Documentation of AAU-Cubesat On Board Computer Software

November 8, 2002

Contents

Abstract

This document is documentation of the software written for the On Board Computer (OBC) on the AAU-cubesat. Since the software has been written by many people organized in different groups and since the development has been ongoing for long time there exists no single complete source of documentation of the software. This document try to give an overview of the software by describing structure, threads, functions and datatypes. But many things are not described in this document and must be found in the documentation from the specific group responsible for a specific piece of software,

This document is assembled by group 02gr733, but some parts of the text has been based on documentation supplied by other groups and/or persons working with the cubesat project. Work on this document is intended to be ongoing through the complete developement cycle of the satellite and it is expected that the document will include more and more details the nearer the project gets to its deadline.

Chapter 1

Overview

1.1 Introduction

The software that controls the satellite run on the OnBoard-Computer (OBC) which is a custom build board centered around the Siemens C161-RI processor. The OBC runs the RTX166 operating system from Keil. This text will document the software written for the OBC. This will be done by describing each thread running on the OS independently including its interfaces, further descriptions of system level behaviour will be given as well as details regarding specific hardware-near functions.

1.2 Thread Overview

In figure 1.1 all the software threads running on the OBC is depicted in a hierackily top-down diagram. The SuPerVisor (SPV) thread is the most important thread, since it serves as switchboard when the threads communicates and since it is the SPV that takes system wide decisions such as for example entering low-power-mode.

The threads beneath the SPV all represent a satellite subsystem such as for example the communications unit or powercontroller unit. On the third level is specific helper threads that are part of the threads of the second level.

Finally the BEACON thread in the first level is drawn with a dashed line since it only exists temporary, namely from system rest to first ground contact.

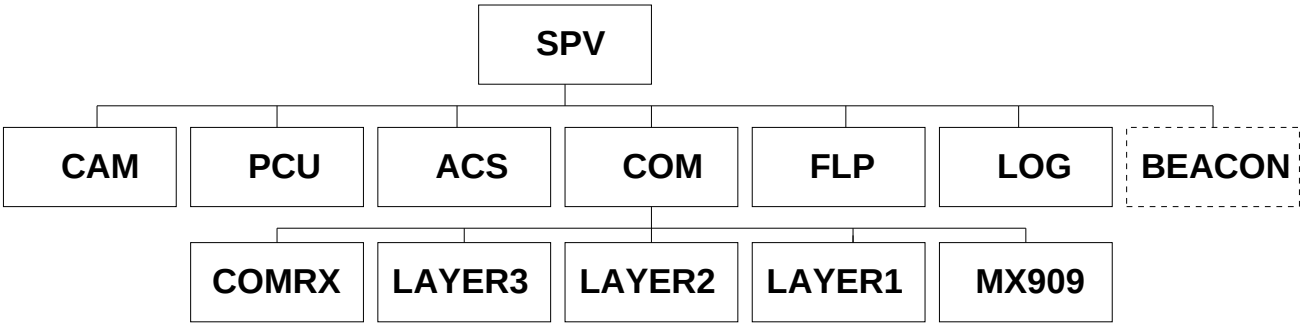

Figure 1.1: Overview of threads running on the OBC

The table below gives a short explanation of each thread. Further descriptions of each thread will follow in the next chapter.

Acronymn	Name	Function
SPV	Supervisor	The main thread wich controls the satellite
CAM	Camera	Responsible for camera operations
PCU	Power Control Unit	Controls the power hardware and level of activity
ACS	Attitude Control System	Determines Satellite attitude and control ACS HW
COM	Communication Unit	Communication Downlink from satellite to Earth
FLP	FlightPlan	Executes Commands uploaded from ground
LOG	Logging	Handles log data for ground
BEACON	Advanced Beacon	Transmits advanced beacon until earth contact established
——	————	———-Secondary threads————
COMRX	Communication Recieive	Communication uplink from earth, forwarded to SPV
Layer 1	AX25 protocol layer 1	Implements layer 1 of AX25 protocol
Layer 2	AX25 protocol layer 2	Implements layer 2 of AX25 protocol
Layer 3	AX25 protocol layer 3	Implements layer 3 of AX25 protocol
MX909	MX909 modem driver	Controls the modem, encapsulates AX25 data in mobitex format

1.3 Thread Communication

The threads communicate with each other through the mailbox facillities of the RTX166OS. The message object that is sent between threads is defined as:

```
struct Message_Agent{
   unsigned char sender;
   unsigned char command_field;
   void *data_block;
   unsigned long data_block_size;
};
```

The message agent structure as shown above, contains four entries.

- `Sender:`
  Identifies where the message is coming from; LOG, COM, CAM, ACS, PCU, FLP or SPV.

- `Command_field:`
  Identifies what function the sender wants to activate in the receiving component.

- `*Data_block:`
  This is a pointer to the data used by the function selected in the command_field. If no data is used, this value must be NULL.

- `Data_block_size:`
  This field will inform the receiver of the size of the data block.

Each component has several functions defined as commands. (In order to simplify the notation standard C semantic is used, thus the #define statements. It is intended that the definitions presented here to be used in the relevant .h include files for the different subsystem source files, in order to make some cohesion between the documentation and the code.)

Example of Thread Communication

The folloeing will illustrate how the communication features of the OS and the above defined message agent data structure is used in practize for communication between threads.

If for example the SPV wishes to find out what the power status is, the following code will do the trick:

```
//Code in SPV-thread:
Message_Agent *message;                          //Create a message agent
int result;

message = xmalloc(sizeof(Message_Agent));
message->sender = SPV;                           //We are sending from the
                                                 //SPV task
message->command_field = GetStatus;              //We want to know
                                                 //the status
message->data_block = NULL;                      //Function takes no args
message->data_block_size = 0;

result = os_send_message(PCU_M,0,message,4);     //Send the message
                                                 //for info on arguments
                                                 //see RTX166 documentation
//Code in PCU-thread:
//(At this point in the code, we assume that message agent
//has been parsed and that the GetStatus function has been called.)

status_data *reply_data;                         //Create structure
reply_data = xmalloc(sizeof(status_data));

//Read relevant values from the hardware, and fill reply_data structure

Message_Agent *message;                          //Create the message
int result;

message = xmalloc(sizeof(Message_Agent));
message->sender = PCU;                           //From the PCU task
message->command_field = PCU_data;               //Reply to GetStatus
message->data_block = reply_data;                //And this is what the
                                                 //status is
message->data_block_size = sizeof(reply_data);

result = os_send_message(SPV_M,0,message,4);     //send it!
```

It is important to notice the way the message is allocated, and subsequently deallocated.

When constructing a message, always dynamically allocate storage for it with xmalloc(); as shown in the above example. When receiving messages, it is vital that it is freed with xfree(); when it has been parsed, and no longer contain any useful information. The same goes for the data_block field, always allocate the storage with xmalloc(); and when done with it, use xfree(); on it.

To make the process of sending a message simpler a function: `sendMessage()` has been inlcuded which takes care of message allocation and message agent initialization, the function is defined as follows:

```
SendMessage(char sender, t_rtx_handle reciever, char command, void* data, long data_size)
```

With return type `void` and the arguments:

- `char sender:`
  Identifies the sending thread, Fx. PCU or COM

- `t_rtx_handle reciever:`
  Is a RTX166 handle on the mailbox that is to receive the message. These are simply the thread abbreviation with an appended "_M" string, fx. PCU_M or COM_M

- `char command:`
  Is the command that is to occupy the `command_field` of the allocated message agent. These will be defined in the headerfiles for each thread.

- `void* data:`
  Is datapointer to any data that is to go with the command.

- `long data_size:`
  Is the length of the datablock pointed to by `data` if any.

Chapter 2

Main Application Threads

This chapter will descibe each of the major tasks in the OBC software. Each thread will be described in terms of purpose and function, interfaces and a list description of all callable function will be given.

2.1 SPV - Supervisor

2.1.1 Purpose and Functional Description

The supervisor (SPV) is the organizer of the whole satellite software system. Its main function is to route information around in the satellite software structure , and make sure that vital operating parameters are within the specified limits. The SPV has a single mailbox where the subsystems can post messages, this means that in its idle state it will just wait (the thread is blocked) for input from the other threads.

When something arrives in the mailbox, the SPV is awakened and it will examine the message in order to figure out where (which subsystem) it came from. When the origin has been determined the actual command type is examined to see what the subsystem 'wanted'. An example could be that the flight plan sends the supervisor a message saying that it should get the status of the other subsystems. It will then create a message for each of the subsystems who's status information is relevant (COM, CAM, PCU, and ACS), where it indicates that the message is from the SPV and that it wants to get status information. After posting the messages to the subsystems mailboxes, it returns to its own mailbox, and waits for the reply. When a reply message arrives and it indicates that it is a reply to the get status request issued earlier, it will interpret the data, ie. data from the PCU will reveal how much power is left on the batteries.

If the data doesn't contain any critical values (ie. low power) the SPV will create a message for the LOG module, with the status data. In the event that the SPV receives a status message from the PCU, where the indicated power level of the batteries is too low for safe operation, it will set the internal lowpowerflag and issue lowpower messages to all the subsystems, informing that power is very low and they need to stop what they are doing to start conserving power. After the power level has been restored, and the SPV receives a status message from the PCU indicating this, the lowpowerflag is cleared, and the SPV sends out powerOK messages to the subsystems, to inform them that they are allowed to resume normal operation.

The SPV will also check the lowpowerflag before issuing 'heavy' jobs to the subsystems, for example when it receives a set ACS mode message from the flight plan, it first checks the lowpowerflag and if this indicates that the power level is critical, the request is ignored and the event is logged.

2.1.2 Interface Descriptions

The following tables will document the interface to each of the other threads on the OBC software. The table will show command name, direction of the command (ie. transmitted by SPV or received by SPV) and what response is expected of the receiving thread.

SPV-CAM Interface

Command	Direction	Response
GetStatus	Transmit	CAM should return CAMData structure
TakePicture	Transmit	Instructs the camera to take a picture
TransmitPicture	Transmit	Instructs Camera to send picturedata to SPV
LowPower	Transmit	No powerhandling is implemented in CAM
PowerOK	Transmit	No powerhandling is implemented in CAM
CAMData	Receive	Housekeeping information from CAM
PictureData	Recieve	Location of picture and picture length

SPV-PCU Interface

Command	Direction	Response
GetStatus	Transmit	Request housekeeping from PCU
SetBootMode	Transmit	Specifies the value of the OBC bootport, can either PROMBoot or FlashBoot
SystemReset	Transmit	Tells PCU to go through complete system reset
TurnOnSubsytem	Transmit	Tells PCU to turn on particular subsystem(s) (CAM, ACS, COM)
TurnOffSubsytem	Transmit	Tells PCU to turn off particular subsystem(s) (CAM, ACS, COM)
KickTheDog	Transmit	Resets the external OBC watch-dog-timer implemented by PCU
BasicBeaconOff	Transmit	Makes the PCU cancel the basic beacon signal
PCUData	Recieve	PCU housekeeping information
ErrorShutDown	Recieve	Tells SPV that a subsystem has been shut down by protection hardware
PCUPowerOK	Recieve	PCU informs SPV that powerlevel has returned to normal
PCULowPower	Receive	PCU informs SPV that powerlevel has turned critical

SPV-ACS Interface

Command	Direction	Response
GetStatus	Transmit	Request housekeeping from ACS
ACSData	Recieve	ACS housekeeping information
UpdateKeplerElements	Transmit	Provide new TLE’s for the ACS system
ACSParam	Transmit	New operational parameters for ACS
SunData	Transmit	New sundata from PCU
SwitchACSMode	Transmit	Request new ACS mode
Lowpower	Transmit	Tell ACS to go into low power mode
PowerOk	Transmit	Tell ACS that power is restored
TimeSynchronized	Transmit	Informs ACS that system time is valid

SPV-COM Interface

Command	Direction	Response
GetStatus	Transmit	Request housekeeping from COM
PictureData	Transmit	Instruct COM to begin downloading picyure data
LOGData	Transmit	Instruct COM to begin downloading the log content
TxSyncData	Transmit	Send satellite internal time to ground, part of time syncronization algorithm
LowPower	Transmit	Informs COM that global low power mode has been entered
PowerOK	Transmit	Informs COM that normal power mode has been entered
RxNewFlightPlan	Receive	A new flightplan has been received from ground, forward o FLP thread
RxKeplerElements	Receive	New kepler elements uploaded from ground, forward to ACS
RxTimeSync	Receive	Perform time syncronization with ground
RxFlushLog	Receive	Flush log contents to COM thread for downlink
RxNewSoftware	Receive	New Software received, do a flash reprogramming
RxSyncData	Receive	Calculated time from time syncronization algortithm
RxCommand	Receive	Execute any satellite command, data in Command Agent format
RxI2CCommand	Receive	Execute any I2C command data in Command Agent format, data block written to I2C
RxSetDebugOut	Receive	Enable/Disable output from errorhandler to ground
RxTransmitPic	Receive	Request picture data from camera thread for downlink
RxNewAcsParam	Receive	New ACS parameters have been uploaded from ground, forward to acs thread
COMStatus	Recieve	COM housekeeping information

SPV-FLP Interface

Command	Direction	Response
GetStatus	Recieve	FLP requests housekeeping from whole satellite
TakePicture	Receive	Tells SPV to initiate take picture sequence, forwarded to CAM
SetACSMode	Receive	Tells SPV that ACS mode is to be changed, forwarded to ACS
ExecuteCommand	Receive	Tells SPV to execute the command in the datablock of message (CA format)
NewFligthPlan	Transmit	Sends a new flightplan to FLP thread

SPV-LOG Interface

Command	Direction	Response
LOGData	Recieve	Log data entry recieved from LOG, forwarded directly to COM
WriteLogString	Transmit	Writes a string in the log (also defined as WriteHouseKeeping)
ReturnLog	Transmit	Tells LOG to return all LOG contents to SPV as one piece of RAM

2.1.3 Function Descriptions

The following paragraphs will describe the various functions that are implemented in the SPV. Most of them are helper functions that simply makes the switch-case constructions more readable by removing the actual implementation of the already described interface to callable functions, these functions start with **DO** followed by the message that they handle.

`void DoSetACSMode(Message_Agent *msg_in)`

Checks if the specified acsmode is valid based on the current powermode. If valid then ACS is told to change mode, otherwise a log entry is written specifying the reason for denying ACSmode shift.

`void DoExecuteCommand(Message_Agent *msg_in)`

Takes from the incoming message from the COM thread the relevant data that are encapsulated in a `Command Agent` structure and forwards the data as a normal `Message Agent` to the target subsystems, which ex-

ecuted the command as specified.

`void DoExecuteI2CCommand(Message_Agent *msg_in)`

Takes from the incoming message from the COM thread the relevant data that are encapsulated in a `Command Agent` structure writes this data directly to the I2C-bus thereby making it possible for the ground user to execute all I2C commands.

`void DoFlushLOG()`

Tells the LOG thread to assemble all log data in one continous data-area and pass it to the COM-thread for downlink.

`void DoGetStatus()`

Request statusinformation form all subsystems. Futher the function checks that the subsystems reply to the GetStatus command; if a number of MAXGETSTATUS status requests have been issued without a reply from a particular subsytem then a system reset is performed. Finally the function writes the MCU temperature in the log (SPV status).

`void DoSyncTime()`

Implements part of the time syncronization algorithm as described in section **??** on page ??. This function is used to get the internal and absolute time of the satellite, when the groundstation request a RxTimeSync.

`void SyncTimeReceived(SyncTimeData *STData)`

Implements part of the time syncronization algorithm as described in section ?? on page ??. When the groundstationg sends a RxSyncData with the corresponding data needed, the SyncTimeReceived function is used to set the absolute time onboard the satellite, using the data received.

`void SendMessage(char sender, t_rtx_handle reciever, char commandField, void *data_block, long data_block_size)`

General helper function to make it easy to send messages without care for memory allocation etc. Arguments are self explaining.

`unsigned long GetAbsTime(void)`

Returns the absolute time as defined by the OBC realtime clock. Standard Unix time is used.

2.1.4 Start-up and Watchdog Function

The following will describe what happens when the SPV is initially started and it will describe how the SPV together with the PSU hardware implements a watchdog timer mechanism.

Start-up

Whenever the SPV starts up as a result of a kill-switch release or a system reset initiated either by the errorhandler or the PSU-hardware the SPV does the following before blocking on the incoming messages queue.

1. Attaches timer interrupt for WDT function (see next paragraph)
2. Informs PSU to turn on the following subsystems:
   - ACS

- COM

3. Informs PSU to turn off basic beacon

Watch Dog Timer Functionality

The interrupt attached to the SPV is triggered each 5 seconds. When this happens the SPV issues a **Kick-TheDog** command to the SPU-thread which in turns commands the software running on the PSU-hardware to reset the watchdogtimer implemented there. If the SPV fails to trigger the WDT then the PSU-hardware will power-cycle all subsystems, wait 5 minuttes and the boot up the OBC again.

If the SPV is not interrupted within 10 seconds and no messages has been recieved during this interval then the SPV will timeout and call the errorhandler and then kick the WDT. This means that if somehow the interrupt clock source fails (e.g. due to radiation) then the system will stay alive for limited intervals in which the internal errorcount managed by the errorhandler will increase and when the threshold is reached the system will be reset. More info on the errorhandler in section **??** on page ??.

Since the WDT functionality depends on the SPV-tread, PSU-thread, operating system and I2C-bus to work, it should be well suited to indicate if the software is operating properly.

2.1.5 Time synchronization

The synchronization of the absolute time on the satellite, is used to ensure that the ACS will position the satellite at the right time regarding the flight plan. First the groundsatation will send a command request to the satellite for a time synchronization (RxTimeSync) (see figure. 2.1).

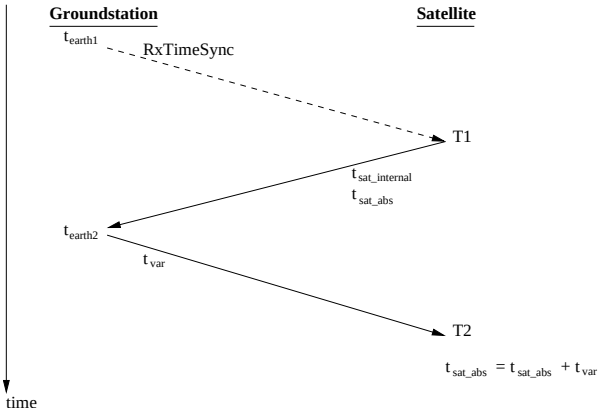

Figure 2.1: The data and communication for a time synchronization request.

Afterwards, the satellite will reply (TxSyncTime) with the absolute time (t_{sat_abs}) and internal time ($t_{sat_internal}$) on the satellite. The absolute time will be used to calculate what the difference between the satellite and the groundstation is. First the time to send a frame is calculated (t_{earth1}-T1) as this has to be included when the satellite synchronizes the time:

$$t_{frame} = \frac{t_{earth2} - t_{earth1}}{2}$$

Afterwards, the time difference the satellite needs to adjust its absolute time with (t_{var}) is calculated:

$$t_{var} = t_{frame} + t_{earth1} - t_{sat\_abs}$$

When the t_{var} is calculated the value is transmitted to the satellite (RxSyncData), which afterwards will synchronize the absolute time in correspondens with the t_{var} value.

2.1.6 Datatypes

The following will document the various datatypes defined by the SPV

SPVStatus

This structure holds statusiformation for the SPV and OBC. Specifically it contains information regarding the amount of free memory for the software and the temperature of the MCU-chip.

```
struct SPVStatus
{      unsigned long freemem;
        unsigned int MCUtemp;
}typedef SPVStatus;
```

SATTimes

This is used when the satellite answers a RxTimeSync command with the internal and absolute time on the satellite. Internal time is the number seconds past last reset and absolut time is the value of the internal RT-clock.

```
struct SATTimes
{      unsigned long internaltime;
        unsigned long abstime;
}typedef SATTimes;
```

SyncTimeData

Is sent to the satellite when the satellite is to synchronize its absolute time. The structure contains the amount of seconds that the satellite RTC must be corrected to be synchronizedn with the groundstation.

```
struct SyncTimeData
{      long time;
}typedef SyncTimeData;
```

2.2 FLP - Flightplan

2.2.1 Purpose and Functional Description

The FLP thread is responsible for the execution of the flightplan, which is a plan containing tasks that the satellite must perform at given times. The way this works is as follows:

1. Upon system initialization the FLP thread is created and a default flightplan is used initially.
2. When a new flightplan is recieved in the thread's mailbox from the ground station it is recieved as a pointer and a length. This data is then copied into a linked list of tasks, where each task is a **Task** data structure.

   The pointers to data in the Task structure will also be copied from the continous block of data sent to the mailox of the FLP-task. This is done in four steps: first the length is read from the data in the block, an area this size is allocated, data is copied from continuesly block to newly allocated block and finally the pointer in the task structure is set to point at the copied data.

   The first task in the list will then be the next task that is going to be executed, the second task the one after etc. The nextTask parameter of the last element will be NULL so the end of the list can be detected.

   When the complete linked list is created from the continous datablock then that block is deallocated.

3. The time until the first task in the list is to be executed is calculated (based on timeOfNextExecution), and the thread is set to sleep for this period of time. If this time turns out to be negative, i.e. deadline is missed, then the task is dropped and an entry is put in the log. If the task is periodic then the timeOfNextExecution is set to OStimeGet()+periodicTask.

4. After this sleep period it is checked if a new flightplan has arrived in the mailbox. If one has arrived everything from point 2 and on will be repeated. If not the first task in the linked list is sent to the SPV module, where it will be executed.

5. If the task's periodicTask value is larger than 0, it means it should be periodicly executed with the given period. In this case the task's timeOfNextExecution will be set to timeOfNextExecution + periodicTask and the task will be placed in the linked list according to this value. This re-insertion is explained below. If the task's periodicTask value is 0 it will be removed from the linked list.

6. Everything from point 3 and on is now repeated.

The re-insertion of a periodic task will be done in the following way:

1. Three pointers are created: one will point to the task that is to be re-inserted (p_reinsert), the two others will be used for going through the linked list, one of them will point to the current element(p_cur) while the other will point to the previous element(p_prev).

2. The p_prev pointer is set to the current element pointer (except initially where it will point to NULL), and the p_cur pointer is set to point to the next task.

3. If the timeOfNextExecution value of the task p_cur points to is, smaller than the one p_reinsert points to or p_cur points to NULL, the right place in the linked list has been found, and the nextTask value of p_prev will be set to p_reinsert, while the nextTask pointer of p_reinsert will be set to p_cur. If not everything from point 2 and on is repeated.

2.2.2 Interface Description

The FLP thread only has interfaces with the SPV thread through the mailboxes of each thread. THis interface has already been documented in the section describing the SPV thread.

2.2.3 Function Descriptions

`void FLP_task (void)`

The FlightPlan Task - Handles incoming messages from SPV and sends FLP task for execution to the SPV thread. Further it filters out tasks which deadlines have not been met and reports such incidents in the log.

`Task *CreateDefaultFlightPlan()`

Creates a default flightplan which consists only of a satellite statuscheck to be executed every 30 seconds. This is the initial fligthplan of the satellite.

`Task *GetNextFPTask(Task *nextTask)`

Traverses the linked list and returns next task to execute. If current task is periodic it will be inserted into the list at its correct place again.

`Task *CreateFlightPlan(Task *FlightPlan, unsigned long length)`

Creates a flightplan as a linked list from the raw flightplan recieved from ground Generates the linked by traversing the received bytestream and copies data to node structures.

```
void DeleteFlightPlan(Task *FlightPlan)
```

Traverses the linked list of the flightplan and deallocates all dynamically allocated memory.

2.2.4 Datatypes

The following will describe the various datatypes defined by the fligthplan.

Task

This structure describes a task that the fligthplan must perform at a given time. It contains an event field that described what action is to be taken and it contain information on task recursion as well as a pointer and length of any data that may be associated with the task. Further since the tasks are organized in a linked list then there is a pointer to the next task in the list.

```
struct Task
{       unsigned char event;
        unsigned long periodicTask;
        unsigned long timeOfNextExecution;
        unsigned long length;
        unsigned char xhuge *data;
        struct Task xhuge *nextTask;
}typedef Task;
```

2.3 CAM - Camera Control

2.3.1 Purpose and Functional Description

The purpose of the camera control thread is to behave as an interface between the SPV and the low-level camera code. The camera code will block on its mailbox and it will react to either a GetStatus request, TransmitPicture or a TakePicture request from the SPV.

The status returned is simply the temperature of the camera. When receiving a TakePicture command the camera will be turned on by sending a TurnOnSubsystem request to the PCU thread, then a lowlevel function is called wich disables the OS and makes the camera take a picture and store it at a predefined memory address. Hereafter the camera reenables the OS and the camera threads shuts down the camera by sending a TurnOffSubsytem request.

The TransmitPicture command from the SPV supplies an argument that specifies a blocknumber to transmit. I.e. the picture data area is divided in a number of blocks each with a size of BLOCKSIZE, these block can then be requested individually by the ground station.

2.3.2 Interface Description

The following will describe how control is transferred to the low level camera software and further it is explined how the picture is transmittet block by block.

Handing over Control to the Hardware

The following code segment shows what goes on when a picture is taken, specifically how control is transferred from the OS to the hardware:

```
SendMessage(CAM,PCU_M,CAM_ON,0,0);      //Turn on the camera
os_change_prio(127);                    //Raise priority of CAM task
```

```
os_wait_token(I2CSem,1,NILCARD);        //Get the I2C-bus exclusively
IEN=0;                                  //Turn off master interrupt
take_picture(0,0);                      //Let the HW do its stuff
IEN=1;                                  //Turn on interrupts
os_send_token(I2CSem,1);                //Release I2C-bus again
os_change_prio(CAM_PRI);                //Goto normal priority level
SendMessage(CAM,PCU_M,CAM_OFF,0,0);     //Turn off Camera again
```

This procedure makes sure that the camera has access to all needed ressources before it attempts to take a picture and it makes sure that nothing interrupts the system during the operation of taking the picture.

Picture Blocks: TransmitPicture function

When a SPV message with the command of TransmitPicture is received by the CAM-thread then it replies with a message containing a pointer into the picture data-area. This pointer is calculated according to:

$$data\_pointer = outbuffer + BLOCKNUMBER * BLOCKSIZE$$

where:
$outbuffer$: is the location of the picture in memory
$BLOCKNUMBER$: is the specified blocknumber
$BLOCKSIZE$: is the size of each block

The idea behind this scheme is that the picture can be downloaded in a number of small chunks rather than one big. This implies that the COM-unit will not be ”flooded” by data from the CAM thread and thereby makes it impossible for the groundstation to communicate with the satellite untill the complete picture is downloaded.

2.3.3 Function Descriptions

The camera thread does not implement any functions. All functionality is managed by the main switch statement. The hardware specific function that the camera uses in order to take the picture is described in chapter **??** on page ??.

2.3.4 Datatypes

The following datatypes are defined by the camera code.

camerastatus

This datastructure represent the camera housekeeping data which is only the temperature of the camera chip.

```
struct camerastatus
{
       int temperature;
}typedef camerastatus;
```

2.4 Log - Datalogger

2.4.1 Purpose and Functional Description

The LOG thread's responsibility is to time stamp and store log messages from the different subsystems. The LOG threadruns in a thread that basicly just waits at a mailbox for messages, when a message is recieved it processes it, and then waits for the next message.

The LOG thread can receive the following three commands through its mailbox: WriteHouseKeeping, WriteLogString and ReturnLog; WriteHouseKeeping is used when the data that is going to be logged is status data, WriteLogString is used to log a simple string message while ReturnLog is used to send a block with all the log data to the SPV, which will then make sure that the log data is sent to the ground station.

The storage of the log elements is implemented by using a linked list, that links the log elements together based on when they have been logged. The **Log** datastructure is used to store entries in the list.

When a new log message is recieved the previous log message is set to point to the new one, while the new log message is set to point at NULL, this is done so it iss possible to detect the end of the linked list. The commands WriteHouseKeeping and WriteLogString both just creates an entry like this.

The command ReturnLog will go through the linked list of log entries while createing a block of continuos data with the data from all the elements. When it has copied the data from a log element to this data block it will deallocate that element, so in the end, the linked list will be deleted, and all the data will be sent to the SPV.

2.4.2 Interface Descriptions

The LOG thread shares an interface with the SPV thread through the mailboxs of both threads. This interface has already been documented in the section describing the SPV thread, but the following will provide more information on intepretation of the contents of at the datapointer of the received message.

Further the LOG thread implements the `logstringwriter()` function which can be called by all other threads. See functional description in next subsection.

Messages from SPV

- Command: **WriteLogString:**
  Writes a simple string to the log
  Data_block: Pointer to the string of data
  Data_block_size: number of bytes in string

- Command: **WriteHouskeeping:**
  Writes a block of data to the log
  Data_block: Pointer to the block of data
  Data_block_size: number of bytes in block

- Command: **Returnlog:**
  Requests that the log is sent to SPV
  Data_block: NULL
  Data_block_size: NULL

Messages to SPV

- Command: **LogData:**
  Is the reply to the Returnlog command from SPV
  Data_block: Pointer to block of log data
  Data_block_size: length of datablock

2.4.3 Function Descriptions

`long GetLogSize(Log *logList)`

Traverses the linked list of log entries and calculates the total number of databytes in the log.

`void FlushLog(Log *logList)`

Allocates a continous memory area that is `GetLogSize(logList)` long. Traverses the linked list and copies all data to the memory area and frees the nodes of the list. When traversed a **LogData** message is sent to the SPV.

`void WriteToLog(Log *LastElement, Message_agent *RecvData)`

Inserts data from the recieved message (RecvData) into the newly allocated log node (LastElement).

`void LogStringWriter(char *text, char Sender)`

Convenience function that allocates and sends a message to the log thread, which contains a pointer to the `text` string and identifies itself as being sent from `sender`. Can be called from all threads.

2.4.4 The Log Datatype

Is used to store information about the log entries as well as the link to the next entry in the linked list.

```
struct Log
{
  unsigned char sender;  //Sender ID - who wrote it
  unsigned long time;    //Internal time for entry
  unsigned long length;  //Datalength in bytes
  unsigned char type;    //No apperant use???
  void *data;            //Pointer to data
  struct Log *nextLog;   //Pointer to next node
}
```

2.5 COM - Communication AX25

2.5.1 Purpose and Functional Description

The COM modules main task is to handle the communication between the satellite and the groundstation. This communicaion is split into different layers according to the OSI model (see figure 2.1).

Application	⟷	COM / COMRX
Presentation		
Session		
Transport		
Network		
Data Link	⟷	AX25
Physical	⟷	Mx909 / Radio

Table 2.1: Communication according to OSI model

COM / COMRX

The topmost layer of the communication between the ground station and the satallite consists of two tasks: COM_task and COMRX_task. Where the COM_task is used for the communication from the satellite to the groundstation, and the COMRX_task from the groundstation to the satellite.

If a message is sent to the COM_task from the SPV, it is investigated if it is data that is to be send to the groundstation or if it is a command to the COM_task regarding some operation, e.g. “LowPower”.

When the satellite receives data from the groundstation the COMRX_task checks to see if it is all the data that is received or if it just part of the data. When all data is received, it is sent to SPV (see figure 2.2)

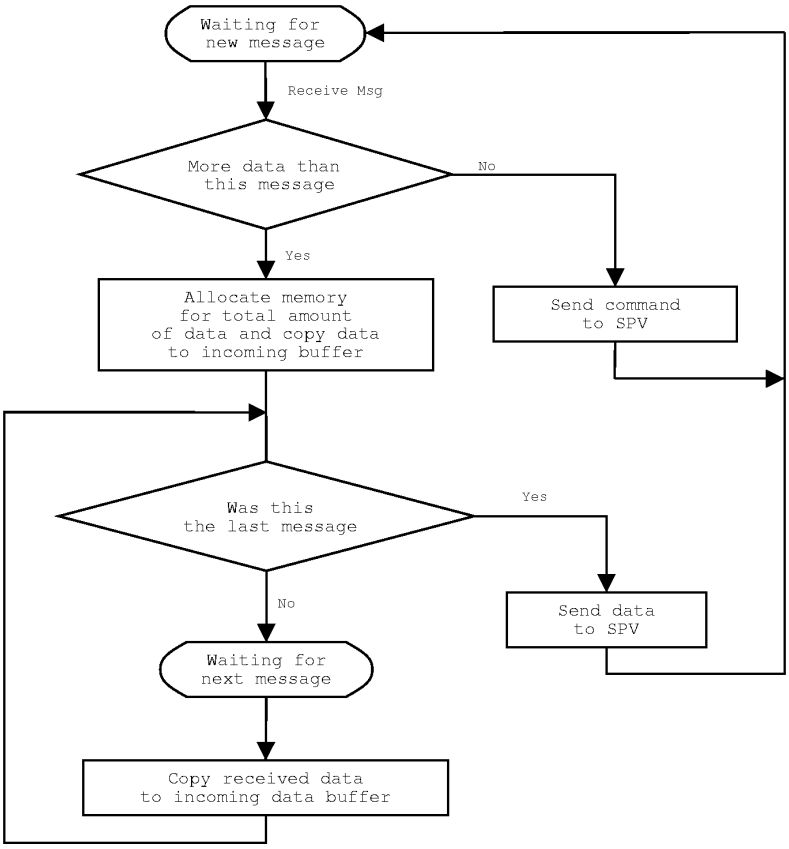

Figure 2.2: The state diagram of the COMRX_task.

Each frame sent between the COM layers must be in the following format (when sent to and received from the lower layer):

```
struct Header{
   unsigned long Length; //Length of the following data
   char CommandType;     //The kind of data that follows the header
   unsigned int CRC;     //For future use
};
```

AX25

In the communication between the groundstation and the satellite the AX25 protocol is used in the data link layer, because this protocol is designed for data communication for amateur radios and the communication will be in the amateur frequency band. I the following, each layer is shortly described. For the complete doumenentation, see [http://www.tapr.org/tapr/pdf/AX25.2.2.pdf].

Layer 3

The upper layer of the AX25 protocol has three main purposes.

One is to maintain the communication between the application layer (COM/COMRX) and the data link layer (AX25).

Another purpose is to split/assemble large parts of data when sending/receiving, since the maximum size for data through the AX25 protocol is 256 byte, and the applications need to send larger portions of data.

The last main purpose is about sliding window. In this implementation the window size is set to 1, because of the low baud rate (9600 bps). This means that Layer 3 must hold the last frame that has been sent (if a retransmission should be necessary).

Layer 2

Another important purpose of Layer 2 is to add the AX25 header, footer and bitstuffing to data being sent (and remove it from received data). The AX25 header consists of a start of frame flag, source and destination addresses, a SSID (Secondary Station Identifier) for each address a control field and a PID (protocol Identifier) field, and the footer consists of a CRC (Cyclic Redundancy Code) and end of frame flag (see figure 2.2).

Header (15 bytes)							Footer (3 bytes)	
Flag	Source	SSID	Dest.	SSID	CTRL	PID	CRC	Flag

Table 2.2: AX25 header and footer description.

Besides encoding of packets, Layer 2 can establish a connection between the ground station and the satellite and send data through this connection using acknowledge. It is also possible to send connectionless data without acknowledge; this is use when sending pictures from the satellite to the ground station.

When using connection oriented communication the layer can be in 5 different states:

Disconnected: When no connected is established between the ground station and the satellite.

AwaitingConnection: When Layer 3 asks to establish a connection or the sequence numbers (NR, SR, VA, VS, VR) are out of sequence.

AwaitingRelease: When Layer 3 asks to drop the connection

Connected: When a connection between ground station and the satellite is established.

TimerRecovery: When Timer 1 runs out (no acknowledge for packet)

Layer 1

The lower layer of the AX25 implementation (Layer 1) maintains the communication to the physical layer (Mx909). When sending data, it formats data into 16 byte frames according to the mobitex standard and sends it to the physical layer. When receiving data, it searches for start-of-frame and end-of-frame flags, and when a whole frame is received it sends it to Layer 2. The flow of Layer 1 is shown in figure 2.3 on the next page.

Mx909

Well... I can only say nonsense about this layer, so i should propably not say anything at all...

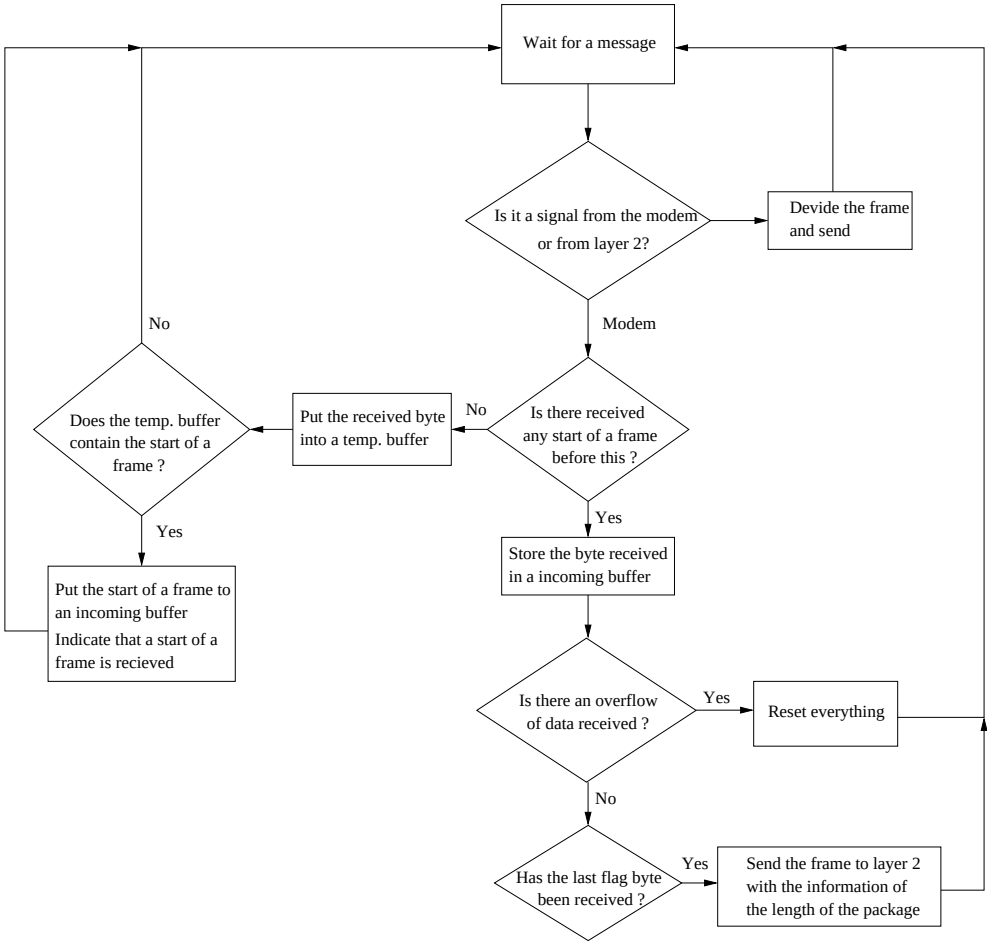

Figure 2.3: The flowchart of the layer 1 task.

2.5.2 Interface Descriptions

When sending data from one thread to another it is in general implemented using mail boxes. Each thread uses one mail box in each direction. This gives ten mail boxes for the communication from the SPV to the HW (in both directions), when using five layers between these two parts (see figure 2.4). The figure should be interpreted in such a way, that when SPV wishes to send data to the ground station, it sends the data to the mailbox "COM_M" with a transmit command. The COM task then adds a header with the size of the data and the type of data. It then sends it to the mail box "TxData_M" in Layer 3 and so forth.

In table 2.3 it is shown, in which layer headers and footers are added to the data sent. Furthermore, in table 2.4, it is shown, what the header/footer consists of, and how large it is.

			$0 \rightarrow \infty$ bytes	
App.			data	
		8 bytes	$0 \rightarrow \infty$ bytes	
COM		COMHDR	data	
	17/18 bytes	8 bytes	$0 \rightarrow 256$ bytes	
L2	L2HDR	COMHDR	data	L2FTR

Table 2.3: Headers and Footers in the protocols

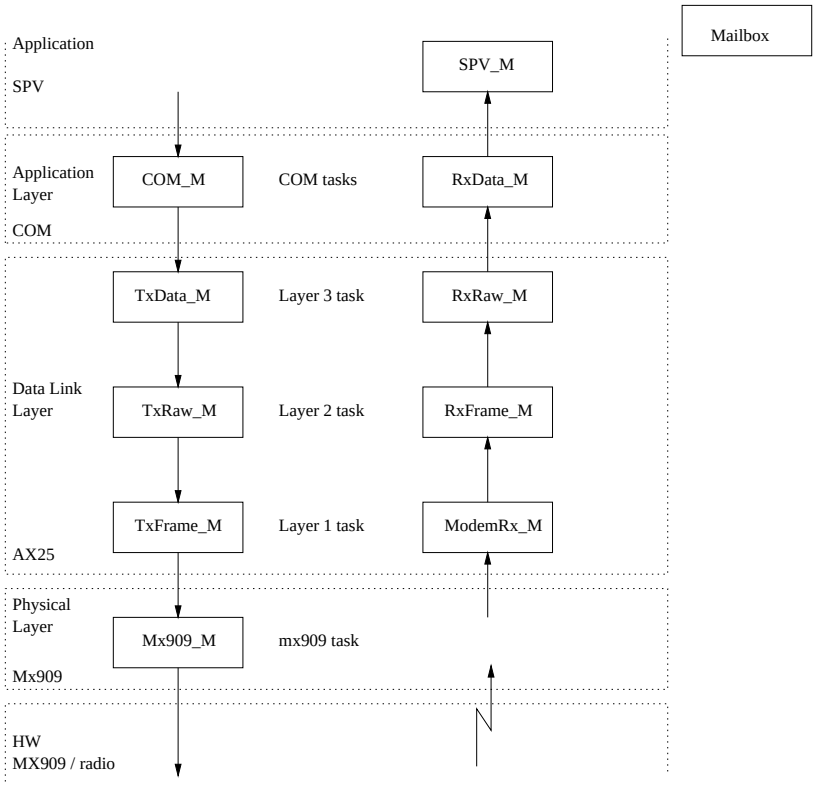

Figure 2.4: The mailboxes used for the AX25 protocols communication.

L2HDR (17/18 bytes)

1	6	1	6	1	1	1
FLAG	SRC	SSID	DST	SSID	CTRL	(PID)

COMHDR (8 bytes)

4	1	1	2
LENGTH	CMD. TYPE	···	CRC

L2FTR (3 bytes)

2	1
CRC	FLAG

Table 2.4: Description of headers and footers

Interface from SPV to COM

When sending data from SPV to COM, it must be in the format described in section 1.3 on page 3. SPV can send the following commands to COM.

GetStatus: Get the status for the COM module and also sends the beacon to get connected for the first time

PictureData: Send picture data to the groundstation

LOGData: Send log data to the groundstation

TxSyncTime: Send a handshake of the sync time

TxDebugData: Send debug data to the groundstation

LowPower: Tell the COM module that it is time to powersave mode

PowerOK: Tell the COM module that the power is OK and it can go back to normal operation

COM cannot directly send commands to SPV, it only forwards commands received from the ground station or returns status (When SPV has sent a “GetStatus” command).

Interface from COM to layer 3

When the COM module sends commands or data to Layer 3 it must be in the following format:

```
struct packet_p {
  unsigned char command_field;    //Type of data/command
  unsigned char memtype;          //Dynamic or static allocated
  unsigned long length;           //Size of data
  void xhuge *mem;                //Location of data in memory
};
```

It is possible for the COM module to establish a connection and to transmit data (either connection oriented or connectionless). This is done through the following two commands. “memtype, “length”, and “mem” is only used with the “Tranmit” command.

Connect Sends a connect request, to get a connection with ground station

Transmit Handles the transmission of the data

Interface from layer 3 to COM

Layer 3 is only able of sending data to the COM module, when sending this, it is formatted like the following

```
struct data_p {
  unsigned char command_field;    //Type of data/command
  unsigned int length;            //Size of data
  void xhuge *mem;                //Location of data in memory
};
```

Inerface from Layer 3 to Layer 2

When sending data/commands from Layer 3 to Layer 2 it must be in the same format as when sending from COM to Layer 3 (packet_p). The commands available are;

LM_SEIZE_Confirm Nothing to send at the moment (respond to LM_SEIZE_Request)

DL_DISCONNECT_Request A request to drop the connection to ground station

DL_CONNECTION_Request A request to establish a connection to ground station

DL_DATA_Request A request for sending connection oriented data to ground station

DL_UNIT_DATA_Request A request for sending connectionless data to ground station (pictures)

Interface from Layer 2 to Layer 3

Layer 2 is able of sending data Layer 3 when they are received correctly and it can tell Layer 3 that it can send the next packet. Moreover can it send indications when it changes state or errors occur. Data/Commands sent from Layer 2 to Layer 3 is in the same format as for sending from Layer 3 to COM (data_p). The commands available are:

LM_SEIZE_Request Ask Layer 3 to send a packet (retransmission)

DL_FRAME_Acked Layer 2 has got an acknowledge for the last packet, it has sent.

DL_ERROR_Indication An error as occurred.

DL_DISCONNECT_Confirm The connection has been dropped.

DL_DISCONNECT_Indication Peer has dropped the connection.

DL_CONNECT_Confirm The connection is established.

DL_CONNECT_Indication Peer has established a connection.

DL_DATA_Indication Connection oriented data.

DL_UNIT_DATA_Indication Connectionless data (not implemented above Layer 2 in this direction)

Interface from Layer 2 to Layer 1

Layer 2 can send two commands to Layer 1. One is to reset the search for frame flags and the other is to transmit a frame. When sending commands the data structure, data_p, is used (it is described above). The commands are then:

DL_TRANSMIT_Request Transmit a frame.

PL_RESET_Request Reset the search for frame flags.

Interface from Layer 1 to Layer 2

When Layer 1 has received a frame, it sends it to Layer 2 in the format described below:

```
struct frame_p {
  unsigned int length;          //Size of the frame
  void xhuge *mem;              //Location of frame in memory
};
```

Interface from Layer 1 to Mx909

...

Interface from Mx909 to Layer 1

...

2.5.3 Function Descriptions

In the following the functions used by the threads described above will be described:

COM module

`replyOK()`

This function is used to send a “Last packet received OK” to the groundstation.

Layer 2

`SelectnewT1()`
Sets a new T1 value for the timer.

`StartT1()`
Starts timer T1 from zero.

`StopT1()`
Stops the timer T1.

`UI_Check()`
This function makes a check of an unnumbered information frame.

`Establish_Datalink()`
This function is called whenever it is necessary to make a connection or a UA frame is received in the “connected” state.

`Check_Iframe_ackd()`
Is used when an information frame is received, to check if all information frames have been acknowledge.

`Enquiry_response()`
Is used by layer 2 to send an acknowledgment (RR/RNR) to the groundstation.

`Transmit_enquiry()`
Is used to send a RR frame when the timer T1 expires.

`NR_error_recovery()`

This function is used when there has gone something wrong with the sequence numbers used when sending and receiving frames

`Sequence_extractIF()`
Is used to extract the receive sequence and send sequence number from an information frame.

`Sequence_extractSF()`
Is used to extract the receive sequence and send sequence number from an supervisory frame.

`Check_need_for_response()`
Is used to determine if a response is needed when a RR or RNR frame have been received.

`RR_RNR_TimerRecovery()`
Is used by layer 2 when it is in timer recovery state and receives a RR or RNR frame.

`decode_frame()`
Is used to decode a frame when a message is received from layer 1.

`IFrame_busy()`
Is used when an information frame is received to examine if the receiver is busy and the syntax that follows from this in the SDL diagrams.

`expedite_frame()`
The function is used every time a frame is to be transmitted, it sets the P/F bit to the needed value and sets other needed informations for the frame type

`decode_frame_type()`
Is used when a frame is received to examine what kind of frame type that is received.

`clear_exception_conditions()`
Clear all conditions, is used when a disconnect is made. Also when a error has occurred and it is necessary to disconnect.

2.6 Modem driver - MX909

2.6.1 Purpose and Functional Description

The purpose of the MX909 chip and driver is to make the final connection between the software and the radio. This means that the MX909 chip, among many other takes, must make the conversion from digital to analog signal and vice versa. The modem follows the Mobitex^{TM} standard and modulates the signal using GMSK.

The MX909 chip possesses some error prevention, detection and correction features which is:

- Interleaving
- Scramble
- Forward error correction
- CRC checksum

Besides the fact that the modem appends some data to the existing package of data the modem makes the transfer of data more reliable.

The overall idea in the driver is that it sleeps until either an carrier is present or a message from Layer 1 is placed in the mailbox. This is illustrated in figure 2.5.

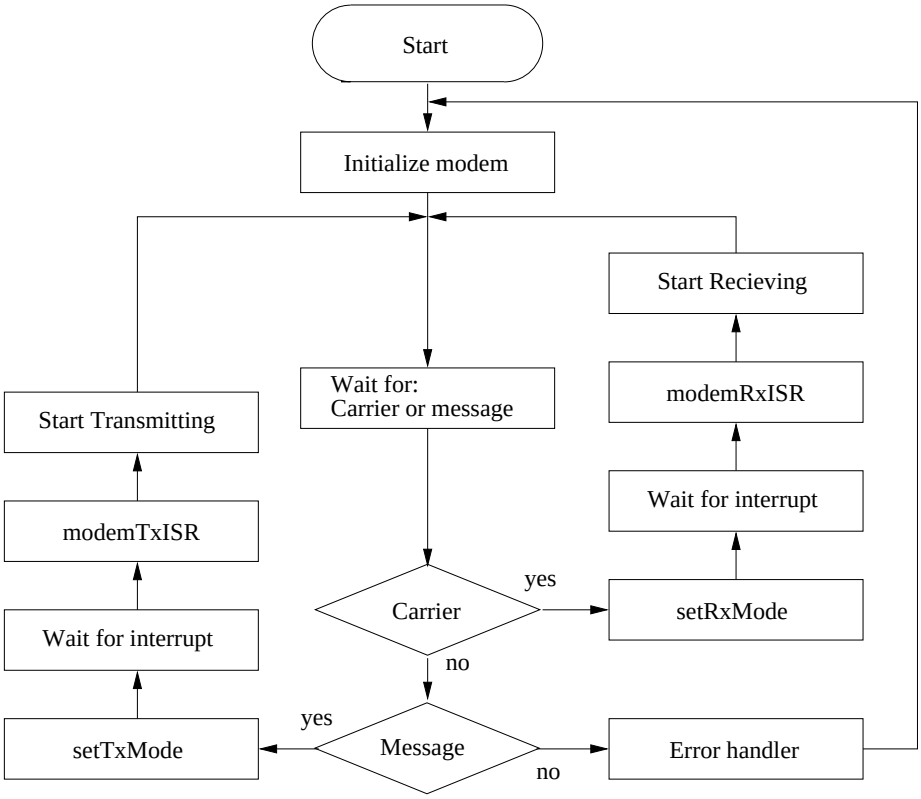

Figure 2.5: Main loop Process Flow Chart.

The implementation of the MX909 driver is done according to the flowcharts of the users manual to the MX909 chip. Minor changes has been made to adapt the driver to the system. These changes will be covered in the following text.

The first thing done after activating the thread is to initialise the modem using the `mx909Init()` function. After that the thread enters a loop which wait for either a carrier from the modem or a message from Layer 1.

If a carrier is present the modem is put into a receiving state by executing the function `setRxMode()`. Since the last step in the function is to send the command `LFSB` to the modem we then wait for a interrupt and execute `modemRxISR()` to check if the task was completed successfully. If it was successfully then we start to receive the data witch was sent. This continues until the carrier discontinues.

If a carrier was not present we check if a message is in the mailbox. If so the modem is put in transmitting state in the same way as it was put in receiving state. This means executing the functions `setTxMode()` and `modemTxISR()` and if all is well we start transmitting the data. After the data in the mailbox has been sent the thread returns to the main loop of figure 2.5.

Every time a task has been written to the modem it generates an interrupt when the task is done. To check whether the task has successfully ended an interrupt service routine (ISR) is executed. In this case the ISR is a function executed after the interrupt has been detected. The ISR function is either `modemTxISR()` if the modem is in transmit mode and `modemRxISR()` if the modem is in receive mode.

Recieving a message

The receiving part of the driver follows the users manual with a couple of minor changes. The flow chart of the receiving process is shown in figure 2.6.

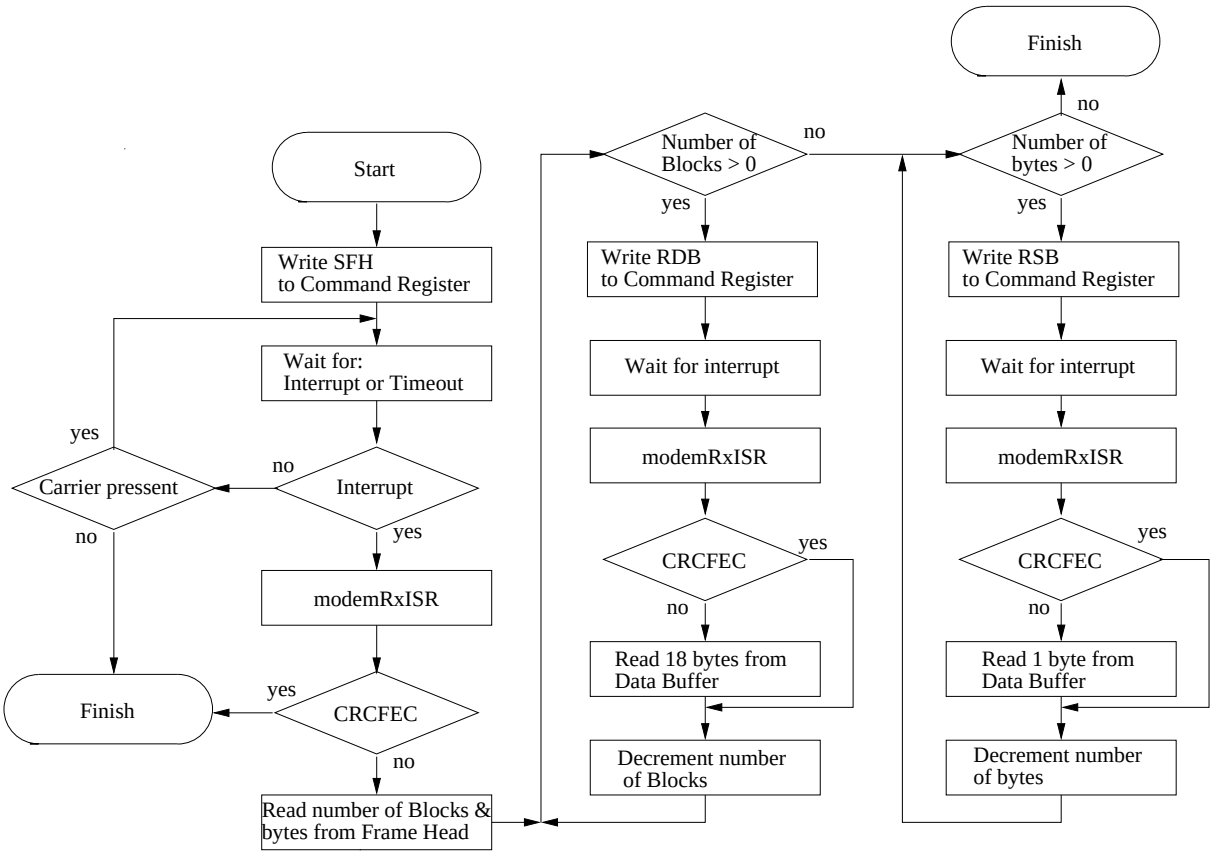

Figure 2.6: Receive Process Flow Chart.

First the we searches for a Frame Head by sending a SFH task to the Command Register. This task makes the modem send out an interrupt when it receives a Frame Head. After `modemRxISR()` the frame is checked to see if any CRC or FEC errors has ocored and if so go back to the main loop. (If the carrier is still pressent the tread goes right back to receiving).

If the Frame Head was located the control bytes are extracted. The control bytes contains the number of bytes that will follow in the Frame. With this information the number of data blocks[1] are received and then the remaining (if any) bytes are received.

The way that either a data block or a single byte is received is very similar witch can be observed from the flow chart in figure 2.6. First the task command is written to the command register. `RDB` if it is a data block and `RSB` if it is a single byte.

When the data is ready an interrupt will occur. After the `modemRxISR()` have verified that the command was executed successfully the data package or single byte is checked for CRC or FEC errors. If there is an error the data is simply not read and the loop continues. If there were no errors the data is read and passed on to the mailbox of Layer 1.

This continues until all the data blocks and bytes have been read from the frame. Afterwards it returns to the main flow process.

Transmitting a message

The transmitting part of the driver follows the users manual again with a couple of minor changes. The flow chart of the transmitting process is shown in figure 2.7.

The Frame Head is sent when the modem is put into transmitting mode in the `setTxMode()`. Therefor

[1] A data block is 18 bytes of data

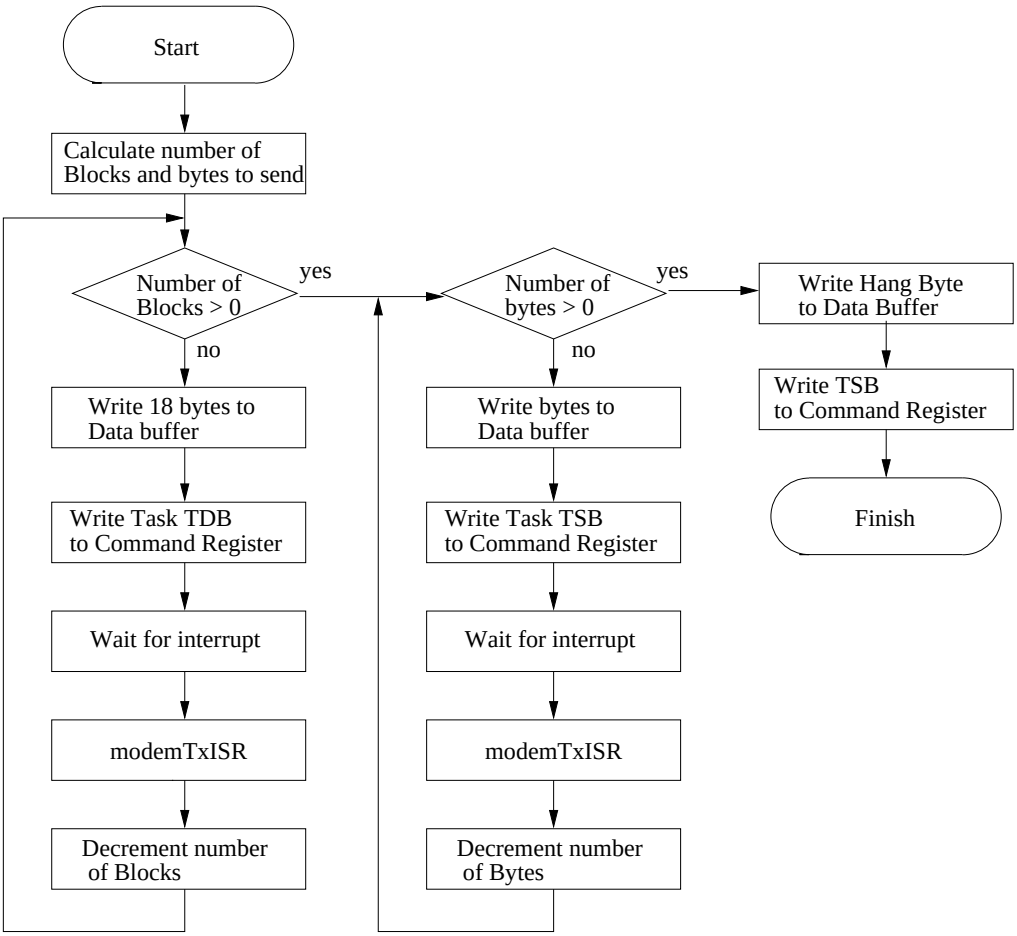

Figure 2.7: Transmit Process Flow Chart.

the modem has already sent the Frame Head when it starts this transmitting loop.

The first thing to do is to calculate the number of data blocks and the remaining bytes. After this the data blocks and single bytes needs to be sent.

It appears from figure 2.7 that the two loops is very similar. First the data block or byte is written to the data buffer of the modem and then the task TDB if it is a data block or TSB if it is a single byte is written to the command register.

When the modem has completed the task - that is read the data block or byte from the data buffer - the modem sends an interrupt and the `modemTxISR()` function is executed to make sure that the task was completed successfully. This continues until all the data has been sent.

To close the frame the Hang Byte is written to the modem and the task TSB is written to the command register.

2.6.2 Interface Description

The main idea were to make the interface between the different modules so simple as possible.

Layer1

To communicate with Layer 1 there have been created some mailboxes. These are:

- `MX909_M`
- `ModemRx_M`

The mailbox `MX909_M` is used to transport data from Layer 1 to the MX909 driver. The mailbox then need to contain a struct of data. The struct used is `data_p` (see section 2.5.2).

When some data has been read from the modem the data is written to the mailbox `ModemRx_M` one byte at a time.

Radio

To communicate to and from the modem the modems data bus is used. More specific the data bus is located on the microcontroller's port 3 (pins 0-7).

When some data needs to be sent to the modem the data is placed on the modems data bus and a task is written to the modems command register. Then when the data has been read the modem sends an interrupt. The interrupt is picked up by the driver and an interrupt service routine (ISR) is activated. In this case the ISR is a function witch is executed after the interrupt has been detected to see whether the task was executed successfully.

The procedure is much the same when data needs to be read from the modem. First the task is written to the command registry and when the task has completed the modem sends an interrupt and the data can be read.

2.6.3 Function Descriptions

In this section the different registers will be covered in the ex tense they were used. In the MX909 chip there are 8 registers. One of these registers is not used in this version of the chip.

The registers consist of 4 write only registers and 3 read only registers. The individual registers is selected by the A0 and A1 chip inputs as shown in table 2.5.

A1	A0	Write to Modem	Read from Modem
0	0	Data Buffer	Data Buffer
0	1	Command Register	Status Register
1	0	Control Register	Data Quality Register
1	1	Mode Register	not used

Table 2.5: Registers in the MX909 chip.

The way to control whether or not the modem is in read or write mode that is if the "Write to Modem" column or the "Read from Modem" column is selected is trough the $\overline{WR}$ (Write to moodem) and $\overline{RD}$ (Read from modem) chip input pins. Is $\overline{WR}$ high and $\overline{RD}$ low then the "Read from Modem" column is selected.

If a register from the "Write to Modem" is selected the data that the modem needs to read is placed on the data bus. If it is a register from "Read from Modem" the data from the modem is placed on the data bus so the driver can read it.

Data Buffer

The Data Buffer - in "Write to Modem" mode - is used to place the data that is to be sent to the radio. If it is selected in the "Read from Modem" mode the Data Buffer contains the data received from the radio (without the Mobitex^{TM} frame structure).

Command Register

This Register is primely used to place the tasks which need to be executed. After a task has completed the modem sends out an interrupt to indicate this.

A task can generally be cancelled by issuing a `RESET` task witch terminates any current action and puts the modem into a known state.

Control Register

This register controls the modem's bit rate, the response times of the receive clock extraction and signal level measurement circuits and the internal analog filters.

Mode Register

The Mode Register is used to change the mode of the modem. This is whether or not the modem is in transmit or receiving mode or if the interrupt output pin on the chip should be enabled. In this way the Mode Register controls inverting bits or not, scramble enable or not, powersave mode or not and finely Data Quality IRQ Enable or not.

Status Register

The Status Register is read only and is an readout of witch state the modem is in. By reading the Status Register it can be told whether or not a command task was completed successfully or not.

Data Quality Register

The Data Quality Register can be used to see the "quality" of the signal. Actually the Data Quality Register is an indication of the signal to noise ratio (SNR). In this case the register is not used.

2.7 PCU - Power Control

2.7.1 Purpose and Functional Description

The PCU thread is resposible for communication between OBC software and the control software embedded in the PCU hardware, therefore it mainly acts as the mean of communicating requests from the SPV unto the I2C-bus. Also the thread is responsible to keep an updated copy of the PCU houskeeping data in memory and based on these data the PCU software determines the power status of the satellite. A functional description of the thread is given in the following flowchart.

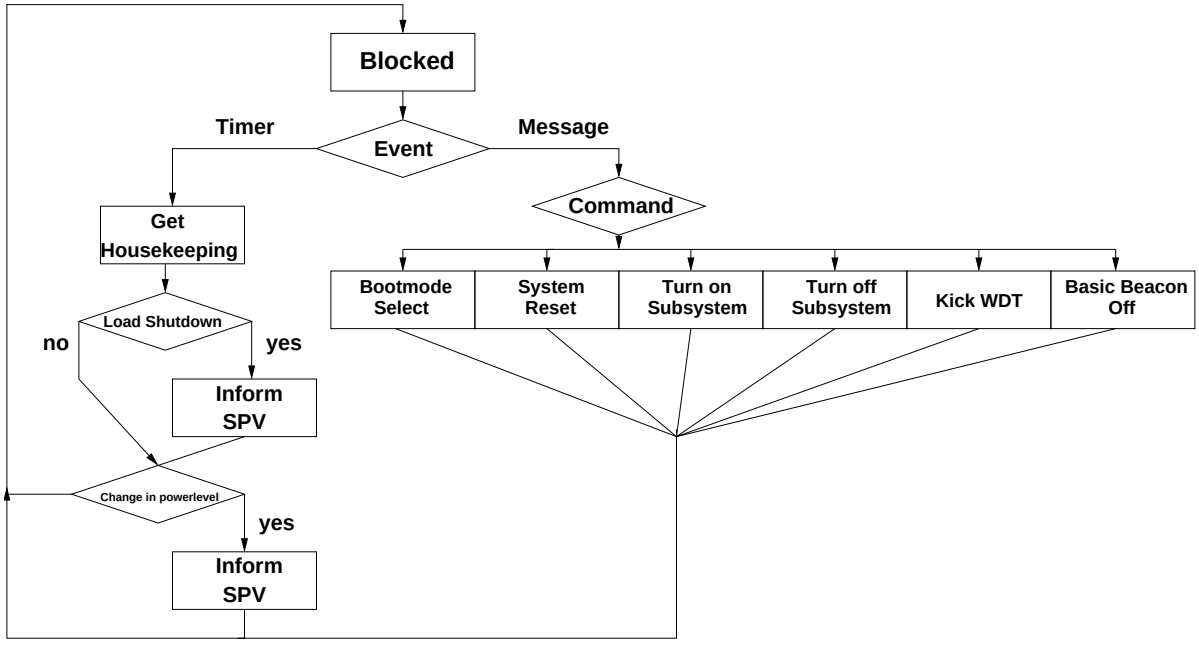

Figure 2.8: Flowchart og the PCU thread

As can be seen the thread reacts to either a incoming message or a timeout signal from the OS which ensures that the thread is run every 5 seconds. If the thread wakes up due to the timeout event it acquires

new housekeeping information from the PCU through the I2C bus and checks this information to see if a subsystem has been shut down by protection circuitry or if the powerlevel has changed. If so then the SPV is informed by sending message from the PSU.

If the thread is awoken due to an incoming message then the message is parsed and actions are taken according to the actual command, whereafter the thread enters the blocked state again.

2.7.2 Interface Description - I2C

The following will define the interface between software on the OBC and on the PCU hardware.

Requests from OBC

The OBC can issue 6 different requests with each will result in a different response from the PSU, which respectively are shown in the table below:

Request	OBC request	PSU response	Module
1	Set boot port to PROM	Boot port is set to PROM	31
2	Set boot port to EEPROM	Boot port is set to EEPROM	30
3	Reset Watchdog	External Watchdog is reset	29
4	Turn on specific subsystem	Specific subsystem are turned on	25-28
5	Turn off specific subsystem	Specific subsystem are turned off	21-24
6	Send specific housekeeping	Requested housekeeping are send	1-9

- Set boot port to PROM
  When the OBC issues the request “Set boot port to PROM” with module number 31 and data length 0 the PSU sets P3.0=0.

- Set boot port to EEPROM
  When the OBC issues the request “Set boot port to EEPROM” with module number 30 and data length 0 the PSU sets P3.0=1.

- Reset Watchdog timer
  When the OBC issues the request “Reset Watchdog timer” with module number 29 and data length 0 the PSU must reset the watchdog register. Note that the timer must also always be reset after any other requests from the OBC.

- Turn on subsystem
  When the OBC issues the request “Turn on specific subsystem” with a module number from 25 to 28 and data length 0 the PSU must turn the specific subsystem on. The subsystem are defined by the module number:

Module number	25	26	27	28
Subsystem	OBC	ACS	CAM	TRD

- Turn off subsystem
  When the OBC issues the request “Turn off specific subsystem” with a module number from 21 to 24 and data length 0 the PSU must turn the specific subsystem off. The subsystem are defined by the module number:

Module number	21	22	23	24
Subsystem	OBC	ACS	CAM	TRD

  If OBC is specified then this corresponds to a system reset in which all subsystems are power cycled.

- Send Housekeeping Information
  When the OBC issues the request ”Send specific housekeeping” with a module number form 0 - 17 and a data length 0, the PSU must return the requested data. The housekeeping data are defined by the module number:

Housekeeping	Module number	Data length
Battery voltage & MPPTC current out	1	4 bytes
Solar cells current 1 & 2	2	4 bytes
Solar cells current 3 & 4	3	4 bytes
Solar cells current 5 & voltage	4	4 bytes
Bus voltage & PSU current	5	4 bytes
OBC & CAM & TRD & ACS temperatur	6	4 bytes
PSU & T6 & T7 temperatur & Load status	7	4 bytes
OBC & CAM current	8	4 bytes
TRD & ACS current	9	4 bytes

The load status uses a 1 for load on and 0 for load off and is depicted at figure 2.9.

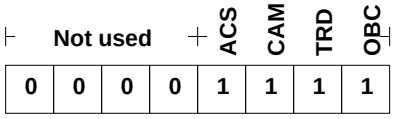

Figure 2.9: The load status byte

PSU messages to OBC

The PSU only have two messaged that it can send to the OBC and that is whether the received data was valid or not:

Message	Module number	Reaction from OBC
Data invalid	20	OBC retransmit the data
Data valid	19	OBC does nothing

Example of Use

2.7.3 Interface Description - SPV

The follwing tables will list commands that can be sent between the PCU thread and the SPV thread. Actions taken by the PCU thread are also listed.

Messages from SPV

Command	**Action taken by PCU thread**
GetStatus	Replies with a PCUdata message that contains housekeeping information
SetBootmode	Addresses the module number on the I2C bus that enables the requested bootmode
SystemReset	Addresses the module number on the I2C bus that makes the PCU reset the system
TurnOnSubsystem	Addresses the module number on the I2C bus that enables the requested subsystem
TurnOffSubsystem	Addresses the module number on the I2C bus that shut down the requested subsystem
KickTheDog	Addresses the module number on the I2C bus that resets the OBC external WDT
BasicBeaconOff	Addresses the module number on the I2C bus that shut down the basic beacon signal

Messages to SPV

Command	Reason to send message
PCUdata	Is the reply to the GetStatus command and contains all housekeeping info from PCU
ErrorShutDown	Informs SPV that a subsystem has been shutdown due to a harware fault
PCULowPower	Informs SPV that the satellite has entered the low power state
PCUPowerOk	Informs SPV that the satellite has entered the normal power state

2.7.4 Function Descriptions

The following will descibe the functions that are implemented in the PCU thread.

`PCU_task()`

Is the main function of the thread. Initially upon initialization it creates the mailbox for the thread and tells the OS to execute the thread periodically every 5 seconds.

After initialization it enters an infinte loop in which it blocks until awoken by the OS due to a timeout or an incoming message. If a message is recieved it calls the `PCU_handlemessage()`. If awoken due to a timeout it will call `PCU_update()` to acquire new housekeeping info and then it calls `PCU_statuscheck()` to detmermine the current powerlevel.

When the message has been handled or when the statuscheck has completed then the thread will block again.

`PCU_handlemessage()`

This function parses the incoming message using a switch-case construction. Actions will be taken according to the incoming message as described in the table in the interface section.

`PCU_update()`

Will update the housekeeping information stored on the OBC by requesting all housekeeping modules from the PCU-HW through the I2C bus. See I2C specification.

If new data suggests that a subsystem has been shut down by hardware then this is informed to the SPV.

`PCU_statuscheck()`

This function will check the battery powerlevel and signal changes to the SPV. The powerlevel is defined by a low and a high threshold battery voltage. To avoid ”flickering” the powerlevel must remain constant for at least 15 seconds before a global powerlevel change is intiated.

2.7.5 Datatypes

The following will describe datatypes defined by the PSU thread.

BootMode

This datatype is used to specify bootmode of the OBC. It is sent from the SPV to the PSU when SPV wants to change mode. And it is also updated as part of the OBC bootprotocol.

```
#define FlashBoot 0
#define PROMBoot 1

struct BootMode
{      unsigned int Mode;
}typedef BootMode;
```

PCUStatus

This datastructure contains all houskeeping information originating from the PSU hardware.

```
struct PCUStatus
{      unsigned int battery_voltage;
        unsigned int solarcell_voltage;
        unsigned int battery_current;
        unsigned int psu_resets;
        unsigned int obc_resets;
        unsigned int input_wh;
        unsigned int solarpanel_current[5];
        unsigned int bus_voltage;
        unsigned int load_currents[4];      //Indexes follow normal definitions
        unsigned int temperatures[4];       //Indexes follow normal definitions
        char load_status;                   //Uses the proposed n^2 definition of su
        char boot_status;                   //current value of bootpin
}typedef PCUStatus;
```

2.8 ACS - Attitude Determination and Control

2.8.1 Purpose and Functional Description

The ACS thread is responsible for calculating the attitude of the satellite in regard to the sun using inputs from the sunsensors and the magnetometer. Further it controls the PIC uC on the ACS PCB.

During normal operation the thread runs once each second and during each run it:

1. Acquires sensor data, as well as other HK data, from the PIC
2. Calculates the attitude
3. validates status of soft and hardware in the ACDS system
4. Sends commands to the actual control algorithms on the PIC as well as commands regarding mode of operation etc.

The thread will also respond to commands from the SPV. These are mainly exchange of status information and system paramaters, as well as commands from the SPV to put the ACS in specific modes, e.g. camera mode.

Extended Kalman Filter

The C code for the Extended Kalman Filter is a rewritten version of the matlab file 'impEKF.m' made by [Krogh and Schreder, 2002]

In order to improve numerical stability, factorization is introduced to the error covariance a priori and a posteriori updates. This Extended Kalman Filter uses Bierman's 'Square Root Free' square root observational update and Thornton's modified weighted Gram-Schmidt algorithm [Grewal and Andrews, 2001]. It is the Matlab version of the EKF to be implemented as C-code in the ADCS RT-thread.

```
Inputs:
Beci  Magnetic field vector in ECI frame (unit)  [3x1]
Seci  Sun vector in ECI frame (unit)             [3x1]
```

```
Bscp  Magnetic field vector in SCP frame (unit)  [3x1]
Sscp  Sun vector in SCP frame (unit)             [3x1]
nctr  Control vector (Current*DutyCycle) in mA   [3x1]
qP    A posteriori quaternion (or initial)       [4x1]
wP    A posteriori ang. velocities (or initial)  [3x1]

Outputs:
qP    Estimated a posteriori quaternion          [4x1]
wP    Estimated a posteriori ang. velocities     [3x1] (rad/sec)

SCP frame: SpaceCraft Principal axis frame
ECI frame: Earth Centered Inertial frame
```

Note that before using sensor data and control torque currents, these must be rotated from the SCB frame to the SCP frame. This is a constant rotation, which may be neglegible if the principal axes turns out to be the same as the geometrical axes. The EKF is divided into: Initialization, Prediction and Filtering

Initialization

- Most of the EKF related initilization is in the CubeSat_library_init file.
- It is triggered externally using 'init' input when the EKF is initialized.

Prediction

- A priori estimates of the state [qM;wM].
- Apply contribution from control torques to a priori state estimate.
- Update state transition matrix
- Thornton's UD temporal update algorithm is used to find P(-) = UDU'

Filtering (Measurement update)

- Rotation matrix (ECI2SCP) is made of a priori q
- Rotate vectors in ECI to SCP
- Update measurement sensitivity matrix
- Find innovations of measurements
- Use Bierman to determine P(+) = UDU' and difference dx between true state and estimate
- State vector update [qP;wP]

2.8.2 Interface Descriptions

The ACS thread communicates with the SPV through mailboxes and with the ACS hardware using the I2C bus. The following will elaborate on these interfaces.

Interface to SPV

The commands and data passed between the supervisor and the ACS are all described in the SPV documentation. In general ACS recieve parameter and commands from SPV, but only replies to the GetStatus request.

Interface to ACS HW on I2C-bus

The ACS reads sensor data from the ACS hardware on the I2C-bus and writes back control signals to the controllers implemented there. To facillitate the communication a number of helperfuntions have been written, which are resposible for moving the informaation around. These functions are found in the file: ACSI2C.c. The functions are:

char acs_change_mode(ubyte mode)

I2C modulenumber: 0x03
Changes ACS mode on the PIC uC on the ACS hardware.

char acs_read_algorithm(void)

I2C module number: 0x01
Will return the number of the algorithm in use on the ACS PIC.

char acs_change_algorithm(unsigned char algorithm)

I2C module number: 0x04
Changes the used algorihm on the ACS PIC.

char acs_external_control(int *data)

Issues a series of I2C commands to put the ACS PIC into external control mode.

char acs_update_reference(uword *data)

I2C module number: 0x02
Moves a new reference to the control algorithms on the ACS PIC.

char acs_housekeeping(ubyte *i2cbuf_in)

I2C module number: 0x00
Reads all housekeeping information from the ACS PIC.

2.8.3 Function Descriptions

The following describes the functinality in each of the functions implemented in the acs-thread. This information is mainly taken from the comments from the sourcecode.

int VecLength(vector xhuge *vect, float xhuge *length)

Function that determines the length of the input vector

int unitvec(vector xhuge *vect)

Function for making vector into unit vector

int RotMatrix(quat *q, vector *vec)

Function for rotating vector (pointed to by vec) using a rotation described by input quaternion (pointed to by q)

int qxqconj (quat xhuge *qA, quat xhuge *qB, quat xhuge *qOut)

Determines the quaternion product of qA and conjugated qB, and returns result to qA
Inputs: qA first quaternion, qB second quaternion (european quaternions)
Output: qOut = qA (*) qB_conj = (C.3) in [Krogh and Schreder]

double raise(double base, int n)

Function for replaicing the pow function. It raises a double base to an integer n and returns a double. Notice that n is integer and ¿= 0. This function is written as a replacement of the pow() function, since this function as implemented in the C166 library did not work correctly with variables placed beyond a 16bit boundary.

int sunModel(double Time, vector xhuge *vec)

On-board reference sun model for determining direction to Sun

int AlbedoCorrection (vector xhuge *posvec, vector xhuge *sunvec, unsigned int xhuge *paramINT)

Albedo correction

- Posvec points to satellite (must be neg)
- Sunvec points to Sun

int posModel(double JD, Kepler xhuge *tle, vector xhuge *vec)

Function for determining satellite position in orbit

- Input in Julian Date since 2000
- Output in meters and ECI frame

int TestSunlight(float xhuge *paramFP, vector xhuge *Posvec, vector xhuge *Sunvec)

Use angle between sunvector and position vector from reference models to determine wether we are in eclipse
Inputs: position vector and sun unit vector (ECI frame)
Output: 1= Sunlight, 0= Eclipse

int magModel(double Time, vector xhuge *Pvec, vector xhuge *Bvec)

Magnetic field model for determining direction of B field

- Input position in meters and ECI frame (ECI2ECEF is in start of function)
- Output B vector in nanoTesla (Notice! different from Matlab files using Tesla)
- Output frame is ECI (ECEF2ECI is in end of function)

int DefaultParameters(unsigned int xhuge *paramINT, float xhuge *paramFP)

Initialize ACS parameters and BlackList

- BlackList has been included as two 16 bit integers
- Parameters are divided into integers and floating points, numbered 0-22 and 100-121, respectively
- some params will be updated when available

int SelectMagAxes(vector xhuge *magSCB, unsigned int xhuge *paramINT)

Creates magnetometer vector - if any axis is blacklisted - then set to zero

int TdataToTemperature(unsigned int twelveBit)

Function for converting measured 12 bit data to a Celsius temperature

int SelectTemp(sixSensors xhuge *Tdata, unsigned int xhuge *paramINT)

Function for selecting not blacklisted temperature sensors

float sunDataProcess(int SunData, int Temp, float CalFactor, float TempCoeff)

Function for doing calibration and temperature correction on sun sensor data

int sunDataSelect(sixSensors xhuge *sundata, pcuSunD xhuge *sundataPanel, vector xhuge *sunDataVector, sixSensors xhuge *Tdata, unsigned int xhuge *paramINT, float xhuge *paramFP)

Function for selecting not blacklisted sun data from primary or secondary sensors and creating sun vector from processed sun data

int ControlCurrents(vector xhuge *ControlSignal)

Function for processing 12 bit control signals into control current Input: Sampled 12 bit value stored in vector struct
Output: Control current [A]

int processI2C(ACSstat xhuge *HK, unsigned char xhuge *dataBytes)

Function for reading I2C bus and saving HK and sensor data pool

```
ExampL: tempdata 8 LSB: XXXXXXXX
sundata  8 LSB: YYYYYYYY
temp+sun 4 MSB: YYYYXXXX

temp 12 bit   : XXXXXXXX | ((YYYYXXXX & 00001111) << 8)
sun  12 bit   : YYYYYYYY | ((YYYYXXXX & 11110000) << 4)
```

int moveDataToPool(ACSstat xhuge *HK, sixSensors xhuge *Tdata, sixSensors xhuge *sunData, vector xhuge *magData, vector xhuge *Controlvector)

Function for placing sensor data into sensor data pool

int markley(quat xhuge *q, float xhuge *paramFP, vector xhuge *magvec, vector xhuge *sunvec, vector xhuge *magdata, vector xhuge *sundata)

Deterministic attitude determination algorithm

int Q2Omega(quat xhuge *q, quat xhuge *q_, vector xhuge *w)

Determine angular velocities from q(k) and q(k-1)

int FIRfilter(vector xhuge *w, const float *FIR, float *DelayX, float *DelayY, float *DelayZ)

FIR filter for determining angular velocity when using Markley for attitude determination

int modeSelect(ACS_Mode xhuge *REQ_mode, ACS_Mode xhuge *PIC_mode, ACS_Mode xhuge *Mode, vector xhuge *w, unsigned int xhuge *paramINT, float xhuge *paramFP, int xhuge *TimeDetumble)

Function for selecting the correct Mode, based on REQ_mode from DHCS and PIC_mode from ADCS-PIC

int PhiUpdate(float xhuge *phi, vector xhuge *w, float xhuge *paramFP)

Updates state transition matrix

int Hupdate(float xhuge *H, vector xhuge *B, vector xhuge *S)

Update Measurement sensitivity matrix

int thorntonUD(float xhuge *phi, float xhuge *UD)

Makes Thornton UD Temporal update algorithm as in [Grewal and Andrews] page 251 table 6.16. This is in order to find the UD factorization of the a priori covariance error: P(-) = UDU'

It first multiplies PHI*U in place as described in [Grewal and Andrews] page 250. It does not include prediction of the state, that is done from the calling matlab function.

Inputs: Phi State transition matrix, U UD factors of a posteriori process noise P(+) [6x6], Din UD factors of a posteriori process noise P(+) [6x6], UDq UD factors of Q, upper unit matrix = I [6x6]

Outputs: UD UD factors of a priori process noise matrix P(-) [6x6]

NOTE1: Since U is upper unit triangular, the diagonal of UD is used for the diagonal matrix D, in order to save space, when storing the error covariance matrix P=UDU. D is, however, stored in the temporary Din for alterations.

NOTE2: Q is diagonal so UD factorization is simple Q=Uq*Dq*Uq', where Uq is a 6x6 identity matrix and Dq is the same as Q. The upper unit triangular matrix and the diagonal made from the process noise Q are stored together in one matrix UDq.

Input matrices Phi, U and Uq are updated in the algorithm.

int bierman(float xhuge *H, float xhuge *paramFP, float xhuge *UD, float xhuge *dz, vector xhuge *dw, quat xhuge *dq)

This algorithm is for making part of Biermans 'Square Root free square root observational update'. It calculates the upper unit triangular matrix U and the diagonal matrix D, being the UD factorization of the a posteriori covariance error. P(+) = UDU'

As inspiration the algorithm from the [Grewal and Andrews] floppy-disk:/CHAPTER6/BIERMAN.M is used. This has been upgraded with an extra for-loop, in order to use algorithm with six measurements.

Instead of using measurements as inputs, the error dz between the estimated sun and magnetic field vectors and the measured sun and magnetic field vectors are passed to the function.

The algorithm only updates the state error dx, not the full state. Reason: Quaternion update is special.

Inputs: H Measurement sensitivity matrix [6x6], R Diag of variance of measurement uncertainty mtrx [6x1], UD UD factors of a priori process noise matrix P(-) [6x6], dz Innovations of measurements [6x1]

Outputs: dw Difference between true w state and estimate [6x1], dq Difference between true q state and estimate [6x1], UD UD factors of a posteriori process noise P(+) [6x6]

NOTE: Since U is upper unit triangular, the diagonal of UD is used for the diagonal matrix D, in order to save space, when storing the error covariance matrix P=UDU. Input UD is overwritten with output UD.

int ekf(vector xhuge *Beci, vector xhuge *Seci, vector xhuge *Bscp, vector xhuge *Sscp, vector xhuge *nctr, quat xhuge *qP, vector xhuge *wP, float xhuge *paramFP)

The extended Kalman filter algorithm. It calls Thornton and Bierman

int TestConv(unsigned int xhuge *paramINT, float xhuge *paramFP, vector xhuge *Beci, vector xhuge *Bscb)

Function for testing if convergence has happened or FIR filter settled

int AttDetMethod(unsigned int xhuge *paramINT, float xhuge *paramFP, quat xhuge *q, vector xhuge *w, int sunlight)

Function determining attitude determination method

int ACSinputs(unsigned int xhuge *data, ACSstat xhuge *HK, quat xhuge *q, quat xhuge *qref, vector xhuge *w)

Saves attitude quaternion and ang. vel vector as housekeeping and determines the intputs for the ACS subsystem

- Housekeeping will be attitude quaternion and ang. vel. vector
- Data to ACS-PIC will be quaternion describing attitude error and the anguler vel. vector

Inputs: q, unit attitude quaternion (type quat), w,angular velocities(type vector) Output: data-¿q, two byte integer to PIC (attitude error), data-¿w, two byte integer to PIC (angular velocities), HK-¿ACSqx, two byte integer to housekeeping (attitude quaternion), HK-¿ACSw_x, two byte integer to housekeeping (angular velocities)

int TestAlgONOFF(unsigned char *AlgorithmONOFF, ACSstat xhuge *HK, unsigned int xhuge *paramINT)

Makes a test to see if it is necessary to update AlgONOFF in PIC (Coil and magnetometer settings)

int InitializeGlobals()

Function for initializing global variables

2.8.4 Datatypes

The following will document the datatypes defined by the ACS system.

Quat

Structure defined in order to handle quaternions.

```
struct quat
{      float q1;
        float q2;
        float q3;
        float q4;
}typedef quat;
```

ACS_mode

Used when ACS mode information is passed to and from the ACS module. Contains a mode identifier and an associated quarternion. Possible modes are listed as well.

```
//Allowable modes:
#define Detumbling              0
#define Fail_safe               1
#define Power_save              2
#define Camera                  3
#define Force_Power_save        4
#define Force_Camera            5
```

```
#define Init                    6
#define NoReply                 7

struct ACS_Mode
{       unsigned int mode;
         quat references;
}typedef ACS_Mode;
```

ACSstat

Contains all housekeeping information from the ACDS system.

```
struct ACSstat
{       int ACSmode;                    //Current ACS mode
         int ACSerrorCode;
         int ACSmagnetometer_x;
         int ACSmagnetometer_y;
         int ACSmagnetometer_z;
         int ACSsun1;
         int ACSsun2;
         int ACSsun3;
         int ACSsun4;
         int ACSsun5;
         int ACSsun6;
         int ACSmagnetorquer_x;
         int ACSmagnetorquer_y;
         int ACSmagnetorquer_z;
         int ACStemp1;
         int ACStemp2;
         int ACStemp3;
         int ACStemp4;
         int ACStemp5;
         int ACStemp6;
         int ACSq1;
         int ACSq2;
         int ACSq3;
         int ACSq4;
         int ACSw_x;
         int ACSw_y;
         int ACSw_z;
}typedef ACSstat;
```

Kepler

The datastructure descibed the kepler elements for the satellite.

```
struct Kepler
{       double Epoch_Time;
         double Inclination;
         double RA_of_node;
         double Eccentricity;
```

```
        double Arg_of_perigee;
        double Mean_anomaly;
        double Mean_motion;
        double Mean_motion_dot;
        long Checksum;
}typedef Kepler;
```

ACS_Param

Used to pass various ACS parameters.

```
struct ACS_Param
{
  char Number;
  long Value;
}typedef ACS_Param;
```

pcuSunD

Used to tell the ACS module about the solarpanel currents. These currents are measured by the PSU and each time housekeeping information are collected then the SPV extracts this information and sends it to the ACS thread.

```
struct pcuSunD
{    int solarpanel_current[5];
}typedef pcuSunD;
```

2.8.5 sixSensors

Structure for sunsensors and temperature sensors

```
struct sixSensors
{      signed int nr1;
        signed int nr2;
        signed int nr3;
        signed int nr4;
        signed int nr5;
        signed int nr6;
}typedef sixSensors;
```

vector

Datastructure to store three dimensional vector

```
struct vector
{      float x;
        float y;
        float z;
}typedef vector;
```

2.9 BEACON - Advanced Beacon Signal

The satellite has to beacon modes; The first is a morse signal generated by the PSU and it is transmittet in intervals until the OBC has been succesfully booted. Hereafter the satellite begins to transmit the advanced beacon. The advanced beacon includes the following information:

```
Call sign [14] : char[]
Text string [18] : 'www.cubesat.auc.dk'
Internal time [4] : long int
SW error count [2] : short int
MCU temperature [4] : long int
Battery voltage [2] : short int
Boot status [1] : byte (FLASH/PROM)
Terminator [1] : '|0'
```

This signal is transmittet once every 2 minuttes in normal power mode and if the powerlevel drops then only every 4 minuttes. When a connection has been established with the groundstation the the beacon will be turned off and the beacon thread is killed. The beacon signal is sent directly to the MX909 modem driver and is therefore not encapsulated in the AX25 protocol.

Chapter 3

Hardware Support Routines

NOTE: This chapter is to be supplied by Rasmus Olesen, which have written most of the functions

3.1 I2C communication

3.2 Camera Operations

3.3 Flash-ROM operations

3.4 MCU Frequency Control

Chapter 4

Other Software

4.1 Error Handling

Software error handling is managed in cooperation between the operating system and two functions written for the purpose. In general terms the errorhandling scheme will inrease a counter each time an error is encountered, log the incident and when the errorcount increases beyond a specific number then a system reset will be performed.

All operating system functions that generates an error will make the OS call an errorhandler. In our case each thread will, as the first part of thread initialization, perform the function call:

```
os_set_error_handler(user_error_handler);
```

Which ensures that not the default OS error handler is called. The user errorhadler is implemented as follows:

```
#pragma reentrant
void user_error_handler (t_rtx_exit err_code,t_sys_call_id system_call)
{ t_rtx_handle task;

  task=os_running_task_id();
  sprintf(errorbuffer1,"error no. %d in function %d called from task no. %d", err_code,
  errorHandler(errorbuffer1,0);
}
```

This function assembles all information available from the OS about the nature of the error in a textstring and the calls the top-level errorhandler which is implemented as follows:

```
#pragma reentrant
void errorHandler (char * errorString, signed int errorStatus)
{  error_count++;

   //If number of errors is exeeded then do system reset using the PCUHW
   if (error_count>AllowedErrors)
   {   unsigned char COMMAND[]={21,0};
         while(1)
        {
         print("ERRORHANDLER: SYSTEM RESET",DOTHER);
```

```
          i2c_write(PCU_I2C,COMMAND,1);
        }
    }

    // Put incident in log if initialization is completed and debug_output enabled
    if (OSTimeGet()>1)
    { sprintf(errorbuffer2,"ERRORHANDLER: %s, Status:%d, ErrorCount %d",errorString,e
      if (Debug_output) logstringwriter(errorbuffer2,ERROR);
      print(errorbuffer2,DOTHER);
    }
}
```

This function checks if the maximum number of errors is exeeded and in that case it resets the system by writing the reset command directly to the PCU hardware using the I2C-bus. If the number of errors are within the allowed range it writes the error to the log (if log error output is enabled) and further hands the errorstring to the print function, which will output it depending on system configuration. However since the system is not ”stable” during system initialization a test is performed in other not to enbale any output during the first second of OBC operation.

The reasoning behind splitting the errorhandler in two functions are that while the `user_error_handler` only can be called by OS functions, te `ErrorHandler` can be called directly by any thread which discovers that somthing is wrong with the software, e.g. the thread was not able to allocate memory.

4.2 Debugging Support

Debugging support is offered by the support by the means of a **print()** function that allows the software to write debugging strings. These strings can be directed to various outputs depending on debugging configuration.

Syntax

The syntax for the debugging function is:
void print(char *txt, char tag, ...)

The ***txt** argument is exactly the same as with the **printf** function. The **tag** argument specifies from which thread the string is written. It is then possible from `debug.h` to enable or disable output from specific threads. The **...** part of the argument string is also as with the **printf** function.

Configuration

Configuration of debugging is controlled by the `debug.h` file. The following options can be defined or undefined:

- **STDDBUG** Enables or disables debugging output globally. If not enabled the **printf** function will expand to an empty function.
- **SEROUT** If this option is defined then all debugging information will be outputted to the serial port.
- **LOGOUT** If this option is enbled the all debuggin information will be sent to the log module and can then be downloaded using the AX25 protocol

Further it is possible to define which subsystems should be debugged by controlling which threads are allowed to outpt debugging information