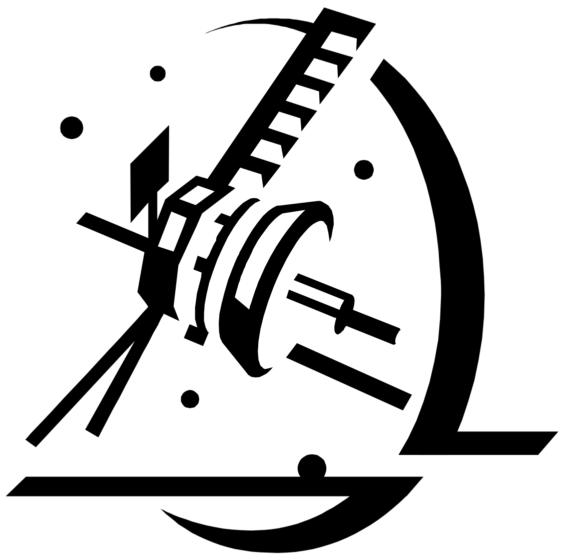
# On-board software



Department: IMM, DTU.
Course: Introduction to satellite systems and design.
Counsellor: Hans Henrik Løvengreen
Date: 30/09/2001

c971744, Cecilie M. Larsen
c970681, Gitte Marianne Nørgaard
c971789, Kim G. Jensen
c971403, Morten F. Fiisgaard
c971483, Søren Hjarlvig
c971651, Tue B. Jensen

## Introduction

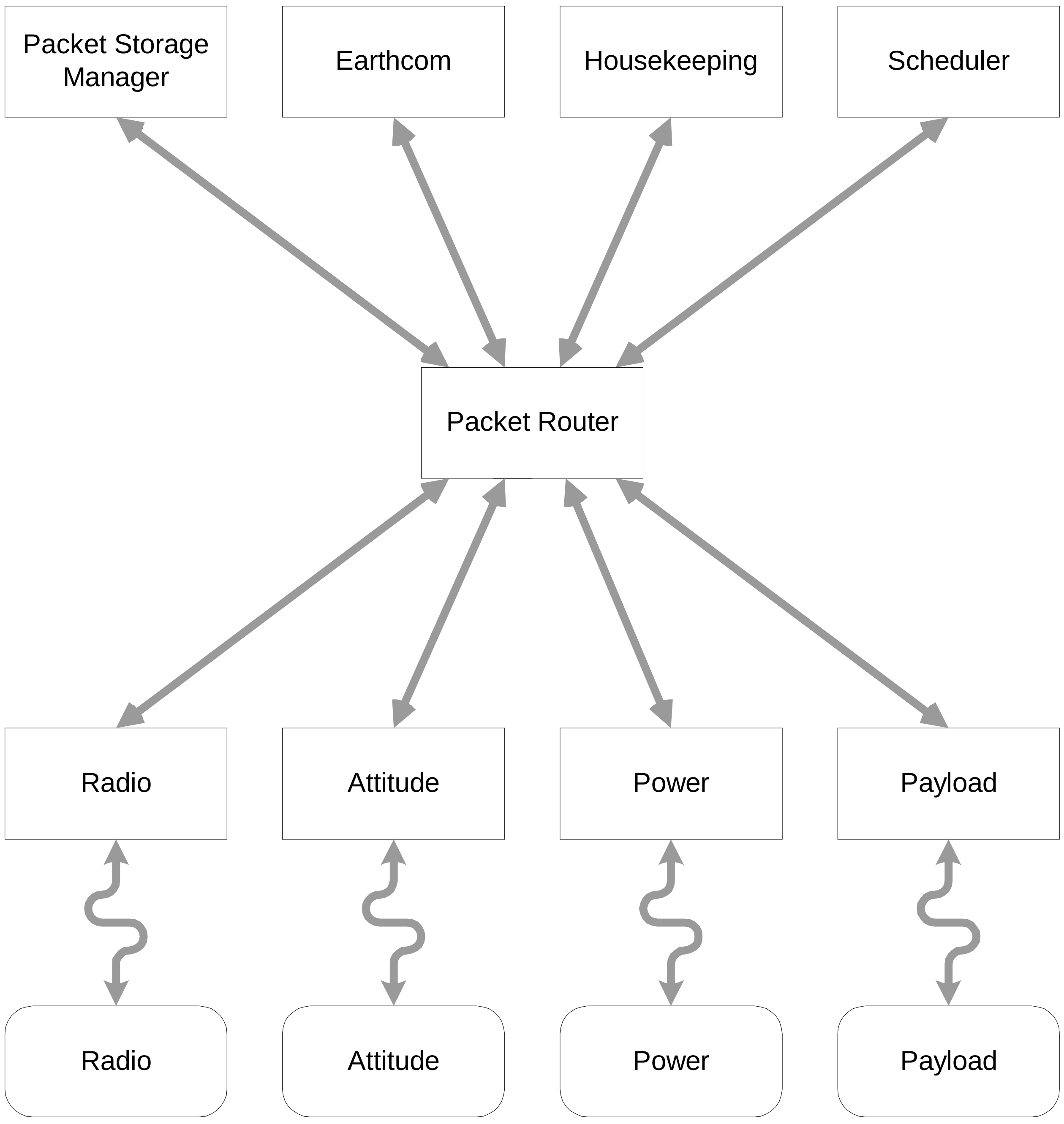
This document describes how the software running on the onboard computer of a satellite could be organized and which services it should provide.
Further more it is described how the different subsystems of the satellite are represented.

The issues covered include:
Basic organization and representation of subsystems
Inter-process communication
Storage management
Prioritising of downlink communication
Scheduling
Collection of housekeeping data
Process priorities & time critical components

The proposed software model is very much inspired from the one used on the Ørsted satellite, it is very modularised and flexible approach that allows parallel development on the different subsystems.
The proposed packet format is also inspired from the one used on Ørsted, which is a subset of the ESA – Packet utilisation standard.

## Basic organization and representation of subsystems

In this paper we will introduce the following system. We will concentrate on the five topmost processes.



Packet Storage Manager: Allocates memory to each packet upon demand.
EarthCom: Stores messages for Earth when there is not radio contact.
Housekeeping: Ask each process to send housekeeping data.
Scheduler: Stores commands and passes them on to the processes on time.
Packet Router: Routes packets from one process to another.

## Packet format

The packet format will definitely be chosen when we know which runtime environment we have, because the environment may have some facilities we can use.

### General message format

To make the internal communication as easy as possible we need to define a general message format that all messages must use. This format should make it easy for the packet router to handle the messages.
We still need to find out how much data the different processes will need to send, so we will have to build in a maximum packet size and a possibility for splitting a packet up in several packets.

**Header**
The header contains information about the data following in the data field, and we will at least need the following fields:

| | |
|---|---|
| Address: | Telling the packet router where to deliver this packet. The address could either be an id that the router knows, or the actual address of the receiving process. |
| Length: | The length of the data, zero if no data is needed. |

Other fields can be necessary depending on the needs in the systems. For instance could a sender address come in handy, when housekeeping data is sent to Earth.
To define these other fields we will need to analyse the needs of the different processes.

**Data**
Each of the processes must decide how to handle the data sent to them for each of their commands, the data area should at least have a command field.

| | |
|---|---|
| Command: | The command tells the receiving process what the packet is about. Each of the processes is able to define their own commands, as long they use the general message format. |

**Tail**
The tail of a message should possible hold a checksum of some kind to enable error detection. As these messages are only used between processes in the satellite they should be error free, but some of the commands could be stored in the scheduler for quite a while, and could possibly get a bit turned during that interval. So we need to determine if this error correction is needed and if so who has the responsibility of checking it. This could for instance be done by the receiving process or by the packet router.

### To do

We will have to consider the needs of each group, in order to design the final packet format. We will then have to explain the groups how to use the format.

We will have to decide if we make the functionality for splitting up a packet to several smaller packets, or if each of the processes has to do it. If most of the processes need this functionality we will do it, but if it is only one process, it might be wiser to hide it away in that process and its message data.

## Packet-router

What is at packet-router and why do we need one?

Basically a packet-router is a piece of soft or hardware, which brings/sends packets from one unit to another unit. A unit is in this case, either a hardware component or a process running in the operating-systems runtime-environment.
If we were without a packet-router, the unit of the system could not send messages to each other. Normally an operating system has a packet-router, but we cannot be sure.

### Design and technology

At this point of the process we will not be able to design the packet-router to the satellite, because the CPU and operating system have not been chosen yet. The following section will therefore only describe some of the design question in a software-based packet-router, and not a fully designed packet-router.

*The asynchronous design*

In this design patterns, the units are not sending their packet directly to the packet–router, but to a queue. The packet-router will look in this queue for packets to send.

Advantage:

The units will not wait or halt, if the packet-router is busy.
The queue can be implemented as a priority queue.

*The synchronous design*

The units are not sending the packets through a queue, but directly to the packet-router

Advantage:

We can use the wormhole technology.

*Wormholes*

A wormhole is a well-known technology for packet routing at the Internet. When the packet-router has received the first segment of the packet, also called the header (see packet description), it will open a connection to the receiver unit. In this way the packet-router can receive the rest of a packet, while it is sending the first part of the packet.

Advantage:

There will be a lower latency, than without wormholes.

It will also be necessary to look in to these subjects:

Routing table
Error handling
Queue

## Proposed solution:

We should use the asynchronous design. The packet router shall have two packet buffers for each software process in the system - one dedicated for input and one for output.
Any buffer shall be able to contain exactly one packet.

*Case example:*
Process A wishes to send a packet with size n, to process B.
A asks the packet router for a new packet of size n.
The packet router passes the request to the packet storage manager, and returns the result to A - either a pointer to the new packet, or an out of memory error.
Process A fills the header of the packet and places the packet pointer on A's dedicated out buffer on the packet router. The packet router looks through the different processes out-buffers. When it locates the packet in A's out-buffer, it looks up the recipient in the packet header, in this case process B. It then moves the package from A's out-buffer to B's input-buffer. It is now up to process B to collect the packet from its dedicated input-buffer. It is the receiving processes responsibility to dislocate the memory used by packets that are no longer going to be used.

If a process tries to place another packet in its out-buffer before the packet router has moved the first one, the process should be denied or have to wait. The reason that the buffers shall be able to contain exactly one packet is to avoid that a faulty process could use up all memory by sending masses of meaningless packets to the packet router. The suggested approach ensures that the previous packet sent by a process has been routed before the process I allowed to submit another packet.

# Packet Storage Manager

## General description

The packet storage manager (PSM) has the responsibility of allocating memory for a new packet needed somewhere in the satellite.
If the packet is for an internal command in the satellite there should always be allocated space for it, if there is not enough free space, then the lowest priority message for Earth should be deleted. Messages meant for Earth can be deleted if we are in need of more memory, as they are not vital for the survival of the satellite. It is impossible to say how important a message meant for a process in the satellite is, so these messages must not be deleted in any circumstances.
If the packet is meant to hold data for Earth it should only be created if there is enough free space, or if the data has a so high priority, that it will be possibly to free enough space by deleting other packet for Earth with a lower priority.

## Proposed solution

We assume that the environment on which we build our system has some kind of memory management that will be able to allocate and free space from the memory in some way. When the PSM gets a request for a packet, it will ask the environments memory management if it there is enough free space. If there is enough it allocates the space and sends the pointer to the process, otherwise it will ask EartCom to delete. If there still is not enough free space the process will get an out of memory error.
This means that each of the processes must be able to handle an out of memory error for a request for a packet to Earth.
As PSM does not delete any packets it is important that each of the processes remembers to delete any packets that is not used any more.
To ensure that no process allocates a packet that fills all of the free space we will define a maximum packet size. The maximum size will be defined according to how much data it is possible to send in one packet through the radio.

### Commands

We have two kinds of data in our system. Data that has been collected and are to be sent to Earth as soon as possible and data that are used internal in the satellite.
Data used internal in the satellite must not be deleted before the process it was meant for has read it. Packets meant for Earth can be deleted if something more important comes around.

GetPacket(size)

> This function return a packet that can be used as an internal command in the satellite, it will not be deleted before a process deletes it.
> The process asking for the packet must fill all fields by itself.

GetPacket(size, priority)

> This function returns a packet that a process can fill with data for Earth; the data field will have size given as argument.
> The packet returned is really a packet in a packet. The inner packet, which is to be sent from EarthCom to the radio when appropriate, is filled with radio address, length and radio send-command when

returned. The outer packet is pre-filled with the address for Earthcom, the length of the packet, the priority of the packet and a command telling EarthCom that this is data for Earth.

An example of a packet addressed to EarthCom:

| Address: | Length: | Data: | | | | | |
|---|---|---|---|---|---|---|---|
| | | Command: | Priority: | Address: | Length: | Command: | Data: |
| EarthCom | ??? | SendToEarth | ??? | Radio | ??? | send | 010101 001011 |

## Possible errors and what to do

If something goes wrong and we generates a lot of packets, we will delete all packets for Earth, and so we will have no information on what went wrong. Therefore we will allocate some space for messages for Earth, so we always get the highest priority packets down.

As a process will need a packet to ask for a packet all processes will need to have a default packet allocated from start that it can use to ask for a packet. Off course all processes must be able to handle the case where a packet request times out.

# EarthCom

## General description

The purpose of the communication module EarthCom is to collect packets containing housekeeping data and payload data and transmit it when radio contact is made.
It shall be possible to add new packets to the collection, to delete some packets to release memory in order to prevent overload and to notify that the radio module is ready for transmission.

## Proposed solution

The module should be able to select which data to transmit and delete first.
EarthCom should maintain a priority list – ordered by the priority of the packets and the order in which the packets is received by this module to enable transmitting the most important data and deleting the least important data.
It is assumed that the received packets to be inserted in the list (and possible later transmitted) contain a field stating the priority of the packet.

When radio contact is made the collected data is to be transmitted to the radio module. The module should also maintain information concerning which packet is being sent at the moment to enable reinserting the packet in the priority list if transmission to Earth fails.

### Assumptions

EarthCom only receives pointers to the packets.
The priority of each packet depends upon which data it is containing and is to be predetermined.

### Rules for list manipulation

*Rules for inserting*

A received packet is inserted at correctly in the priority list.

*Rules for deleting*

The packet with the smallest priority is deleted first.

*Rules for transmitting*

When the radio module is ready to transmit to Earth the packet at the head of the list is moved to a sentlist where it awaits acknowledgement from the radio module. If the transmission succeeds then the transmitted packet is deleted from the sentlist. If the transmission fails then the packet is reinserted in the priority list. This makes it possible to have several packets in transit simultaneously.

### Commands

EarthCom must provide the following commands to manipulate the list.

*Insert (priority, data)*

Description: Insert the data into the internally maintained list.

Layout:

| | | |
|---|---|---|
| 0 | priority | packet pointer |

- The first element is the communication command id. (0 = Insert)
- The second element contains the priority of the data.

- The third element is a pointer to the packet.

*Delete (integer)*
Description: Delete the given number of packets from the memory and the internally maintained list, where integer >= 0.
Layout:

| | |
|---|---|
| 1 | number |

- The first element is the communication command id. (1 = Delete)
- The second element is the number of packets to delete.

*RadioReady*
Description: The radio module is ready to transmit to Earth i.e. this module can transmit a packet to the radio module.
Layout:

| |
|---|
| 2 |

- The element is the communication command id. (2 = RadioReady)

*RadioDone*
Description: The radio module has successfully transmitted the packet to Earth i.e. this module can delete the packet from memory.
Layout:

| | |
|---|---|
| 3 | Packet pointer |

- The element is the communication command id. (3 = RadioDone)
- Pointer to the transmitted packet.

**Pseudo code**

```
prioritylist plist;
sentlist slist;
while(true) do {
        mes = wait for command or timeout;
        if (mes = command)
        then
                switch(command)
                case insert(prio, pack):
                        insert the new (prio, pack) into plist;
                        break;
                case delete(n):
                        delete tail(plist) n number of times;
                        break;
                case radioready:
                        if (plist ≠ {})
                        then
                                (prio, ppoint) ← head(plist);
                                insert (prio, ppoint, new timestamp) into slist;
                                start timer;
                                delete (prio, ppoint) from plist;
                                transmit ppoint to radio module;
                                break;
                        else
                                break;
                case radiodone(ppoint):
                        delete (prio, ppoint, timestamp) from slist;
                        break;
                default:
                        Add unknown command error description to plist;
        else //must be timeout
                for all items on slist do {
                        if (prio, ppoint, timestamp) timed out then {
                             insert (prio, ppoint) into plist;
                             delete (prio, ppoint, timestamp) from slist;
                        }
                }
}
```

# The scheduler module

## General description

The idea of the scheduler module is to provide a way to delay commands to the other modules of the system. It might do this by keeping a list of commands ordered by the time they are supposed to be executed. When time has come for the individual commands to be executed, they are sent to their destination and removed from the list. It shall be possible to add new commands to delay, and to delete commands.

## Assumptions

All packets are created in memory only once, and afterwards referred to through pointers. This means that the scheduler module will not need to know the size of the packets scheduled.

## External requirements

It is essential for the scheduling module that there’s access to a reliable consistent clock.

## Proposed solution

### Natural language description

The scheduler module shall maintain a list of associated (time, packet) pairs in some form. When the time described in one such pair is reached or exceeded, the corresponding packet must be sent to the packet router module, and the (time, packet) pair removed from the list. This means that the packet sent to the packet router must itself be a fully functioning packet, containing the destination module and the command including any arguments, as described by the packet router packet definition. The scheduler module must accept commands to insert/delete pair into/from the list in order to maintain the list.

### Commands

Commands accepted by the scheduler module are as follows.

#### *Insert*

Description: Insert a (time, packet) pair into the internally maintained list.

Layout:

| | | |
|---|---|---|
| 0 | time | packet pointer |

- The first element is the scheduler command id. (0 = Insert)
- The second element contains the time at which the packet should be sent to the packet router.
- The third element is a pointer to the packet which further routing is going to be delayed.

#### *Delete*

Description: Delete all (time, packet) pairs from internally maintained list where t1 <= time <= t2.

Layout:

| | | |
|---|---|---|
| 1 | time t1 | time t2 |

- The first element is the scheduler command id. (1 = Delete)
- The second element contains the lower time boundary of the interval that must be deleted.

- The third element contains the upper time boundary of the interval that must be deleted.

**Pseudo code**

```
timepacketlist tpl;
while(true) do {
        mes = wait for command or timeout;
        if (mes = command)
        then
                switch(command)
                case insert(t,p):
                        insert the new (t,p) pair into tpl;
                        break;
                case delete(t1,t2):
                        delete all (time,packet) pairs from tpl where t1 <= time <= t2;
                        break;
                default:
                        send unknown command error description to the earthcom module;
        else // must be timeout
                while((t,p) = first pair from tpl where t < current time) {
                        remove (t,p) from tpl;
                        send p to packet router module;
                }
}
```

# Housekeeping module

## General description

The purpose of the housekeeping module is to provide a simple, efficient way of collecting housekeeping data from all the different modules on the satellite.
All modules must implement a special status command. When this command is invoked, the module generates a packet containing a timestamp and selected status information about the module and the physical subsystem controlled by the module (if any). E.g. the power module could provide information about:

Battery voltage
Battery temperature
Solar panel voltage
Solar panel temperature
Power consumed by different subsystems
Etc.

What information to include in the status packet is entirely up to the modules. There is no requirement on how the information should be encoded in the packet.
The generated packet is sent to the Earthcom module where it is stored until it is transmitted to Earth.

Even though is not necessary for the housekeeping module to be aware of the contents of the status packets generated by the different modules, it might be useful to have some kind of manager module which could react upon the contents of the status packets when the satellite is on its own. E.g. shutdown overheating subsystems, restart non-responding processes etc.
This would, however, require a strict definition of the status packet format, information about the acceptable tolerances for the different monitored values and information about how to react when values exceeds their tolerance levels. An

alternative solution is of course to leave the error handling to the modules them selves.

## Proposed solution

The housekeeping module must maintain a list of records each containing a process ID, logging interval, packet priority and a timestamp. The timestamp denotes the time for the last issued status command.
It is possible to have several logging intervals for the same process ID; this could be used to ensure that we receive housekeeping data from a module with at least some interval and - if there is enough storage space and transmission time available - more often. E.g. you could tell the housekeeping module to request a high priority status packet from the module every hour and to request a low priority status packet every 5 minutes.
The timestamp field is the initialized with the current time, so the first status command is issued at the time of the insert command plus the interval.
As with the scheduler module, this module of course requires access to a system clock.

### Commands

The housekeeping module must provide commands to manipulate the list.

Insert (process ID, logging interval, packet priority)
Delete (process ID, logging interval)

#### *Insert*

Description: Insert a (process ID, logging interval, packet priority) record into the list.
Layout:

| | | | |
|---|---|---|---|
| 0 | Process ID | Interval | Packet priority |

- The first element is the housekeeping command id (0 = Insert).
- The second element contains the process ID of the module one wish to monitor.
- The third element is the logging interval in seconds.
- The fourth element is the priority of the generated status packet.

#### *Delete*

Description: Delete (process ID, logging interval) records from the list, both the process ID and the interval must match those of the record in order to delete the record from the list.
Layout:

| | | |
|---|---|---|
| 1 | Process ID | Interval |

- The first element is the housekeeping command id (1 = Delete).
- The second element contains the process ID of the record that must be deleted.
    - The third element contains the interval if the record that must be deleted.

### Pseudo code

#### *Housekeeping module*

```
housekeepinglist hpl;
while(true) do {
        mes = wait for command or timeout;
        if (mes = command)
        then
                switch(command)
                case insert(pid,int,pri):
                        insert the new (pid, int, pri, current time) record into hkl;
                        break;
                case delete(pid,int):
                        delete (pid,int) record from hkl
                        where pid = process ID and int = logging interval;
                        break;
                default:
                        send unknown command error description to the earthcom module;
        else // must be timeout
                while((pid, int, pri, lasttime) = all records from hkl where
                                                   (lasttime + int) > current time) {
                        send status command to pid;
                        lasttime = current time;
                }
}
```

### *Status command that all modules must implement*

```
while(true) do {
        cmd = wait for command
        switch(cmd)
        case status(pri):
              statdata = collect local status data;
              statpack = create a packet containing timestamp and statdata;
              send a packet containing pri and statpack to the earthcom module;
              break;
        case …
        …
}
```

# Resources and processes

## Budgets

Due to the very limited amount of resources available, it is necessary to determine the resources needed for the different processes, in order to ensure that it is feasible to operate the entire system.
For the onboard-software we consider two key resources: memory consumption and processor usage.
The maximal memory consumption of each process/command and the processor time needed for each command must be determined.
This information is also useful when the command timeline is made.

The processes must be prepared to handle a failed request for more memory.

The verification of the design will be the most difficult part. Using a tool like Uppaal to verify the real-time problem involves an element of uncertainty because there is no way to verify that the graphs are consistent with our design. The state charts can be efficient to verify that there are no deadlocks or to verify that we can recovery from deadlocks.

## Process failure

If a process has crashed in some way, the system must be notified in some way.
One could consider the following alternatives:

- Each process has to signal the Watchdog regularly.
- There is a process that regularly checks all initialized processes to see if they are alive.
- There is a timer on each input buffer so that if a process does not read from the input buffer for a long timer, the timer timeouts. The process that can restart each process is notified by the timer and restarts the process.

It could be discussed whether it is only the process that has to be restarted or it is the whole system. In that situation one have to consider the chances for the process has crashed due to an error that concerns the whole system compared to the chances for a local error.

Maybe there is some functionality in the hardware so that this problem easily could be solved. We are not quiet sure if this problem is of our concern or if there are any other groups for whom it would be more obvious to solve this problem.

## Process priorities

One could imagine that at some point of time a lot of processes would like to be executed at the same time. As the processor only is of a certain size it is not possible to have all the processes running at the same time. It would be very helpful if a process had a priority, so it easily could be determined which processes should be

executed first. We will then have a bit control of which processes are running, in case all the waiting processes cannot be executed.
Each process should be given a priority. These priorities could be defined from the following question (and perhaps more):

How important is this process:
compared to the others?
in order to make the satellite work?
in order to get contact/communicate with the satellite?
in order to recover from some error-state?

Each process' priority must be carefully determined in order to avoid deadlocks. If an high priority process A is depending on a low priority process B that never would be executed because there are to many high priority processes waiting to be executed, it would result in A waiting forever and never terminate.

Perhaps it would be useful to have dynamic priorities. The processes could have different priorities depending on what state the satellite is in. If the satellite is in some kind of error-state one of the processes with the highest priority must be the processes that can make the satellite go to safe mode (boot-strap). If the satellite is in state where everything works the process to communicate to Earth might have a higher priority than the process to go to safe mode.

## Conclusion

Looking at our design it seems that we have made a good estimate of the design and what to do.
And when the outer circumstances – such as choosing the processor, platform and language - have been decided, we feel that most of our unclarified issues are easily solved. I.e. once we have the documentation of the chosen platform, we can see the potentials of the products and which facilities they offer, hence which facilities we have to implement ourselves.
The implementation itself seems to be possible to do in reasonable time if the chosen language will be ADA – offering the communication and multiprogramming facilities we need. However, if the implementation must be done in C it will be much harder to ensure the same level of reliability.
Another issue is memory protection, since the different modules probably will be implemented as processes, it will be a huge advantage if the processor and operating system support memory protection, allowing the processes to have their own isolated memory space.

Another element of uncertainty is the information we need from the hardware groups – will they be able to deliver the information when we need it?
It is still not clarified if the hardware groups themselves have to implement the processes to control their hardware and communicate with the rest of the system or if it is up to us – it might be a problem for the hardware groups to write a program in e.g. ADA.

| | |
|---|---|
| Cecilie M. Larsen | Gitte Marianne Nørgaard |
| Kim G. Jensen | Morten F. Fiisgaard |
| Søren Hjarlvig | Tue B. Jensen |

## Appendix A: Optimistic Time Schedule

**October**
When a processor and a platform are chosen we are ready to study the manuals and product specifications in order to solve some of the unclarified issues, which are depending on these.
Determine which methods and tools we should use to verify our design.
Determine and inform the different hardware groups which information we need and when we need it, in order to finish the design.
Week 42: Midterm holiday.

**November**
Rewrite the design.
Start implementation on test-platform, implement hardware simulation modules.

**December**
Start implementation on the actual hardware.
Week 50-51: Examinations.
Week 52: Christmas holiday.

**January**
Finish implementation.
Test and verification of the solution.
The critical design is to be handed in.
Week 1: Christmas holiday.
Week 2-4: 3-week course.
Week 5: Winter holiday