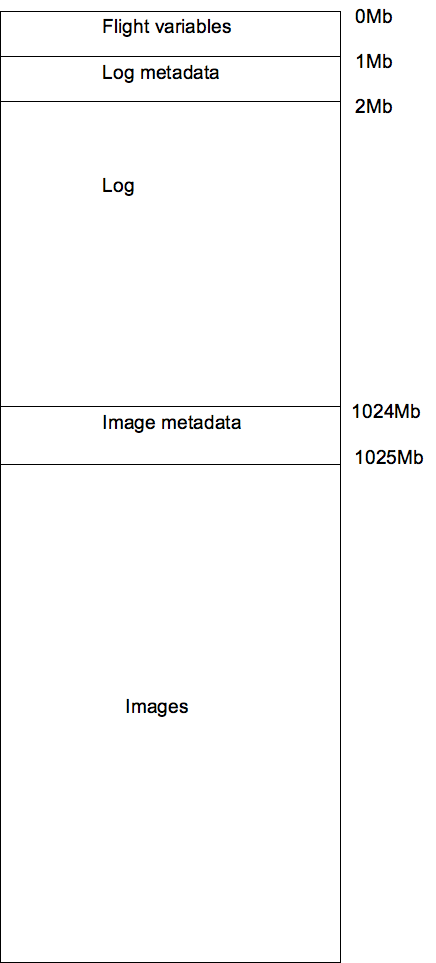


# Memory configuration

The following is a map of the internal memory of the flight controller. Note that this is not to scale.



Below is the layout of data on the SD card. Again, it is not to scale.



On reaching the end of each data area, new data will start being stored at the beginning again. If the ground station has been unable to retrieve the data being overwritten by this point, the mission will have failed for some other reason.

## Error correction

Each data block will be written twice to the SD card for error correction, as we have excessive amounts of space and are vulnerable to bit flipping due to radiation events.

1. <http://cubesatkit.com/docs/datasheet/DS_CSK_FM430_710-00252-C.pdf> for a detailed description and <http://www.cubesatkit.com/docs/cubesatkitsystemchart.pdf> for a general overview of the OBDH hardware [↑](#endnote-ref-1)
2. The ‘Libertad-1’ cubesat flew successfully on April 17th 2007, and functioned in space for 50 days [↑](#endnote-ref-2)