Faculty of Engineering and Science
Aalborg University

Department of Control Engineering

TITLE:

AAU CubeSat Communication Software

PROJECT PERIOD:
D5,
4. September - 20. December, 2001

PROJECT GROUP:
D5-555

GROUP MEMBERS:
Thorkild Guldager Sørensen
Kim Led Bendtsen
Michael Sig Birkmose
Jakob Nørskov
Frederik Olesen

PROJECT SUPERVISOR:
Anders P. Ravn

COPIES: 8

REPORT PAGES: 132

TOTAL PAGES: 154

SYNOPSIS:

This report describes the development of the communication software for the AAU CubeSat. The communication software is responsible for transferring data from the satellite to the ground station (Telemetry) and for receiving data from the ground station (Telecommand).
The AX.25 data-link protocol widely used in the radio amateur community is used as the basis of the communication protocol. To provide functions such as: Pause, resume, prioritizing etc. a combined session and transport layer, denoted as T55X layer, is analyzed, designed, modeled, implemented and tested.
The development process covers the following subjects: Design of a communication protocol, modeling, validation and verification using SDL in ObejctGEODE and implementation using C programming.

Preface

This report is a part of a fifth semester project. It has been developed by group D5-555 in the year 2001, at Aalborg University. The project is part of a larger project at Aalborg University, called the AAU CubeSat project. The AAU CubeSat project is a project where groups of students from different departments and semesters on Aalborg University develop a small satellite. More information about this project can be found at:

$$http://www.cubesat.auc.dk$$

This report describes the analysis, design, implementation and test of the communication software used on the AAU CubeSat.

References are created based on a number referring to an entry in the literature list, e.g [3].

A CD-ROM is included with this report. It contains source code for the developed software (including test software) and SDL files for the modeling and verification done. This report is also included on the disc in different formats.

The group wishes to thank Brian Lodahl (student at Deparment of Communication Technology at AAU) for his assistance in understanding some of the aspects regarding the hardware considered in this project. Furthermore the group wishes to thank Flemming Hansen (DSRI) for his help on the link budget, hardware etc.

....

Thorkild Guldager Sørensen   Kim Led Bendtsen

....

Michael Sig Birkmose   Frederik Olesen

....

Jakob Nørskov

Contents

Chapter 1

Introduction

The purpose of this project is to analyze, design and implement a protocol for communication between the AAU CubeSat and a ground station. More information about the AAU CubeSat will be presented in part I of this report. The protocol used for the communication is the software part of the communication sub-system (CSS) on the CubeSat. The other distinct part of the subsystem is the hardware: Modems, amplifiers, antennas etc. Building the hardware for the CSS on the AAU CubeSat is not a part of this project, so only a minor analysis of the hardware will be done in this report. The main focus in the design part of this report will be on modeling and verifying the protocol in an appropriate tool. For this project the program ObjectGEODE was chosen, and its syntax and modeling format will be used throughout this report. The implementation part of the project will only cover the implementation of the protocol on the AAU CubeSat, the implementation on the ground station is further described in another report [1].

1.1 Structure of the Report

The first part of this report is a mission analysis for the AAU CubeSat project as a whole. It describes the CubeSat concept in general and the constraints it puts on the design and implementation of the satellite. Furthermore the choice of mission for the AAU CubeSat, and the constraints and requirements it puts on the design, is described. The mission analysis in this report is an edited version of the one made by group 01gr930. The original version can be found on the project CD-ROM as:

/mission_analysis/original_mission_analysis.pdf

The second part of the report contains an analysis, following the analysis part of the SPU development method, described in reference [2]. First an analysis of the communication subsystem will be done, this will lead to a requirement specification for the software. To conclude the analysis more specific requirements will be made, based on the previous analysis and input from other CubeSat project groups.

The third part will be the protocol design following the requirements from the analysis part. The design part will include both modeling, validation and verification of the design, this will as already mentioned be done in ObjectGEODE.

The last part of the report will cover the actual implementation of the protocol to run on the on-board computer on the AAU CubeSat. Since the initial design was made without regard to the actual hardware and operating system on the satellite, some changes may be necessary to make an implementation that works well with the on-board computer on the AAU CubeSat.

Part I

Mission Analysis

This part covers the mission analysis for the AAU CubeSat project. Several topics are considered, first the motivation for developing the AAU CubeSat. Next a mission is chosen for the satellite and the satellite is divided into smaller subsystems that can be developed by the different student groups.

Chapter 2

Mission Description

The CubeSat concept, developed by Stanford University, makes it possible to develop a small satellite and have it launched into space at low cost. The concept specifies among other things, that the satellite must be a $10cm \times 10cm \times 10cm$ cube weighing no more than 1 kilogram. This makes it possible to launch up to three CubeSats at a time, from a standard deployment mechanism known as a P-POD (Poly Pico-satellite Orbital Deployer). In this way considerations regarding launch and deployment from the rocket used as launch vehicle, is indeed reduced for the CubeSat developers. Multiple P-PODs can easily be stacked together and many CubeSats can be deployed in a launch. The cost for launching P-PODs is shared between the CubeSats. The cost is also lowered by launching the P-PODs as secondary payloads in launch vehicles. Secondary payloads are added to launches where a primary payload does not use all the available space in the launch vehicle.
The development of a CubeSat fits ideally to the project based education form used at Aalborg University. This has resulted in a project being initiated which involves groups of students from different departments of AAU.

2.1 Choice of Mission

The Department of control engineering at AAU has already had experience with developing satellites: the operating Ørsted and the not yet finished Rømer. These two satellites fall into the category of micro-satellites. The CubeSat is a pico-satellite, and will be much smaller. The size of a CubeSat gives the advantage of low launch cost, but at the same time it constrains the design. It is a challenge to fit an on-board computer, power supply, communications systems and hardware for attitude control into the satellite. Depending on the mission of the satellite, there must also be room for a payload. First of all the project will give the involved students experience with working on a satellite project.

With this being the first, in a line of possibly more CubeSat projects at AAU, it is part of the mission to bring some experience to AAU in the field of developing CubeSats.

Students working on future CubeSats at AAU may learn from the results of this project. For this to be possible it is important to document all project work and receive health data from the subsystems on the satellite, when it is launched.

In order to secure a high level of reliability in expensive satellites, it is common to use components which have been well tested for use in space. However, this being a low cost satellite, some experimentation is done in the choice of components. This is the case with the payload. The satellite is going to take pictures of the earth from a low earth orbit (LEO) at a height of approximately 600 km. For this a CMOS camera is going to be used. It will take color pictures in the visible light spectrum. The pictures will be transmitted to a ground station and made public via the Internet. People using the Internet will have the possibility to request the satellite to take a photo of a geographic location. The scope of photo coverage will be restricted to Denmark. The purpose of this mission is to increase public interest in space science, technology and natural science in general. The satellite can show that it is possible to monitor or take pictures of the earth from LEO with a small pico-satellite like the CubeSat .

It has also been proposed that the satellite should perform a task with a more scientific purpose. This proposal is observation of stars. By monitoring light intensity of stars, it may be possible to discover unknown planets. The proposal for this mission came too late in the satellite design phase, to consider performing a new mission. Monitoring the stars efficiently would require on-board analysis of data from the camera. However, it has been decided that it would be a good idea to take some pictures of stars, in order to decide if the camera and lens together with the other subsystems in the AAU CubeSat could perform this mission. This could provide a good basis for a later CubeSat designed specifically to monitor the stars.

2.2 Launch and Orbit

The launch of the CubeSats is planned by Stanford University in collaboration with the company One Stop Satellite Solution (OSSS). The launch of the first 18 CubeSats is scheduled to be with a Dnepr-rocket from Baikonur Kosmodrom in Kazakhstan on November 15th 2001. The launch placing the AAU CubeSat in orbit is at moment of writing still being negotiated. The CubeSats being secondary loads, they must wait for the primary load to be ready. For that reason a given launch date may be postponed to wait for the primary load. For this reason a precise launch date or the orbit parameters are not yet known. In Figure 2.1 a Matlab simulation of an orbit with respect to the earth's rotation is shown. Parameters for an orbit very similar to the one the first CubeSats are launched into on November 15th 2001 are used in the simulation, in order to have some idea of what to expect. The orbit is circular and has an inclination of 96°and an orbit height of 600 km. The simulation is made for 24 hours beginning 12.00 o'clock at the day November 15th year 2002. We expect the orbit for our own CubeSat to be something similar to the simulated orbit.

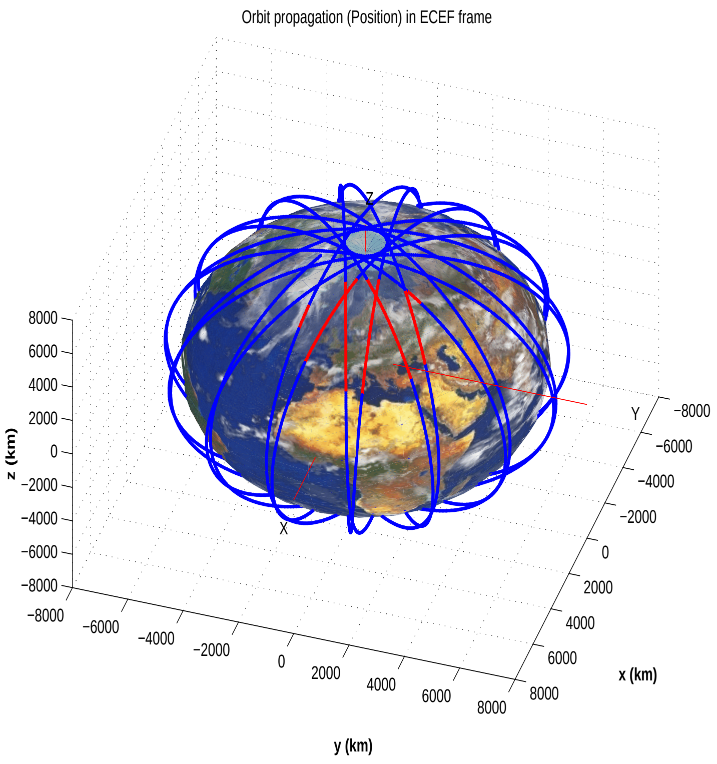


Figure 2.1: Matlab simulation showing how the orbit moves with respect to earth's rotation.

Time	**15th November 2002**				**16th November 2002**		
Start [hour.min.sec]	13.20.10	14.52.20	16.29.50	18.12.40	00.50.20	02.27.10	04.05.30
Stop [hour.min.sec]	13.22.10	15.02.10	16.39.20	18.13.60	00.58.40	02.37.30	04.12.10
Time [sec]	120 sec	590 sec	570 sec	80 sec	500 sec	620 sec	400 sec
Total	Communication over 24 hours: 48 min 0 sec						

Table 2.1: Start and stop times for communication in the simulation shown in Figure 2.1.

2.2.1 Communication Between Satellite and Ground Station

The simulation in Figure 2.1 also shows how often it is possible to communicate with the satellite, when a single ground station at Aalborg University in Denmark is used. The black line showing the orbit, is shaded grey where it passes above 5°over the horizon seen from the ground station. In these periods it will be possible to communicate with the satellite. Every second orbit passing through the area of communication at Figure 2.1, is passing at night time. It is important, when looking at the figure, to remember that it is the earth that rotates approximately 360°per day within the circular orbit.

In Table 2.1 the times where the satellite enters and leaves the range of communication is shown, together with the total time of communication. These times will vary for the satellite depending on the date, but the overall communication time will pretty much be the same for the simulated orbit. With a total of approximately 48 minutes to do all communication including transferring the satellite photo to ground station, this will set a limit to the amount of photos taken per day.

Chapter 3

Mission Analysis

This chapter describes the mission analysis. First the mission analysis describes the mission statement for the satellite. After this the requirements and constraints for the system and for the development of the system are described. The design is described in sections 3.3 and 3.4 and general considerations on hardware implementation in section 3.5.

3.1 Mission Statement

The mission statement is made from the mission description in Chapter 2. The primary purpose and mission with the AAU CubeSat project, is concerning the educational aspect. It is important that the project work fulfills the requirements to project work at AAU. The secondary missions concerns the functionality of the satellite.

Primary mission:
To show that students at AAU are able to design and build a satellite. Hereby also gain experience in designing satellites and pico-satellites in particular.

Secondary mission:

- To communicate with the satellite while it is in orbit, in order to receive house keeping for the on-board subsystems. Hereby gaining knowledge on how well each subsystem works.

- To take satellite photos of the earth and transmit them to a ground station. Hereby testing the attitude control and CMOS camera i LEO.

- To let people via the Internet choose a geographic site to be photographed and then later retrieve the photo from the Internet. Hereby increasing public interest in space science, technology and natural science in general.

- To see if the CubeSat with its configuration of hardware and especially the camera will be able to monitor the stars, in order to find unknown planets. A future CubeSat project considering such a mission could benefit from this knowledge.

3.2 System Requirements and Constraints

In this section requirements and constraints for the system are described. The system requirements must live up to the mission objectives, while the constraints limit the design. The functional requirements define how well the system must perform and the operational requirements determine how the system operates and how users interact with it. Using the CubeSat concept affects both the operational requirements and constraints.

3.3 System Design

Important factors taken into consideration when developing the design for the satellite, listed in random order:

- Keep the design simple, in order not to complicate and increase risk of errors in the system.
- Avoid unnecessary moving parts in design.
- The mission statement described in Section 3.1 must be taken into account.
- There is limited space and total weight must be kept under one kilogram, so this limits the amount of hardware.
- The System must be divided into subsystems with well defined interfaces in order for work to be distributed between project groups.

To carry out the mission chosen for the AAU CubeSat, the direction in which the camera, the antennas and the solar panels are pointed must be controllable. Two design concepts can be considered:

1. Use servo motors to make precise control of pointing direction for camera inside satellite. A simple attitude control is needed to stay in the working area for servo motors. Direction of antennas and solar panels can be controlled with simple attitude control.
2. Antennas, solar panels and camera are fixed inside satellite, and pointed towards their target with precise attitude control.

Design option 2 is preferred over design option 1. In order to keep the design simple and avoid moving parts, the camera, antennas and solar panels will be fixed to the body of the satellite and pointed towards their target using attitude control for the satellite.

Functional requirements	
Performance	Take satellite photo of a 100 × 100 km location in Denmark. When the satellite is right above location, the photo should be no more than 100 km off target. To increase public interest in the photos, they should be high resolution and in colors.
Coverage	The desired daily coverage is one satellite photo of a location in Denmark. However, the coverage or photo quality may be lowered by weather conditions and if the satellite orbit some days does not pass close enough over Denmark.
Responsiveness	Communication is also limited by the amount of times the satellite passes over Denmark. When a photo is acquired in the satellite it should be send back to ground station within a day.
Operational requirements	
Duration	The satellite will take pictures for as long as it is functional
Availability	The satellite is at best within range of communication 10-15 min when passing above horizon seen from the ground station. Every twelve hours it is within communication for short periods approximately 3-4 times. Total communication per day is around 50 minutes.
Survivability	In order to have a reasonable time scope to collect data and experiment with the satellite after launch, the goal is to secure its lifetime in LEO environment to at least a year. This means that it should be able to acquire at least 365 satellite photos in its lifetime.
Data distribution	One ground station located at Aalborg University is used to communicate with the satellite.
Data content, form and format	The photographic material should be compressed and possibly split up into packages before it is send to the ground station. The coordinate for the geographic site is sent together with the compressed photo data. The photo material is then published on a web page.
Constraints	
Cost	Launch: 50000 US Dollars, Satellite: Concept is to design a low cost satellite.
Schedule	1/9-01 Project start, 7/11-01 Preliminary design review, 12/12-01 Critical design review, 15/4-02 Preshipment readiness review, 1/5-02 Delivery of flight model to OSSS, 2/9-02 Shipping from OSSS to Russia, 1/10-02 Launch vehicle integration start, 15/11-02 Launch.
Political	The AAU CubeSat project must fulfill the requirements given for project works at AAU.
Environment	Launch vehicle places satellite into LEO
Interfaces	Operator interface at ground station is to be determined. User interface is a web-site, where a request can be made for a satellite photo of a geographic site in Denmark.
Development	Design must meet the design requirements set forth in the P-POD Payload Planner's Guide.

Table 3.1: Requirements and constraints as suggested by [3, page 15]

3.4 Subsystems

The CubeSat project is divided into the following work packages: The system, the platform and the payload. The System comprises the overall system structure, interfaces on the subsystem level and interfaces with the P-POD. The platform work package contains ground station with control center, spacecraft structure, power supply unit (PSU), command and data handling subsystem (CDHS), communication subsystem (CSS), attitude determination and control subsystem (ADCS). The payload work package comprises of selecting and assembling hardware and determining an interface in order to use the camera. The system work package is developed in close cooperation between the groups involved in the CubeSat project. This results in well defined subsystems and interfaces between the subsystems.

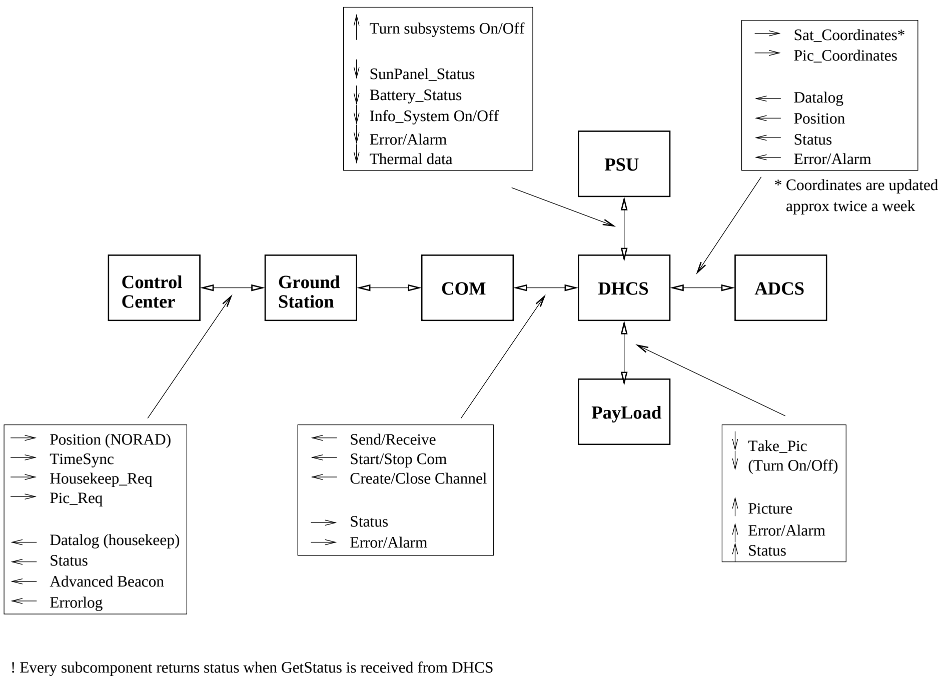


Figure 3.1: Communication between subsystems

3.4.1 Control and Data Handling Subsystem (CDHS)

The CDHS controls the overall functionality of the satellite and takes care of housekeeping in the system. This includes logging of data from subsystems, logging of errors, status of the system, error handling, sends and receives data form the ground and handles the photography.

3.4.2 Attitude Determination and Control Subsystem (ADCS)

This subsystem is needed to control the attitude of the CubeSat during flight. The ADCS is designed and implemented as a fault tolerant control system. With this method the ADCS detects and locates errors in sensors and actuators. It then determines a proper solution to the fault and reconfigures the ADCS to accommodate the fault. Both fault detection, decision on how to accommodate the fault and the reconfiguration is handled within the ADCS.

The ADCS must be able to switch to the attitude control modes, listed in Table 3.4.2. From the CDHS the ADCS receives commands specifying which control mode to use

Attitude control modes	
Detumbling	Used in initialization to stop satellite from spinning
Fail safe	Used in initialization before going to mission mode. It is a very simple attitude control that secures communication between ground station and satellite.
Power save	Simple control to minimize energy consumption and maximize energy input on solar panels. There is a Power save mode for both daylight and eclipse.
Camera control	Attitude control used in mission mode for taking satellite photos. The required precision is decided to be 10°which on ground gives a 100 km error off target. Together with the command to switch to camera control, parameters for the site to be photographed is needed.
Communication control	Used in mission mode to improve communication between ground station and satellite.

and eventually a coordinate for a geographic site to be photographed. If a fault has been detected in the ADCS this is reported to the CDHS together with information about any resulting reconfigurations. Runtime status for the ADCS is also send to the CDHS.

3.4.3 Power Supply Unit (PSU)

The PSU provides, stores, distributes and controls electrical power in the satellite. The power is provided from solar panels mounted on the outside of the CubeSat and it is stored on batteries. Power is distributed to each of the other subsystems through a power-bus. Runtime status for batteries, solar panels and power to the subsystems is send from the PSU to the CDHS.

3.4.4 Communication Subsystem (CSS)

This subsystem makes it possible to receive tele-commands from ground station and send back telemetry. Tele-commands include and send the photo material to ground. Figure 3.2 shows the tele-commands and telemetry.

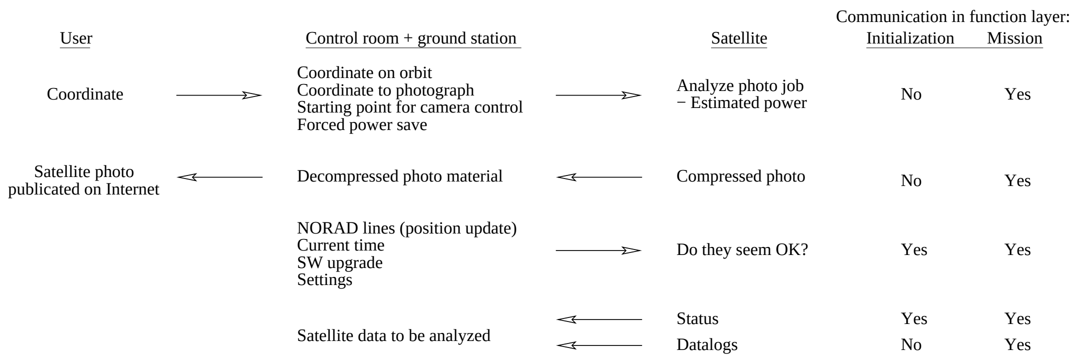


Figure 3.2: This figure shows tele-commands from ground station and telemetry from satellite.

3.4.5 Payload (camera)

The camera is used to acquire the photo material, on a command from the CDHS. The data is stored and ready for being compressed.

3.4.6 Ground Station

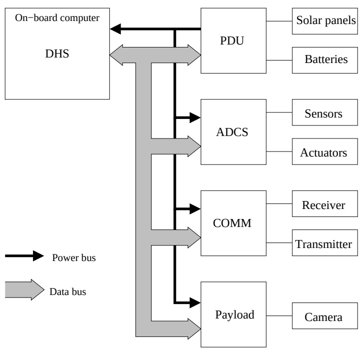
The ground station is equipped with the necessary hardware to send tele-commands to the satellite and receive telemetry. The ground station is connected to a control center. The control center is connected to a public web-page on the Internet. From this web-page a request can be made for a satellite photo centered around a location in Denmark. The control center automatically handles requests from the web-page and sends them to the satellite. When a satellite photo is send to ground it is also automatically published on the web-page. In of case anomalies in the satellite, an operator at the control center must be notified.

3.4.7 Spacecraft Structure

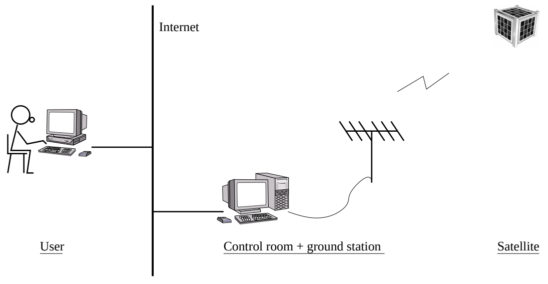
The structural design of the CubeSat, must comply to the design requirements set forth in the P-POD Payload Planner's Guide. Weight distribution and shielding from radiation are some of the criteria to be considered.

3.5 Implementation of System

All subsystems in the satellite are implemented in their own hardware block, as can be seen in Figure 3.3(a). The subsystems in the satellite are communicating over a data-bus, and receive electrical power through a power bus from the PSU. The communication between ground station, control center and user is shown in Figure 3.3(b).



(a) Block diagram describing the implementation of subsystems in the satellite



(b) Block diagram showing hardware on ground

Figure 3.3: The two figures show the complete system divided into satellite and ground hardware.

Chapter 4

Summary

The mission analysis has described the overall considerations for the AAU CubeSat project. The first topic considered was the motivation for developing a satellite at Aalborg University. Since the AAU CubeSat project is a student project, the main goal is to show that student groups at AAU can work together in designing an building a pico satellite. Furthermore it was chosen to fit a small payload into the satellite to carry out a task. Since the CubeSat concept puts very tight constraints on the size and weight of the satellite, the payload for the satellite was chosen to be a small CMOS camera, that will take pictures of Denmark. After the constraints and requirements from the chosen mission and the CubeSat concept in general was analyzed and described, the satellite was split into several subsystems. The rest of this report deals with the communication subsystem, and in particular the software to used in this subsystem.

Part II

Analysis

This part covers the analysis of the communication subsystem, in particular the software in this subsystem. First the hardware is briefly analyzed, followed by a requirement specification for the software. This is followed by more specific requirements, that in detail describes how the defined software should work. Rounding of this part of several application examples will be presented to illustrate how the software is meant to be used

Chapter 5

Communication Subsystem

The communication subsystem (CSS) in the AAU CubeSat project can be considered as two distinct systems, namely the system on the satellite and the system on the ground. This chapter will mainly describe the system on-board the satellite, but some aspects of the ground system will also be covered. The CSS in the AAU CubeSat consist of hardware purchased from One Stop Satellite Solution (OSSS) and the protocol being developed. Section 5.1 will describe the hardware used and the constraints it will put on the system as a whole. The protocol used will be described in Chapters 6 and 7.

5.1 Satellite Hardware

As mentioned, the communication hardware in the AAU CubeSat is purchased from OSSS. They deliver a fully developed and tested solution, consisting of electronic hardware and antenna for the satellite and a modem for the ground station. The remaining hardware for the ground station: Antenna, transmitter and receiver, will have to be purchased separately. A simple sketch of the communication system can be seen in Figure 5.1. The figure shows the different hardware parts in the system and how they interact. All software used for normal communication will be executed on the main CPU on the satellite. In the event that the on-board computer malfunctions or never becomes operational after launch an emergency system should be implemented. This system, also shown in Figure 5.1, will consist of a small micro controller, most likely a PIC, to transmit a simple emergency beacon. When the PIC is powered up it will send out emergency beacons, until the on-board computer signals it to stop over the I2C bus. When not sending out beacons the PIC expects a signal from the on-board computer every minute or else it will start sending out beacons again.

5.1.1 Antenna

The antenna delivered by OSSS is a center loaded dipole antenna with the length tuned to the frequency of the transceiver. The antenna whips are made from spring wire and

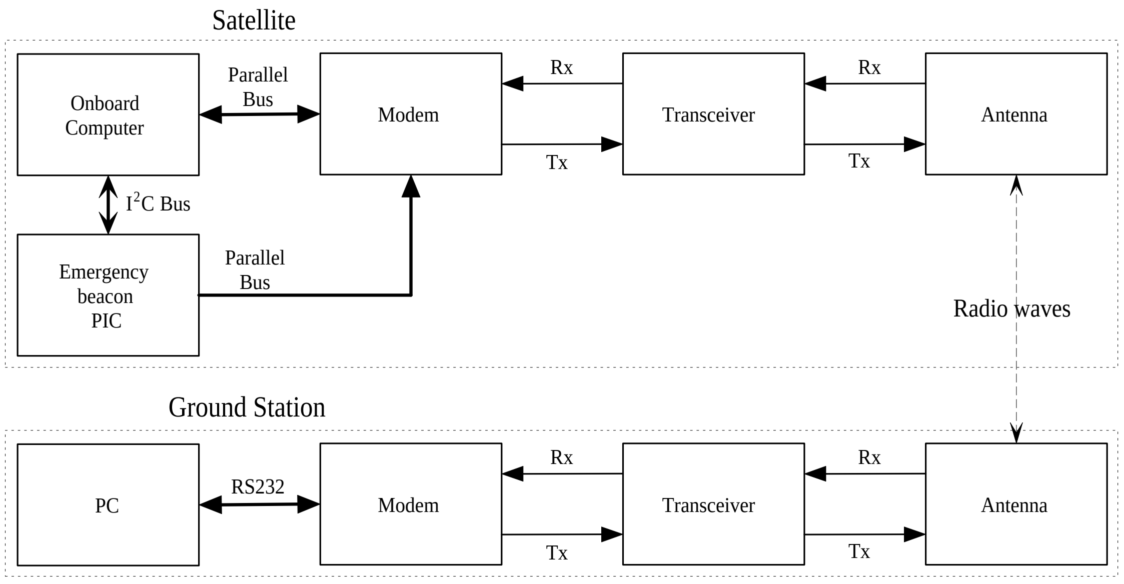


Figure 5.1: Sketch of the communication subsystem

mounted on a delrin ring which is mounted on the outside of the CubeSat. The whips are stowed in grooves in the antenna ring before the CubeSat is deployed into space. The whips are held in the ring by a light string, which is sprung by a heat element controlled by the on-board computer.

5.1.2 Transceiver

The transceiver includes all hardware, filters, amplifiers etc., used to receive and transmit radio signals through the antenna. The hardware operates in half duplex mode, meaning that the transceiver cannot receive and transmit at the same time. The transceivers frequency is programmable in the range from 425MHz to 485MHz. Furthermore the transceiver is only guaranteed to operate with a bit-rate up to 19200 bps.

5.1.3 Modem

A MX909A GMSK (Gaussian minimum shift keying) modem chip made by MX-COM Inc. is used. The MX909A contains all of the baseband signal processing and Medium Access Control (MAC) protocol functions required for high performance wireless packet data communication. The modem is connected, as shown in Figure 5.2, to the main CPU on the satellite using a byte wide parallel bus. The chip is designed with the Mobitex communication system and packet data format in mind, but a custom system is easily implemented. Furthermore it supports the use of (required by Mobitex) Forward Error Correction code (FEC), but this comes at the cost off four extra bits per byte transmitted. Since the link budget (see appendix A) for the AAU CubeSat shows a very low probability of error, the FEC will not be used. Like the transceiver the modem supports a maximum bit-rate of 19200 bps. The bit-rate is set using two capacitors, this means that it can not be changed on the fly. The hardware supplied by OSSS have the bit-rate set to 9600.

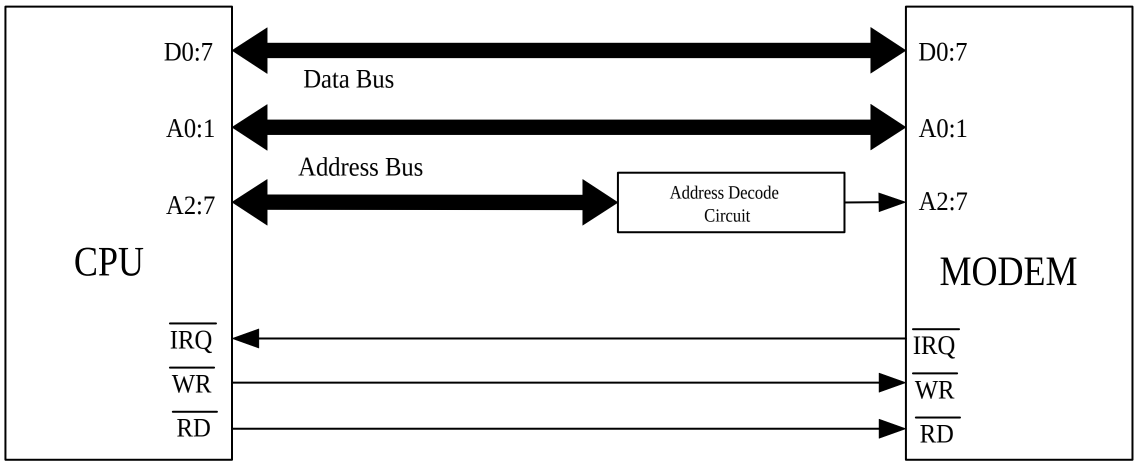


Figure 5.2: CPU<->Modem interconnection.

5.2 Ground Station Hardware

As mentioned earlier the only hardware OSSS provides for the the ground station is a modem, so the remaining hardware will be bought separately. Since the AAU CubeSat orbits the earth in a LEO (Low Earth Orbit) approximately 600 km above ground and the transmitter on the satellite transmits with two watts in the s-band, a simple yagi antenna on a rotor should be sufficient for the ground station. The transceiver used will be chosen from at large variety of amateur radio equipment available. The actual decision of antenna and transceiver is not within the scope of this report and will not be considered further.

Chapter 6

Requirement Specification

This chapter will describe the requirements to the communication protocol. Different topics and sources have to be considered for the requirements e.g. requirements from the AAU CubeSat mission, the hardware used and requests from the other groups working on the AAU CubeSat project.

6.1 System Description

The system being developed is a protocol to handle communication between the CubeSat and the ground station. The protocol have to fulfill the following requirements:

- Provide a flawless link for communication.
- Compatible with the call names used in the radio amateur community.
- Possibility to pause and resume the communication, when exiting and entering sight of communication.
- Possibility to prioritize between different types of data.
- Support multiple "channels" - similar to the port concept in TCP.
- Operate on a half duplex link, which yields the following requirements to the protocol:
  - Supported in MAC (Media Access Control) layer.
  - Prioritizing between satellite and ground station.
- Possibility to transmit a beacon.

6.2 Functionality

An already existing data-link protocol, AX.25, has been found and is suitable as a basis for the new protocol. A detailed description of the AX.25 protocol is found in Appendix B. The AX.25 protocol has been chosen for the following reasons:

- It handles error detection and retransmission of corrupt data.
- It supports call names.
- It supports half duplex.
- It is widely used in the radio amateur community.
- It is well documented.

As seen in Appendix B AX.25 supports both connection-oriented and connection-less transmissions. The new protocol will use the connection-oriented part of AX.25 except for transmission of the satellite beacon.

The main drawback with AX.25 is its lack of support for priorities. Furthermore its capabilities to pause and resume communication are not sufficient. When the AX.25 looses its connection all data waiting in its transmit buffer are flushed, this means that transmission of a data block over several passes is not supported by AX.25. An appropriate solution to these problems will have to be developed.

Following the OSI model, described in Appendix D, the AX.25 is the data-link layer (second layer), prioritizing and pause/resume functions will have to be handled in higher layers. According to the OSI model the desired functionality requires both a transport and session layer. In this project it have been chosen not design two independent layers, but to incorporated all the functionality into one layer called the T55X layer. The overhead in design and implementing two independent layers was deemed to large. Furthermore the protocol being developed the functionalities required from the OSI session and transport layer are easily combined in one layer.

A complete block diagram of the protocol stack is sketched in Figure 6.1.

For the applications using the protocol it will seem as a transparent link to the ground station, with the ability to send different types of data with different priorities.

Not all layers from the OSI model are necessary, e.g. a network layer is not needed since the protocol will only be used between the ground station and the satellite.

6.2.1 T55X Layer Functionality

The T55X layer will include functionality to handle prioritizing and pausing/resuming communication. Furthermore multiple channels will be available to a higher layer, these

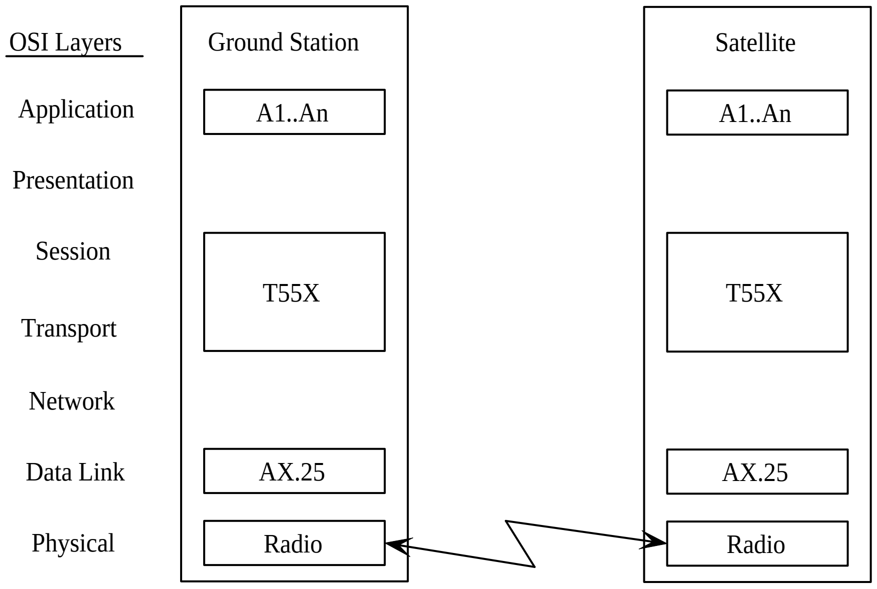


Figure 6.1: Layer structure of the protocol

are used to transmit different types of data. An overview of the main parts of the T55X layer can be seen in Figure 6.2.

To solve the problem with pausing and resuming communication, the T55X layer have to segment large data blocks into smaller packages to pass to the data-link layer (AX.25). If a package is lost in the data-link layer when connection is lost, it will be retransmitted by the T55X layer when the connection is reestablished. The segmenting is done individually on the different channels. Data segments received on a channel will be stored until the complete data block have been received, only then will data be send to a higher layer.

The T55X Transport Controller (TTC) shown in Figure 6.2 have several different tasks. Its main function is to prioritize between the different channels and always transmit the most important data. Since only half duplex communication hardware is available it needs to prioritize between the satellite and ground station. This function is also

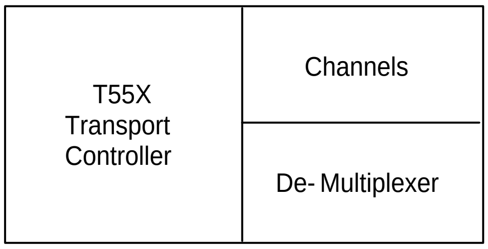


Figure 6.2: Overview of the T55X layer.

included in the TTC. Finally the establishment of a communication connection will also involve the TTC. When a higher layer request a connection, the TTC will handle all communication with the AX.25 required to establish the appropriate connection.

The last part of T55X is the De-multiplexer shown in Figure 6.2. This block is closely tied with the segmenting done in the channels when data are transmitted. When a data block is received from AX.25 the de-multiplexer sends the block to the correct channel or to the TTC if the data block received is internal control data.

In Figure 6.3 a s MSC (Message Sequence Chart) illustrates a simple usage of the protocol. The assumptions for this MSC are:

- Communication has been established.
- Channel 1 has the highest priority and its data is send in one packet.
- Channel 2 has the lowest priority and its data is send in two packages.

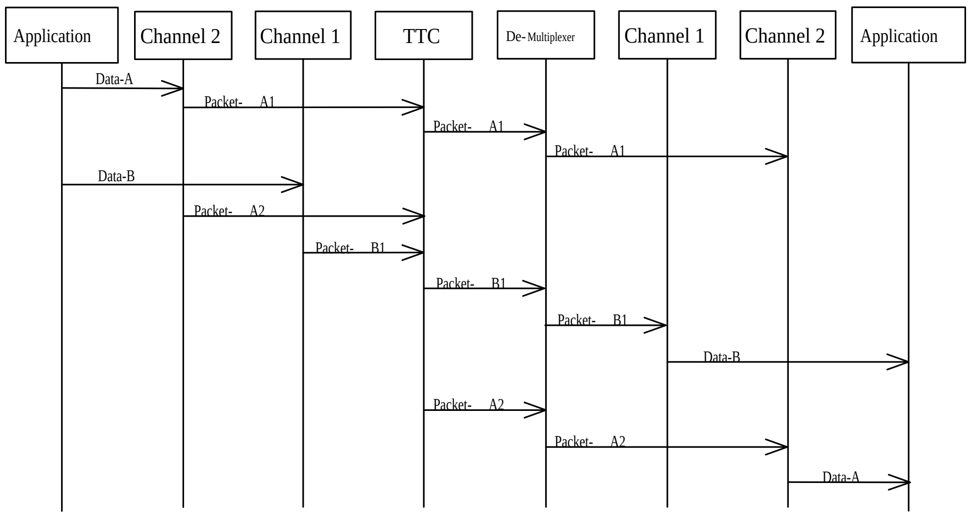


Figure 6.3: MSC of the T55X layer.

Figure 6.3 shows how the channels segment the data into packages and are allowed to send them, by the TTC if they have the highest priority. It also shows that the application is not informed about any packages before all packages have been received.

Chapter 7

Specific Requirements

In this chapter the topics: Definitions and functionality, are covered in order to achieve the specific requirements for the communication subsystem interface to the overlying applications. All the specific requirements are derived from a meeting with the groups responsible for the subsystem utilizing the protocol.

7.1 Definitions

The purpose of this section is to define the data structures involved in the interface to the overlying applications.

7.1.1 Data

The data sent between the protocol and the interfacing subsystem is represented as a pointer and a length of the data in bytes. The size of the length parameter is set to 32 bits, for connection oriented data, which allows for data sizes up to 4 GB. The alternative would be 16 bits which only allows for 64 KB, and the size of the largest data to be sent (images) is at least 100 KB.

For connection-less data the maximum length is 256 bytes, hence beacons has to be within this limit.

7.1.2 T55X Channel ID

This data structure identifies a virtual channel in the transport layer of the protocol stack. The requirement from the subsystems using the protocol is that 5-7 channels should be available. To accommodate for future extensions the range of the ID is set to 0-15, which allows for 16 unique channels.

7.1.3 T55X Channel Priority

In order to prioritize the data sent on the space link several options are available:

1. Use the ID as the priority as well, so that each channel has a unique priority, where the lowest ID has the highest priority.
2. Divide the channels into priority groups so that for instance channels 4 to 7 has the same priority and channels 8 to 11 have the same priority.
3. Keep the ID and priority independent of each other, so that the ID says nothing about the priority. Prioritizing would require that every data sent on the channel should have a priority of its own.

From these possibilities the first one, is chosen, because it is sufficient for the interfacing sub systems. This means that there will be no explicit data structure to identify the priority, it will appear from the ID.

7.1.4 Transmit Buffer

Each channel has a buffer to hold the data sets (pointers and lengths) to be send on it. The buffer is a dynamic list (per channel) that expands as more data sets are send. To prevent this list to expand indefinitely (beyond memory limits), a function that returns the size of this buffer will be implemented. This can then be used to regulate the buffer size from application level.

The actual data will be stored at the given pointer, with the length indicating its size. When the buffer is sent the memory block is freed, and will return to the memory heap for use by other applications.

7.1.5 Call-back Functions

To notify the applications of sporadic events, such as when a complete data block has been received in a channel or when errors occur, a call-back function is invoked to notify the application about it. The name of the call-back functions are not specified in the protocol, only the structures of them are, and it is the responsibility of the application to set up appropriate call-back functions that conform to the structure, and handles the events.

There are two different types of call-backs:

1. Channel call-back.
   To handle channel events like data received.
2. General call-back.
   Handles system events like disconnect.

General Call-back

There is only one instance of this call-back function and it is used to notify the application of events related to the entire communication protocol, and to receive connection-less data (beacon).

The structure of this call-back function is:

```
void (*func)(char type,char* startPointer,int length)
```

The different values of the type parameter are:

- `DATA_RECEIVED` - A complete connection-less data block (beacon) has been received and its location is stored in the pointer and length parameter.
- `CONNECTED` - The communication system is connected to the other part (satellite / ground station)
- `DISCONNECTED` - This event is triggered either if the link is broken (out of reach) or the other part explicitly disconnected.
- `OUT_OF_MEMORY` - If one of the channels fails to allocate memory in its receive buffer this event is triggered.

An example of this type of call-back function is:

```
void handleEvents(char type,char* startPointer,int length)
```

Channel Call-back

This call-back is unique for each channel and it is called whenever a complete data block has been received at the channel. The structure of this call-back is:

```
void (*func)(char* startPointer,int length)
```

The call-back function must be a void function with the received data (pointer and length) as the parameters.

An example of this type of call-back function is:

```
void receiveImage(char* startPointer,int length)
```

7.2 Functions

This section describes the specific functions in the communication sub system available to the overlying applications. The functions are divided into three categories: System functions, general functions and functions related to a channel.

7.2.1 System Functions

Initialize Communication

- **Input:** General call-back.
- **Function:** Closes all open channels and sets up the communication with the specified general call-back function. It is up to the applications to inform the other part (satellite / ground station) that this function has been called.
- **Output:** -
- **Syntax:**

```
void initCommunication((*func)(char type, char* startPointer,
                       int length));
```

7.2.2 General Functions

Connect

- **Input:** -
- **Function:** Sends a connect request to the data-link protocol and will wait for a response and return the result.
- **Output:** The function returns a value that indicates the result of connection request:
  - `CONNECTION_OK`: Connection established.
  - `CONNECTION_FAILED`: Connection failed, due to time-out.
- **Syntax:**

```
char connect();
```

Disconnect

- **Input:** -
- **Function:** Closes all open channels and sends a disconnect request to the data-link protocol.
- **Output:** -
- **Syntax:**

```
void disconnect();
```

Pause

- **Input:** -
- **Function:** Will disconnect at data-link level, but will keep the data to be send and received.
- **Output:** -
- **Syntax:**

```
void pause();
```

Resume

- **Input:** -
- **Function:** Connects the data-link level protocol after a pause.
- **Output:** -
- **Syntax:**

```
void resume();
```

Send Connection-less Data

- **Input:** Start and length.
- **Function:** Sends the data, specified in the start and length parameter, connection-less.
- **Output:**
  - `DATA_SEND`: Data successfully scheduled for transfer.
  - `DATA_NOT_SEND`: Data scheduling failed, e.g. `length`>256.
- **Syntax:**

```
char sendConLess(char* start, int length);
```

Get Status

- **Input:** -
- **Function:** It will read the status of the communication system and return it.
- **Output:** Can return the following values:
  - `COM_RUNNING`: Communication is established and running.
  - `COM_PAUSED`: Communication is established but is currently paused.
  - `COM_STOPPED`: Communication is not established and is not running.
- **Syntax:**

```
char getStatus();
```

7.2.3 Channel Functions

The functions related to a channel are:

Open Channel

- **Input:** ID, channel call-back and number of elements in the transmit buffer.
- **Function:** Creates a new channel with the specified ID, transmit buffer and call-back function.
- **Output:** The function returns a value that indicates the result of it:
  - `CHANNEL_OPENED`: Channel opened successfully.
  - `CHANNEL_ALREADY_OPENED`: Channel is already open.
  - `CHANNEL_NOT_OPENED`: Channel could not be opened, e.g due to lag of memory.
- **Syntax:**

```
char openChannel(char id,void (*func)(char* data,int length) callBack);
```

Close Channel

- **Input:** ID.
- **Function:** Removes the channel with the specified ID, and cleans up the channels memory usage.

- **Output:** -.
- **Syntax:**

```
void closeChannel(char id);
```

Send Data

- **Input:** ID, start and length.
- **Function:** Puts the start and length of the data at the next element of the transmit buffer. The data will then be send when scheduled to by the TCC in the T55X transport layer. If there is no memory left for another element in the list, an error is produced.
- **Output:**
  - `DATA_SEND`: Data successfully scheduled for transfer.
  - `DATA_NOT_SEND`: Data scheduling failed, e.g. no memory for buffer.
- **Syntax:**

```
char sendData(char id, char* start, int length);
```

Data Left to be Send

- **Input:** ID.
- **Function:** Subtracts the length of data being sent from the data that has already been sent.
- **Output:** Returns the amount of bytes that have not yet been transmitted.
- **Syntax:**

```
int leftToTransmit(char id);
```

Elements in Transmit Buffer

- **Input:** ID.
- **Function:** The function will return the number of elements in the transmit buffer of the channel with the specified ID.
- **Output:** Elements in transmit buffer.
- **Syntax:**

```
int transBufferElements(char id);
```

Flush Transmit Buffer

- **Input:** ID.
- **Function:** Clears all elements in the transmit buffer.
- **Output:** -
- **Syntax:**

```
void flushTransmitBuffer(char id);
```

Flush first element in Transmit Buffer

- **Input:** ID.
- **Function:** Clears the element currently being transmitted.
- **Output:** -
- **Syntax:**

```
void flushCurrentTransmit(char id);
```

Data in Receive Buffer

- **Input:** ID.
- **Function:** Returns the amount of bytes in the receive buffer.
- **Output:** Amount of data in the receive buffer.
- **Syntax:**

```
int dataInReceiveBuffer(char id);
```

Chapter 8

Application Examples

This section describes how the previously described functions could be applied to achieve the overall functions of the communication system.

8.1 Start Communication

This function is only used when the satellite does not know its orientation and needs to ping the ground station. To start the communication the following pseudo code could be used:

Satellite:

```
function beaconMode(){
  while(getStatus != CONNECTED){
    sendConLess(data); //Send out beacon with e.g. housekeeping data
    sleep(RTT+T); //sleep Round Trip Time + computation time
}
```

Ground station:

```
//Initialize the general callback function.
initCommunication(handleEvent);

//The general callback function
function handleEvent(type,ptr,length){
  if (type == DATA_RECEIVED ) then {
    store(ptr,length);
    connect(); //Blocking function call
  }
}
```

The pseudo code is illustrated with a MSC in Figure 8.1. The initiating side is the satellite which sends out a beacon. When a beacon gets through to the ground station it answers the beacon with a connect request. This connect request blocks until a connection has been established or it times out. If a connection has been established the loop on the satellite breaks and a connection has been established.

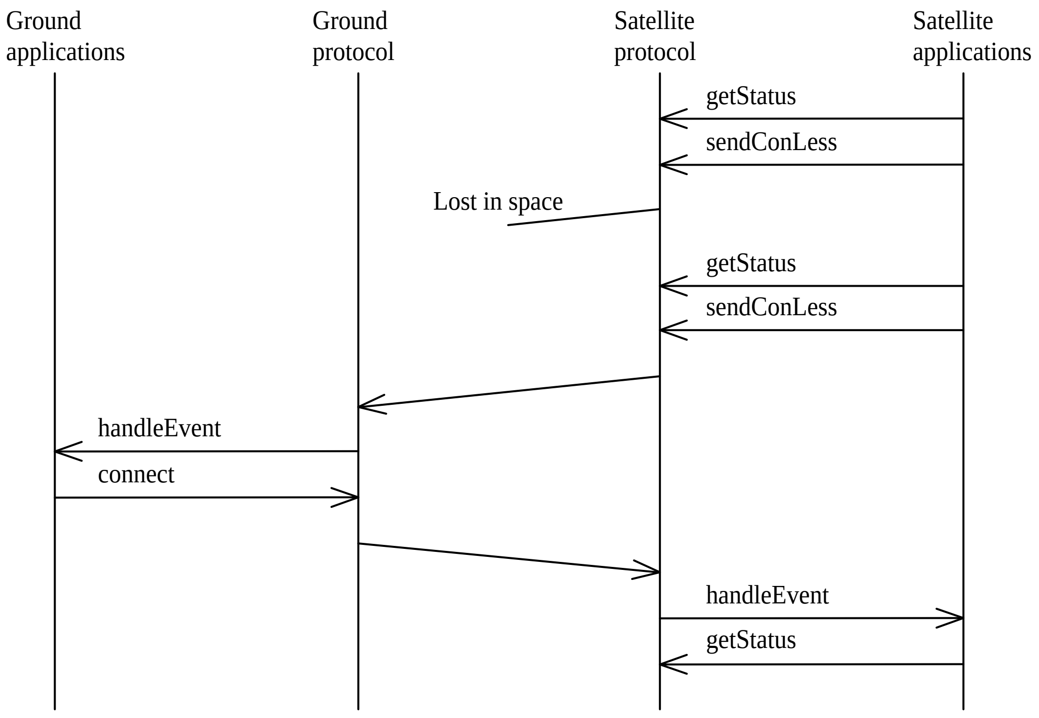


Figure 8.1: A MSC showing how the satellite connects.

If the transmission is to be paused, the `pause()` function should be used. This will halt the transmission, until the connection is reestablished by the `resume()` function. To reset all transmissions the `disconnect()` function must be used. After calling a `disconnect()` it is necessary to call the `connect()` function again.

8.2 Send / Receive Data

This is an example on how to send and receive data between the ground station and the satellite.

Ground station:

```
openChannel(3,callbackFunction);
sendData(3,data); //sends data to sendbuffer
sendData(3,data2); //sends data to sendbuffer
}
```

Satellite:

```
openChannel(3,receiveData);

function receiveData(start,length){
  store(ptr,length)  //Save the incomming data
}
```

On both the ground station and the satellite, channels with the same ID and priority are opened. In this example the call-back function on the ground station is not defined because it will not receive any data. On the satellite a call-back is registered as `receiveData` where the data will be received. When the complete amount of data in `data` is received at the satellite the call-back function is called and the application can store the data. The same thing happens when `data2` is received at the satellite.

8.3 Time Synchronization

The normal approach to time synchronization is a hardware solution that requires the possibility to time stamp a package when the first bit is transmitted. On the AAU CubeSat this solution is not possible, so a software approach will be used instead. This solution is described in the following.

8.3.1 Synchronization Description

To synchronize the timers on both the satellite and the ground station, a method to compensate for slow link must be used. This method will try to compensate by measuring the round trip time for the packages, and does therefore also require that the round trip times to be as uniform as possible.

The algorithm (also illustrated in Figure 8.2) would follow this sequence:

1. Ground Station sends its current time (T0).
2. Satellite updates its clock with the received time (T0).
3. Satellite returns the same time (T0) back to the ground station.
4. Ground station gets its current time (T1) and subtracts the received time (T0). This would be the round trip time (rtt), and the transmission time (dT) would be half this value.
5. Ground station sends current time added with the transmission time (T1+dT).
6. Satellite subtracts the received time (T1+dT) with its current time (T0). Satellite now knows round trip time (rtt), as well as current time (T1+dT) and updates its clock accordingly.

7. Satellite returns its current time (T1) added with the round trip time (rtt).
8. Ground station verifies that the received value matches it current time (T1+rtt = T2), if not it continues from point 4, otherwise it sends an OK message to the satellite, indicating end of transmission.

A reasonable tolerance in error should be used here, since variance in trip times will affect the accuracy of this synchronization method.

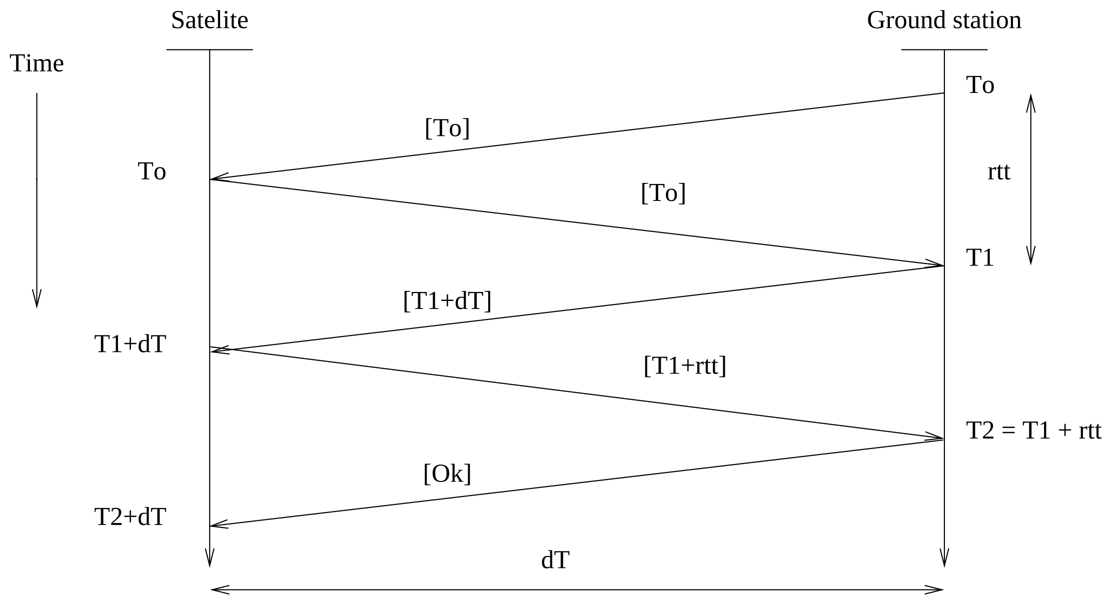


Figure 8.2: Time synchronization message exchanging

Chapter 9

Summary

This part have outlined the major considerations behind the communication protocol for the AAU CubeSat.

First the hardware in the communication subsystem on the satellite was described, which yields this first basic requirements for the protocol. This being the ability to work with half duplex radio hardware.

Next a requirement specification was made, based on the input from other CubeSat project groups at the university. The main functionality of the protocol was chosen and the protocol stack described. An already existing data-link protocol, AX.25, was chosen as base for the new protocol. AX.25 was chosen because it provides the basic functionality required for the AAU CubeSat, furthermore it is well documented and widely used by the radio community and several other CubeSat projects around the world. With AX.25 as a base an extra layer, T55X, was needed to fulfill the requirements set by the AAU CubeSat project.

After the general outline of the T55X layer, more specific requirements for the layer was chosen based on the input from the other projects groups that will interface with the protocol. All functions chosen and made available to overlying applications was described in detail.

This part was rounded of with application examples to show how the protocol should be used.

During all these tasks the main objective was to define the interface to the other subsystem, since this was the biggest problem at the beginning of the project.

Part III

Design

This part covers the design of the T55X layer. The T55X layer will be modeled in SDL using ObjectGEODE. First the internal and external signals in the model are described, next the main functionality will be described. After the model is completed, test specification for simulation, verification and validation will be presented. This part ends with the actual result of the simulation, verification and validation.

Chapter 10

T55X layer

This chapter describes the design of the T55X layer. The layer will be developed with the requirements from the analysis part in mind, but not the constraints given by the on-board computer and operating system on the satellite, as these considerations belongs to the implementation part. This chapter gives an overview of the T55X, followed by descriptions of the functionalities in the T55X layer and the different state machines. As mentioned previously it has been chosen to use the SDL language to describe the T55X layer and ObjectGEODE to model and verify the design.

10.1 Overview of T55X Layer

This section provides an overview of the T55X layer as it is derived from Chapter 7, which defines the different tasks and interfaces of the T55X layer. Figure 10.1 shows the SDL model of the layer. This model includes the different state machines described in section 6.2, as well as the signal flow in the layer.

10.1.1 T55X Signals

Several signals are present in the T55X layer. Some of these are used for internal communication in the T55X layer, while others are used to interface with the application layer and the data-link layer. An overview of these signals can be seen in Figure 10.1

Application Layer Signals

The functions defined in section 6.2 are used as signals between the application layer and the T55X layer with a few simplifications. The simplifications are due to the fact that this is a design with verification in mind. Application functions whose only purpose is to pass status informations are left out. Also functions which can be composed of other functions will be left out. Furthermore the transmit buffer is reduced to a size of one, which removes some of the flush functions. Finally the resume connection function has been removed as it turned out to do the exact same as the connect function.

The functions omitted from the design are therefore:

- Left To Transmit
- Elements in Transmit Buffer
- Flush Entire Transmit Buffer
- Flush the First Element in Transmit Buffer
- Data in Receive Buffer
- Resume connection

The remaining signals are shown in Figure 10.1 and will be explained. One of the main differences between SDL signals and function calls, is that signals return no value.

- `init` - This signal is used to initialize the communication system on one side.
- `sig_receiveEvent` - This signal indicates an event to the application. The signal is sent with a parameter indicating which event occurred. The structure of this signal can be seen in Table 10.1. The beacon received is considered to be an event with the data put in the connection-less data field.

Length:	8 bit	0 bit to 256 bytes
Carries:	Event type	Connection-less data

Table 10.1: Structure of a `sig_receiveEvent` signal.

  The event type can have the following values:

  - Running - To indicate that the connection is established and that this side has the token, meaning that this side is allowed to send data, while the other side is not.
  - Waiting - To indicate that connection is established and the other side has the token.
  - Disconnected - Indicating that the two sides are currently disconnected.
  - IncommingData - Which will be accompanied with connection-less data.

- `connectTo` - This signal is used to establish a connection between the two T55X layers.

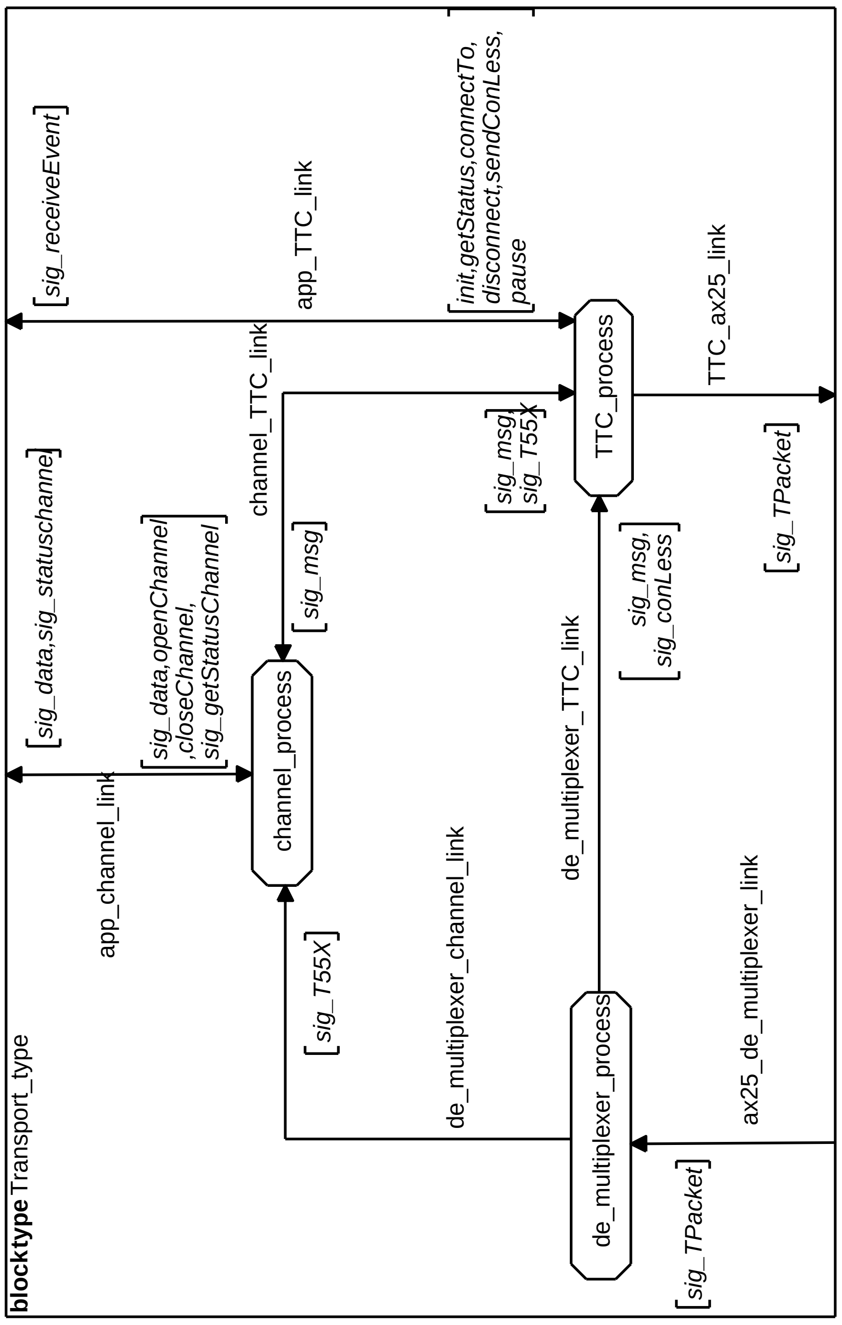


Figure 10.1: An overview of the T55X layer.

- `disconnect` - This signal is used to close a connection between two T55X layers and it will flush all data currently present in all channels on both sides.
- `pause` - This signal is used to close the connection between two T55X layers. Data transmissions a resumed when connection is reestablished.
- `sendConLess` - This signal carries data that is sent without a connection.
- `sig_data` - This signal sends data to and from an application. The signal provides a channel id and the data received.
- `getStatus` - This will tell the TTC to send a `sig_receiveEvent` to the application, indicating the current status.
- `sig_getStatusChannel` - Enquires the TTC about the status of a given channel.
- `sig_statusChannel` - This signal is a respond to `sig_getStatusChannel` and indicates whether a channel is open og closed.

Data-link Layer Signals

The only signal used on the interface to the data-link layer is `sig_TPacket`. The structure of this signal can be seen in Table 10.2. The `info` field specifies a primitive and a primitive type. Table 10.3 shows all primitives used along with the primitive types which are explained below:

Length:	8 bit	0 bit - 4 gigabyte
Carries:	Info field	Data

Table 10.2: Structure of a `TPacket` signal.

- **request** - Used when sending a packet to AX.25.
- **indication** - Used when AX.25 has a packet to deliver.
- **confirm** - A confirmation that a packet was correctly received on the other side.

The `data` field can hold parameters for a primitive, for instance a `DL_ERROR` uses the `data` field for an error number. The `DL_DATA` uses the `data` field for a T55X Packet.

Internal signals

The internal signals will be described in further detail in section 10.2. Here the frame structure for the signal parameters between processes will be discussed.

Primitive name	Possibilities
DL_CONNECT	request, indication, confirm
DL_DISCONNECT	request, indication, confirm
DL_DATA	request, indication
DL_UNIT_DATA	request, indication
DL_ERROR	indication

Table 10.3: Primitives and possible primitive types.

- `sig_T55X` - This signal indicates a T55X data packet. The structure of this packet is shown in Table 10.4. The `packet status` field is used when establishing a connection and negotiation is needed. The `packet type` field gives the possibility to classify different packet types e.g. start, stop and command packets. The `channel id` field indicates the channel on which the data is sent. The `type info` field contains different kinds of data, depending on the `packet type`.

Length:	4 bit	4 bit	8 bit	24 bit	<= 16 megabytes
Carries	Packet status	Packet type	Channel id	Type info	Data

Table 10.4: Structure of a T55X packet.

- `sig_msg` - This signal carries a message between the processes, the structure of the signal is shown in Table 10.5. Its main purpose is sending commands.

Length:	8 bit	8 bit
Carries:	Command	Id

Table 10.5: Structure of a message signal.

The possible commands in the `command` field are:

  – ChannelGotData - Sent from the channel state machine to the TTC when the channel has received data from an application.
  – SendData - Sent to the channel when it should send data to the TTC.
  – ChannelIsClosed - Sent to the TTC, when a channel has been closed.
  – KeepOnSending - Command sent to the other side, indicating that it can send its next data.
  – PriorityData - Arrives in the TTC when data arrives on a channel.
  – ConnectIndication - Arrives on both sides upon sending a connect request.
  – DisconnectIndication - Arrives, like ConnectIndication, on both sides upon sending a disconnect request.

  - NoMoreData - Used to send to the other side, when there is no more data on this side.
  - NoMoreDataOnChannel - Indicates that a channel has no more data to send. Sent immediately after the last data has been sent from a channel.
  - FlushBuffer - Tells the channel state machine to flush data on all channels.
  - GetToken - Used when both sides has no more data to send, and the last side informed of this suddenly receives data. Avoids that the two sides sends data simultaneously.
  - Retransmit - Tells the other side that the last package had errors and must be retransmitted.
  - LastPacketDiscarded - Sent to the TTC if the de-multiplexer receives a package, which it has all ready received before.
  - ErrorIndication - Received if a received package has errors.

- `sig_conLess` - This signal carries connection-less data between the de-multiplexer and the TTC.

10.2 Functionality

In this section the functionality in the T55X is described in detail.

It may be a good idea to consult the SDL figures on the project CD-ROM in path "design/Sdl/", to understand in further detail, what is described in this section.

The main functionalities in the T55X are:

- Send beacon
- Connect
- Disconnect
- Pause
- Send data
- Negotiate
- Error handling

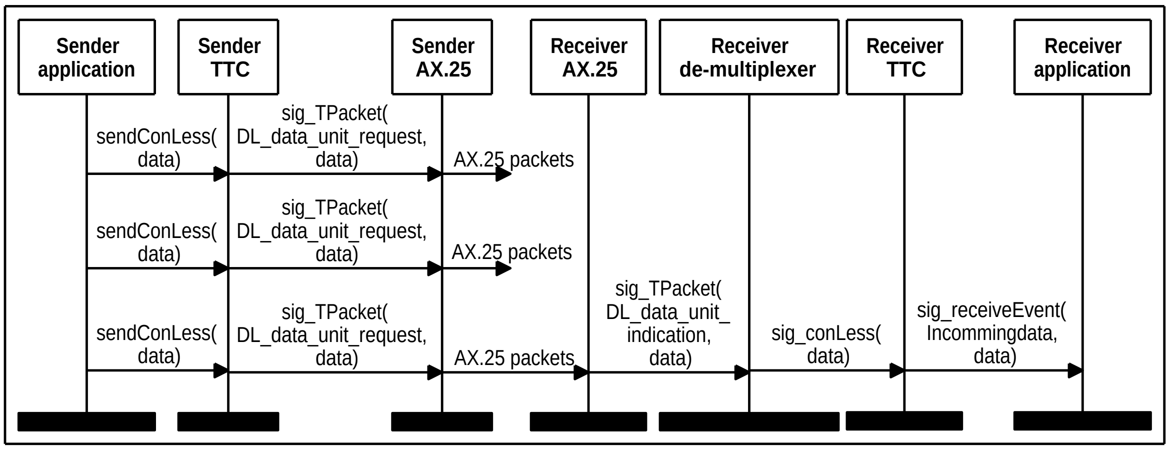


Figure 10.2: MSC showing signals involved in sending a beacon.

10.2.1 Send Beacon

When no communication is established, it should be possible to send a beacon with a certain interval. The size of this interval and when to send a beacon will be controlled by an application.

Figure 10.2 shows when signals for the "send beacon functionality" are sent.

The application will send a `sendConLess` signal to the TTC containing the beacon data. The TTC will then create a `TPacket` signal with the beacon data and a `DL_DATA_UNIT_REQUEST` primitive. This primitive is used in AX.25 to send connection-less data, which means that there is no guarantee that the data will arrive on the other side.

If the receiving side picks up a beacon, it will be delivered as a `TPacket` in the de-multiplexer. This will then send a `sig_conLess` to the TTC, which will send the beacon data to an application via the signal `sig_receiveEvent`.

10.2.2 Connect

When the two sides are within range of communication, one of the sides might want to connect, either while it receives a beacon or while it has calculated that the satellite is within range of communication. It sends a `connectTo` signal to the TTC. The TTC will then send a `TPacket` to AX.25 with a `DL_CONNECT_REQUEST` primitive.

Figure 10.3 shows when the different signals involved in the "connecting" functionality are sent.

If the connect request is successful, the receiving side receives a `DL_CONNECT_INDICATION`, while the sending side receives a `DL_CONNECT_CONFIRM`. These primitives will be sent from the de-multiplexer to the TTC, which will then set the receiving side in *waiting* state and the sending side in *running* state enabling this side to send data. The TTC's will furthermore inform applications on both sides that the connection has been

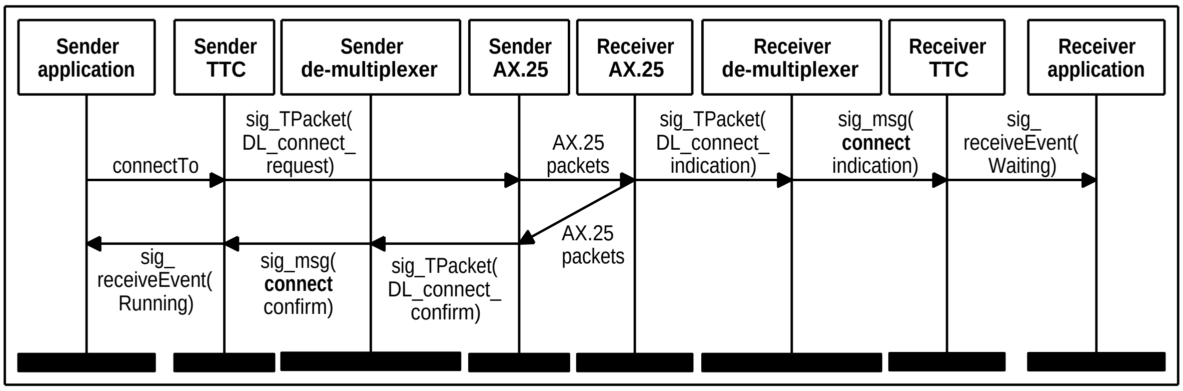

Sender application
Sender TTC
Sender de-multiplexer
Sender AX.25
Receiver AX.25
Receiver de-multiplexer
Receiver TTC
Receiver application
connectTo
sig_TPacket(DL_connect_ request)
AX.25 packets
sig_TPacket(DL_connect_ indication)
sig_msg(connect indication)
sig_ receiveEvent(Waiting)
AX.25 packets
sig_ receiveEvent(Running)
sig_msg(connect confirm)
sig_TPacket(DL_connect_ confirm)

Figure 10.3: MSC showing signals involved in connecting.

established by sending the state of the side via a `sig_receiveEvent`.

10.2.3 Disconnect

Figure 10.4 shows when the different signals involved in the "disconnecting" functionality are sent. The pause signal in the figure indicates, that there is no difference between the last signals in disconnecting and in pausing.

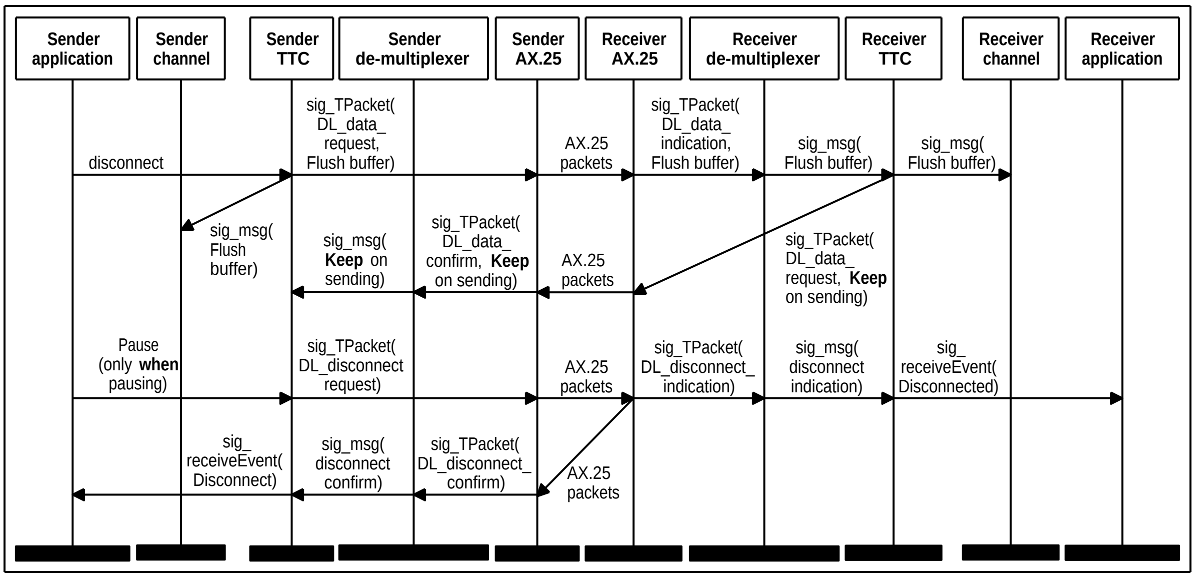

Sender application
Sender channel
Sender TTC
Sender de-multiplexer
Sender AX.25
Receiver AX.25
Receiver de-multiplexer
Receiver TTC
Receiver channel
Receiver application
disconnect
sig_TPacket(DL_data_ request, Flush buffer)
AX.25 packets
sig_TPacket(DL_data_ indication, Flush buffer)
sig_msg(Flush buffer)
sig_msg(Flush buffer)
sig_msg(Flush buffer)
sig_msg(Keep on sending)
sig_TPacket(DL_data_ confirm, Keep on sending)
AX.25 packets
sig_TPacket(DL_data_ request, Keep on sending)
Pause (only when pausing)
sig_TPacket(DL_disconnect request)
AX.25 packets
sig_TPacket(DL_disconnect_ indication)
sig_msg(disconnect indication)
sig_ receiveEvent(Disconnected)
sig_ receiveEvent(Disconnect)
sig_msg(disconnect confirm)
sig_TPacket(DL_disconnect_ confirm)
AX.25 packets

Figure 10.4: MSC showing signals involved in disconnecting and pausing.

When a side sends a `disconnect` to the TTC, the TTC will set `order` to disconnecting, indicating to other events received in the TTC that its in a special mode. The TTC will send a `FlushBuffer` request to the other side, which responds with an acknowledgment upon receiving. On both sides the TTC will send a `sig_msg` to the channel state machine requesting that it flushes its buffers. The status of the channels will be unchanged.

The TTC will also tell an application that it has been disconnected, again by sending a `sig_receiveEvent` containing the state.

10.2.4 Pause

The signals in the pause functionality are shown on Figure 10.4.

Pause could be used to temporarily stop the communication. Pause is initiated by sending a `pause` signal to the TTC. The TTC waits for the token before sending a `TPacket` with a primitive `DL_DISCONNECT_REQUEST`. When `DL_DISCONNECT_INDICATION` and `DL_DISCONNECT_CONFIRM` is received on either side the state is set to disconnected. The TTC will also tell the application that it has been disconnected, again by sending a `sig_receiveEvent` containing the state.

10.2.5 Send Data

The send data functionality consist of a set of sub-functionalities, which may be used in any order:

- Channel status.
- Open/close a channel.
- Data/no data on channel.
- Send data from highest priority channel.
- No more data to send.

Channel Status

Figure 10.5 (1) shows signals involved in getting the status of a channel.

For an application to be able to send data to the channel state machine, the channel has to be open. To check whether a channel is open or not, an application can send the signal `sig_getStatusChannel` with a channel number as parameter. The channel will then send a `sig_statusChannel` signal back to the application, telling whether or not the channel is open.

Open/Close a Channel

Figure 10.5 (2) and (3) shows the signals involved in opening and closing a channel.

The application can open a channel by sending a `openChannel` signal to the channel state machine. In the same way, if for some reason an application wants to close a channel, a `closeChannel` signal is send to the channel. In the later case the channel state machine

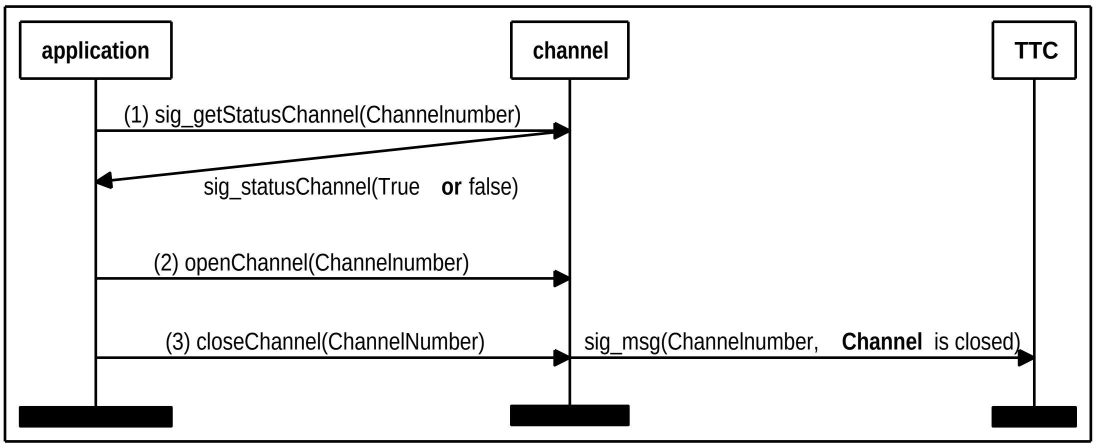

application
channel
TTC
(1) sig_getStatusChannel(Channelnumber)
sig_statusChannel(True or false)
(2) openChannel(Channelnumber)
(3) closeChannel(ChannelNumber)
sig_msg(Channelnumber, Channel is closed)

Figure 10.5: MSC showing signals involved in getting the status of a channel and opening and closing a channel.

will inform the TTC that a channel has been closed by sending this information in a `sig_msg`. The TTC will not send any more data from this channel even if there is more data on the channel.

Data/No Data on Channel

Figure 10.6 shows signals involved in sending data from one side to the other.

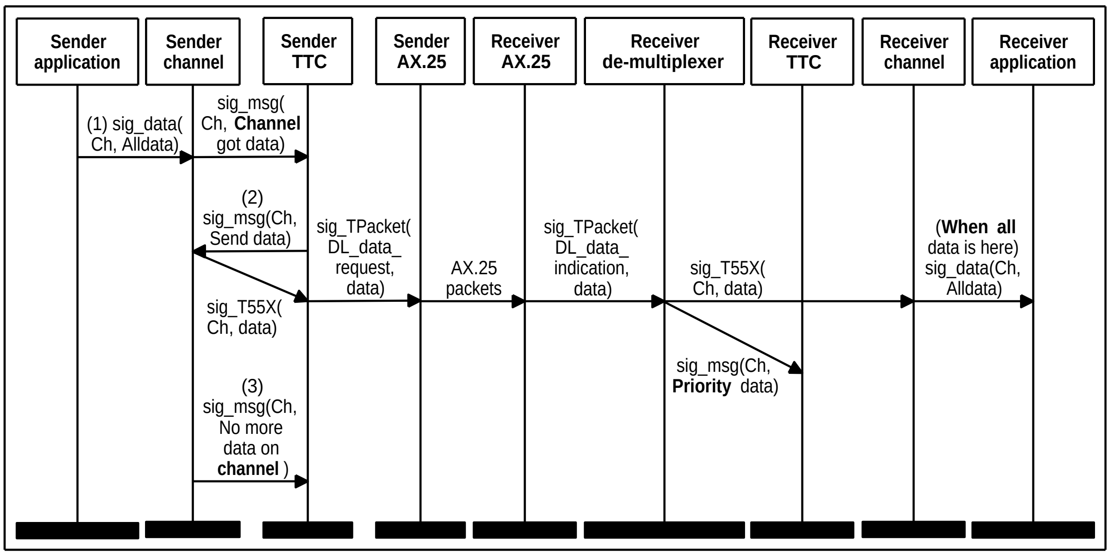

Sender application
Sender channel
Sender TTC
Sender AX.25
Receiver AX.25
Receiver de-multiplexer
Receiver TTC
Receiver channel
Receiver application
(1) sig_data(Ch, Alldata)
sig_msg(Ch, Channel got data)
(2) sig_msg(Ch, Send data)
sig_T55X(Ch, data)
sig_TPacket(DL_data_ request, data)
AX.25 packets
sig_TPacket(DL_data_ indication, data)
sig_T55X(Ch, data)
(When all data is here) sig_data(Ch, Alldata)
sig_msg(Ch, Priority data)
(3) sig_msg(Ch, No more data on channel)

Figure 10.6: MSC showing signals involved in sending data to the other side.

In Figure 10.6 (1) it is shown how data is sent to a channel. When a channel is open, the application can send data to it via a `sig_data` signal. The channel will then send a `sig_msg` to the TTC, informing it that it has data. The TTC can then start to send the data, when it is ready and this data has the highest priority.

In Figure 10.6 (3) it is shown what happens when the last data has been sent. When a channel sends its last data packet, it will also send a `sig_msg` to the TTC, telling it that there is no more data on this channel. The TTC can then either start to send data on another channel, let the other side send or simply wait for more data to arrive.

Send Data From Highest Priority Channel

Figure 10.6 (2) shows the signals involved in the sending a single data package from one side to the other.

When the TTC is ready to send a packet it will send a `sig_msg` to the channel state machine, telling it to send data from the channel with the highest priority. When the channel state machine receives this message it will create and send a T55X packet to the TTC via a `sig_T55X` signal. The TTC will then send a `TPacket` containing the primitive `DL_Data_Request` and the T55X packet and go into *waiting* state, allowing the other side to reply.

When the `TPacket` signal arrives in the de-multiplexer on the other side, the T55X packet is sent to the channel via the `sig_T55X` signal. The T55X packets will then be reassembled in the buffer for the given channel and when the entire dataset has arrived it will be sent to an application via a `sig_data` signal. From the de-multiplexer a `sig_msg` signal is also sent, upon arrival of the `TPacket`. This signal will inform the TTC, that data has arrived on the given channel.

The TTC will then check if it has data with a higher priority, than what it has just received. If it has it will begin to send this, or else it will send a `TPacket` to the other side, telling the other side to keep on sending data.

No More Data To Send

In Figure 10.7 the signals sent when the two sides has no more data to send are shown.

When one side has sent all its data, the TTC will send a `TPacket` to the other side, telling it that this side has no more data.

If the other side does not have any data to send either, it will send a similar package, but it will remain in *running* state. When the first side receives the signal it will also enter the *running* state.

This way both sides are ready to send data. The difference between the two sides is that on side has received a packet without sending another out. This results in a difference between the counter on the last packet sent and the packet counter received from the de-multiplexer. On the other side these two are the same, while a package has been sent from here. When the side with no difference between the counters receives data to transmit it has to ask for the token before it can send its data. With this solution both

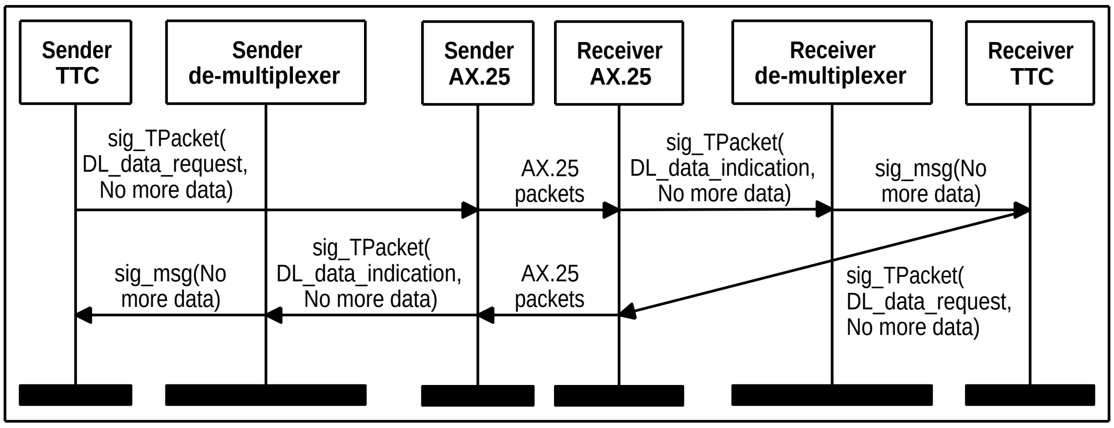

Sender TTC
Sender de-multiplexer
Sender AX.25
Receiver AX.25
Receiver de-multiplexer
Receiver TTC
sig_TPacket(DL_data_request, No more data)
AX.25 packets
sig_TPacket(DL_data_indication, No more data)
sig_msg(No more data)
sig_msg(No more data)
sig_TPacket(DL_data_indication, No more data)
AX.25 packets
sig_TPacket(DL_data_request, No more data)

Figure 10.7: MSC showing signals involved when both sides has no data to send.

sides can initiate the communication simultaneously and still avoid the situation where both sides have the same status and the same token.

10.2.6 Negotiate

This functionality is used to make sure that the last package sent before an error or a disconnect is still received on the other side. When connection is lost the last package sent may have not arrived on the other side. Important packages are therefore retransmitted when the connection is reestablished. All packages are important, with a few exceptions:

- Packages requesting a retransmit.
- All packages that are not sent as data packages. (Unit data, connect and disconnect packages.)

The basic idea is that when side A receives a package from side B, this package also acts as an acknowledgment of the last package sent by side A. By storing the last outgoing package and giving it a number one larger then the last received package, it should always be possible to locate the last package sent.

Figure 10.8 shows how the counters used for negotiation are set. The packet counter is a concept that exists in de-multiplexer and the TTC. The numbers in the box just below the process names on the figure (e.g. [2] under the de-multiplexers and [0,2] under the TTC's) are:

- TTC, left - The packet number stored in the last packet sent (`lastPacket`).
- TTC, right - The packet counter in the TTC. Increased when message is received from the de-multiplexer. The current TTC packet counter will be set in the next packet sent.

- De-multiplexer - The de-multiplexer packet counter is set when the de-multiplexer receives a data packet.

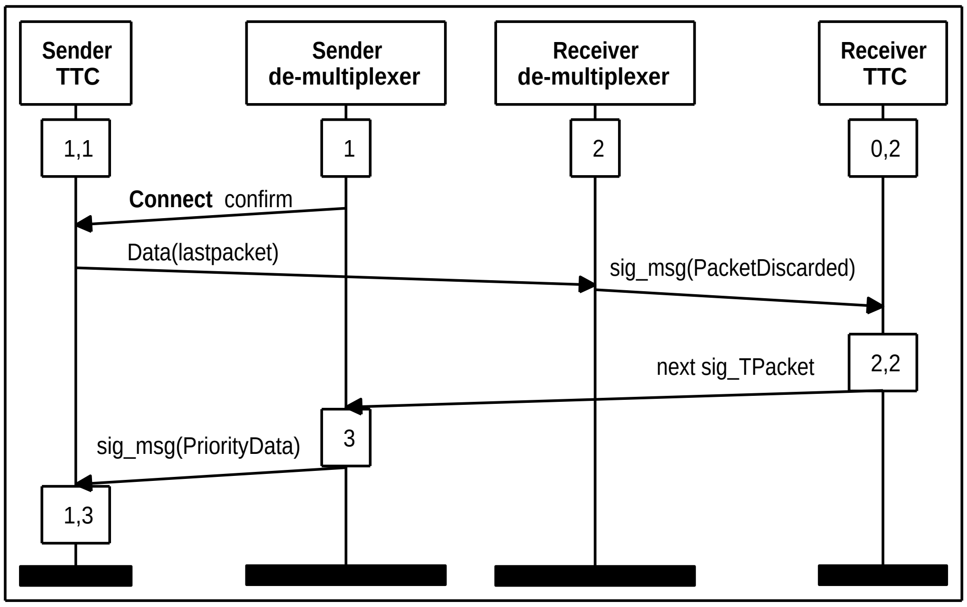


Figure 10.8: MSC showing the values of the counters in a connect example.

The counter is a two bit counter, giving a total of four possibilities. The value of the counter will therefore be between zero and three. Another bit, the `negotiation bit`, is used to indicate that a package is used for negotiation.

The packet counter in the receiver de-multiplexer is not set when it receives the `lastPacket` from the sender TTC. The reason for this is that the `negotiation bit` is set and that the counter in the packet is one smaller than the packet counter in the receiver de-multiplexer. The packet is therefore discarded and the receiver TTC is notified of this. The receiver TTC will then send a packet, thus giving the counter in the new `lastPacket` the value of the TTC's packet counter. When the data packet is received in the sender de-multiplexer, the packet counter in the sender de-multiplexer is set to the counter in the received packet plus one. The sender will update its packet counter when notified of the data reception.

Figure 10.9 shows the signals involved in negotiating and gives an idea of how negotiation is conducted. A retransmit signal will always choose path (1a), whereas connect confirm may select either (1a) or (1b) depending on the packet counter values. If path (1b) is selected, communication will proceed as normal. On path (1a) however another selection must be made as the packet sent may have all ready been received, which can be detected by checking the packet counter values. Path (2a) shows the situation where the packet was not received before and therefore is sent to the channel. Path (2b) shows the situation where the packet is discarded. In both cases the TTC is notified and communication will proceed as normal.

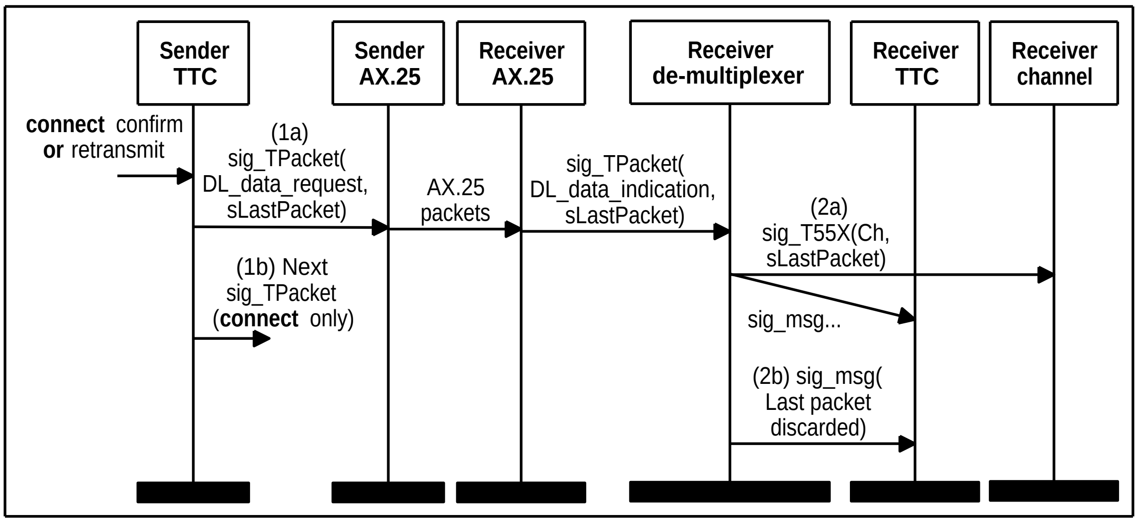

Sender TTC
Sender AX.25
Receiver AX.25
Receiver de-multiplexer
Receiver TTC
Receiver channel
connect confirm or retransmit
(1a) sig_TPacket(DL_data_request, sLastPacket)
AX.25 packets
sig_TPacket(DL_data_indication, sLastPacket)
(2a) sig_T55X(Ch, sLastPacket)
(1b) Next sig_TPacket (connect only)
sig_msg...
(2b) sig_msg(Last packet discarded)

Figure 10.9: MSC showing signals involved when negotiating upon a connect confirm or a retransmit.

The two sides will negotiate when a packet is retransmit is requested or upon a connect confirm. These two situations is described further below.

Retransmit

Retransmit errors occur on the receiving side and is an indication that the package sent from the other side must be retransmitted. The errors are sent from the AX.25 through the de-multiplexer to the TTC. The TTC sends a `TPacket` to the other side requesting a retransmission. Before retransmitting `lastPacket` the TTC sets `negotiate` bit indicating that a negotiation is commencing. An important note is that a retransmit package does not overwrite `lastPacket`. This way it is possible to have multiple retransmit requests and still get the correct result.

Connect

When a side sends a connect request, it will eventually receive a connect confirm if the connect was successful. The side will then proceed by checking the values of packet counters in the TTC and the `lastPacket`. If the values are the same, the side will proceed as normal. Else it will send its `lastPacket` with the `negotiate` bit set, making sure that the packet is received correctly on the other side. The other side will check this by checking the values of the received packet and the packet counter in the de-multiplexer. If the value of the counter in the received packet is the highest, the packet is received as normal. Else it will be discarded and the side will proceed as normal.

10.2.7 Error Handling

The purpose of error handling is to ensure that no data will be lost, no matter which error is received from the AX.25. When dealing with errors it is important to realize that errors can arise at all times. The different errors are categorized into groups, depending on the effect they have on the data and the status. The following groups have been categorized:

- Errors with no importance.
- Fault detected in packages received.
- Both sides gets disconnected, meaning that all data in AX.25 was discarded.

The first category is status information from AX.25 which, as indicated, does not have any effect on the T55X layer.
An error from the second category indicates that data has been discarded on the receiving side. The receiving side will send a message to the sending side requesting that the last packet is retransmitted. The sending side will then retransmit its `lastPacket` with `negotiate` bit set. The specific functionality involved in this is described in section 10.2.6.
When an error of the last category occurs, all packages lost will be retransmitted, using the normal negotiate method described in section 10.2.6.

10.2.8 Auxiliary Functionality

There are two auxiliary functionalities in the design of the T55X. These are:

- Initialization.
- Get status.

Initialization

The initialization must be sent from an application to the TTC before any other signal and must be sent on both sides. It will send a `sig_msg` to the channel, telling it to flush all buffers.

Get Status

If an application wants to know the status of the T55X it can send a `getStatus` signal to the TTC. The TTC will then send its current state in a `sig_receiveEvent` signal. The received states can be either disconnected, running or waiting.

Chapter 11

Test Specification

This chapter will specify the different tests, that must be conducted in order to validate and verify the model. Before reading further it is recommended reading Appendix C.

11.1 Environment

When the simulation is to be conducted an environment has to be created. The AX.25 environment was originally planned to be a model of the AX.25 layer but since this model was not available a simplified model is created. The model should have the following capabilities:

- It should be able to send and receive primitives and changing these as the AX.25 prescribes.
- Simulate errors by dropping 'TPackets' and sending out error messages.

Application environments are created specific to the simulation/test case. Even though the model has been created to handle errors, the data link model has not implemented a feature to drop packages, and it is therefore not possible to test for errors in the T55X for now. Before tests are executed in exhaustive mode the errors are neglected. A possible way to implement the errors would be to insert a state machine between the two AX.25 state machines which will replace some of the signals it gets with errors.

11.2 Functionality

In order to test each functionality, a simulation will be performed to check if the responses are as presumed. Table 11.1 comprises the output expected from a signal sent to the T55X layer.

The tests in Table 11.1 will test each individual functionality. This means that it does not test how the different functionalities interact when requested simultaneously. For each functionality tested, it is recommended to refer to the appropriate figures in section 10.2.

When testing the different functionalities, they must reside in the correct test environment for the results to be valid. The test environments for the tests in Table 11.1 are described below:

- Send beacon - Will be sent before anything else, thus there is no connection between the satellite and the ground.
- Connect - Same environment as for send beacon.
- Disconnect - Connection has to be established between the two sides.
- Pause - Same as disconnect.
- Send data - Channel 2 is closed on the satellite. And connection is established between the two sides.

11.3 Simulation

When the different functionalities have been tested individually, they should be tested together. In order to do this a test scenario has been created. The expected output and the purpose of the test steps are displayed in Table 11.2. This test will check if the model can behaves as it is expected to, but it will not check if it always behaves correctly.

Tests where the simulator in ObjectGEODE gives random inputs to a simulation should also be conducted to find errors that will not emerge in tests constructed by the designer.

11.4 Verification

A verification of the model must be conducted, to ensure that the syntax of the model is correct, e.g. there is no deadlocks, livelocks etc. The verification tool in ObjectGEODE should be used for this test. It is imperative that the verification is completed, as errors may have serious consequences on the satellite.

11.5 Validation

To check if the model behaves as expected there are two options: Verify, which cover all traces in a state graph or searching for the behavioral pattern. The search option has been chosen to be used because of the easier implementation.

Functionality	**Input**	**Expected output**
Send beacon	Satellite application: `sendConLess( '112233445566778899')`	Ground application: `sig_receiveEvent(IncommingData, '112233445566778899')`
Connect	1. Satellite application: `getStatus`	Satellite application: `sig_receiveEvent(Disconnected)`
	2. Satellite application: `connectTo`	Ground application: `sig_receiveEvent(Waiting)` Satellite application: `sig_receiveEvent(Running)`
Disconnect	Satellite application: `disconnect`	Ground application: `sig_receiveEvent(Disconnected)` Satellite application: `sig_receiveEvent(Disconnected)`
Pause	Satellite application: `pause`	Ground application: `sig_receiveEvent(Disconnected)` Satellite application: `sig_receiveEvent(Disconnected)`
Send data	1. Satellite application: `sig_getStatusChannel(2)`	Satellite application: `sig_statusChannel(false)`
	2. Satellite application: `openChannel(2)`	No output
	3. Satellite application: `sig_data(2, '0123456789ABCDEF')`	Ground application: `sig_data(2 '0123456789ABCDEF')`

Table 11.1: Test scenarios for the functionality in the T55X layer. Signals are only sent from the satellite. This will however test signals sent from both sides as the two sides are identical in the model.

Step	Input	Expected output	Purpose
1	Satellite application: `sendConLess( '112233445566778899')`	Ground application: `sig_receiveEvent( IncommingData, '112233445566778899')`	Initializes the model.
2	Satellite application: `connectTo`	Ground application: `sig_receiveEvent( Waiting)` Satellite application: `sig_receiveEvent( Running)`	Establishes connection.
3	Satellite application: `sig_getStatus- Channel(1)`	Satellite application: `sig_statusChannel( false)`	Find out if channel 1 is open.
4	Satellite application: `openChannel(1)`	No output	Open channel 1.
5	Satellite application: `sig_data(1, '0123456789ABCDEF')`	Ground application: `sig_data(1, '0123456789ABCDEF')`	Send data on channel 1.
6	Satellite application: `sig_getStatus- Channel(2)`	Satellite application: `sig_statusChannel( false)`	Find out if channel 2 is open.
7	Satellite application: `openChannel(2)`	No output	Open channel 2.
8	Satellite application: `sig_data(2, '0123456789ABCDEF')`	Next signal send before output is ready	Send data on channel 2.
9	Satellite application: `disconnect`	Ground application: `sig_receiveEvent( Disconnected)` Satellite application: `sig_receiveEvent( Disconnected)`	Send a disconnect and flush the buffers.
10	Satellite application: `connectTo`	Ground application: `sig_receiveEvent( Waiting)` Satellite application: `sig_receiveEvent( Running)`	Establishes connection again.

Step	Input	Expected output	Purpose
11	Satellite application: `sig_getStatus-Channel(2)`	Satellite application: `sig_statusChannel(true)`	Find out if channel 2 is open.
12	Satellite application: `sig_data(2, '0123456789ABCDEF')`	Next signal send before output is ready	Send data on channel 2.
13	Satellite application: `pause`	Ground application: `sig_receiveEvent(Disconnected)` Satellite application: `sig_receiveEvent(Disconnected)`	Disconnect.
14	Satellite application: `sig_getStatus-Channel(1)`	Satellite application: `sig_statusChannel(true)`	Find out if channel 1 is open.
15	Satellite application: `sig_data(1, '0123456789ABCDEF')`	Status is disconnected	Send data on channel 1.
16	Satellite application: `connectTo`	Ground application: `sig_receiveEvent(Waiting)` `sig_data(1, '0123456789ABCDEF')` `sig_data(2, '0123456789ABCDEF')` Satellite application: `sig_receiveEvent(Running)`	Establishes connection again, and receive data on the two channels.

Table 11.2: Test scenario for testing the functionality in the T55X layer when all functionalities are working together.

In Figure 11.1 a MSC tree has been created which describes some of the main behavior in the model. The first consideration is at the top level hierarchy which is an 'and' composition. This means that the initialize signal must be sent at some point. This is reasonable since it should be the first signal at each end. The 'repeat' composition is needed because the `connectTo` signal can be transmitted more than once. Following downward is the 'and' composition which guarantees that there is an connection is established before sending any data. The three leaves: SendData, pause, disconnect can all be run multiple times because of the 'repeat' above the 'or' composition. The MSC leafs are shown in figures 11.2 to 11.6.

This validation only shows that this specific behavior is possible not that this is always true.

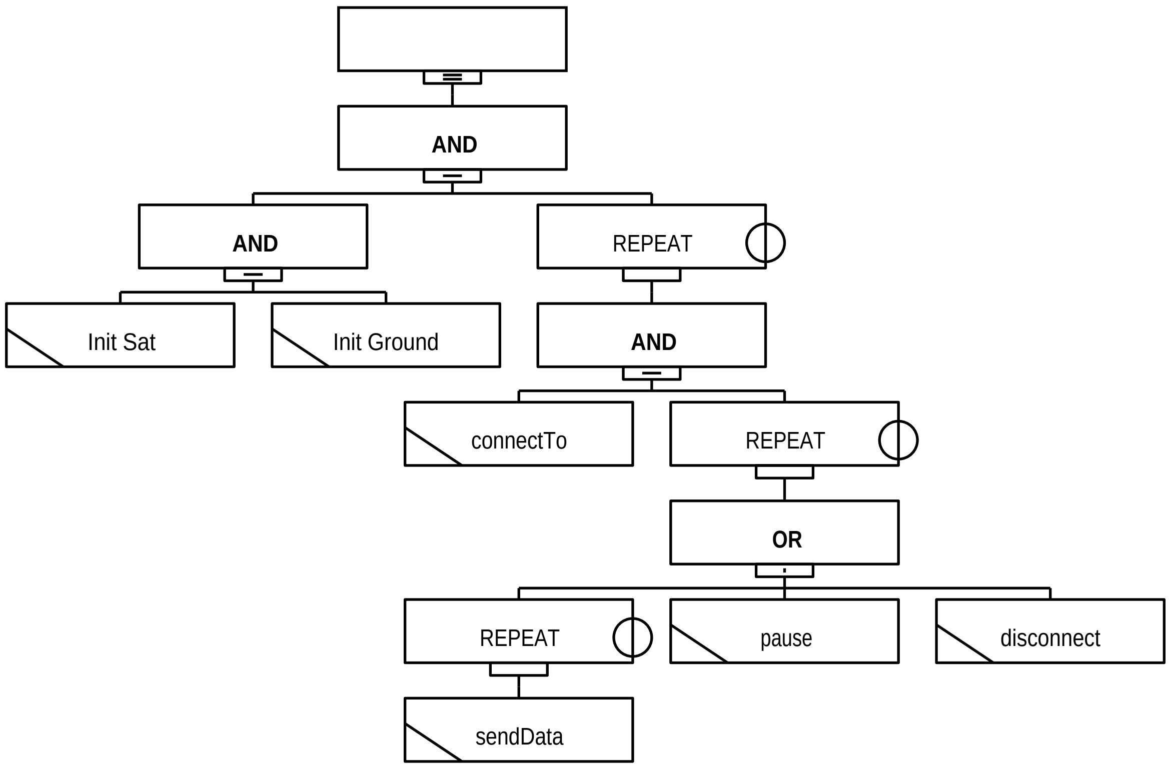


Figure 11.1: Behavioral MSC tree of the SDL model

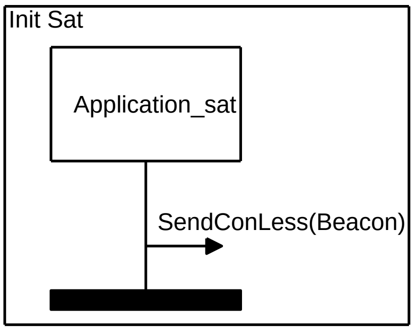


Figure 11.2: Leaf which describes both initialization at ground and satellite.

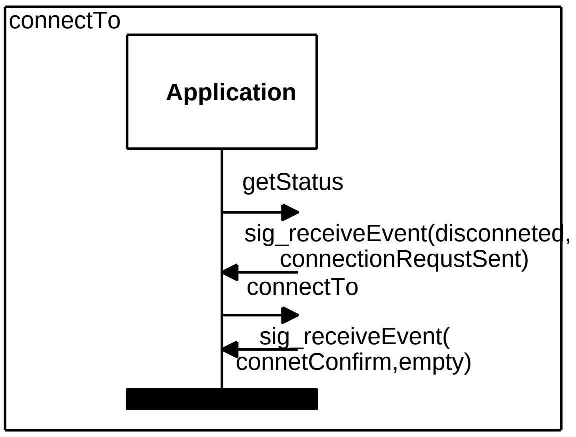


Figure 11.3: Leaf describing the connectTo pattern.

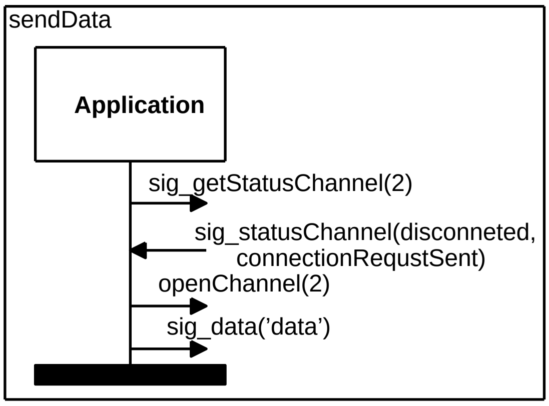


Figure 11.4: Leaf describing the sendData pattern.

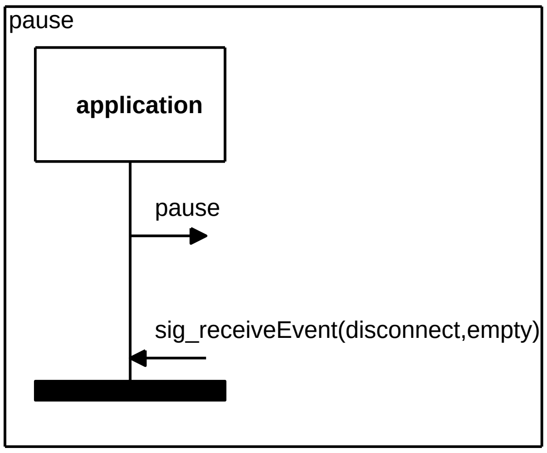


Figure 11.5: Leaf describing the pause pattern.

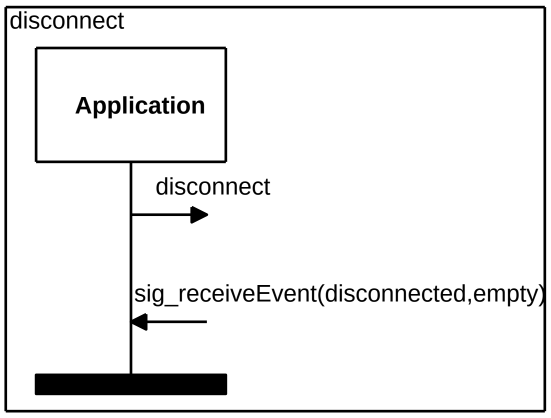


Figure 11.6: Leaf describing the disconnect pattern.

Chapter 12

Design Validation

In this chapter the design test is described. The tests will be conducted accordingly to the test specification in Chapter 11. Through out this chapter references to directories on the project CD-ROM are made as e.g. /SDL/

12.1 Simulation

The functionalities which are tested using non exhaustive simulation are:

- Send connection less data
- Connect
- Send Data
- Pause
- Disconnect

These will be tested using the AX.25 environment described in section 11.1. An application is created which can be used to test all the different functionalities. This can be seen in /SDL/ in the `transport.pr` file. All the tests are performed when both the T55X layers are initialized. The satellite has been chosen to be the initiating part.
Common for all test are that a scenario was built as described in the test specification. The input signals from 11.1 are feed to the model manually, afterwards ObjectGEODE chooses random transitions to complete the simulations. The MSC created by ObjectGEODE was edited so the only items left were the application and the interfacing processes. The unedited MSC can be found in /MSC/functionality/ . On figure 12.1 to 12.5 the MSC results can be seen.

All of the test had the expected result as specified in the test specification.

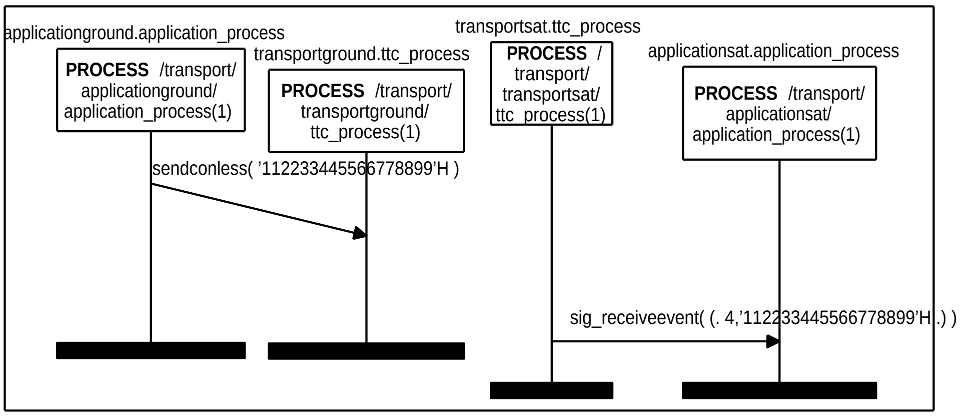

applicationground.application_process
transportground.ttc_process
transportsat.ttc_process
applicationsat.application_process
PROCESS /transport/ applicationground/ application_process(1)
PROCESS /transport/ transportground/ ttc_process(1)
PROCESS / transport/ transportsat/ ttc_process(1)
PROCESS /transport/ applicationsat/ application_process(1)
sendconless('112233445566778899'H)
sig_receiveevent((. 4,'112233445566778899'H .))

Figure 12.1: MSC for the send connection less functionality. First the signal sendconless is seen with the data parameter 112233445566778899 which is later seen received on the application on ground. The number 4 indicates an event type for connection less data.

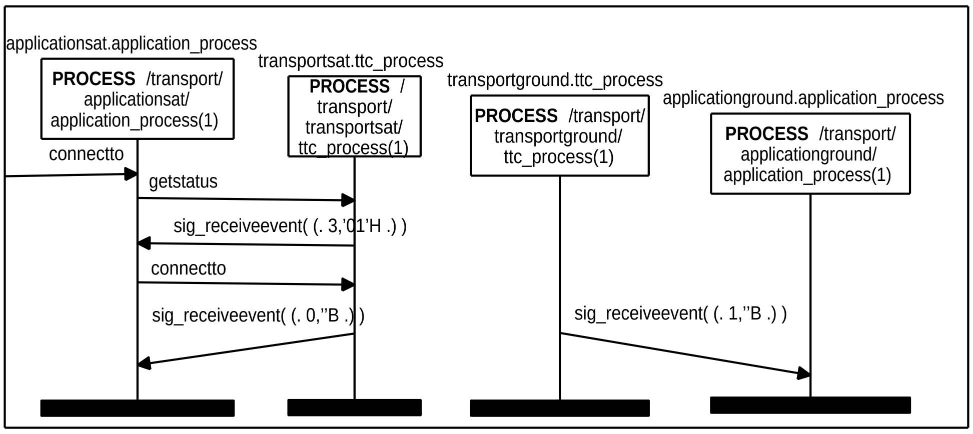

applicationsat.application_process
transportsat.ttc_process
transportground.ttc_process
applicationground.application_process
PROCESS /transport/ applicationsat/ application_process(1)
PROCESS / transport/ transportsat/ ttc_process(1)
PROCESS /transport/ transportground/ ttc_process(1)
PROCESS /transport/ applicationground/ application_process(1)
connectto
getstatus
sig_receiveevent((. 3,'01'H .))
connectto
sig_receiveevent((. 0,''B .))
sig_receiveevent((. 1,''B .))

Figure 12.2: MSC for the connect functionality. First the application checks the connected state of the TTC. The TTC responds with a `sig_receiveEvent` with parameters 3 indicating disconnected and '01'H indication no connect request has been sent yet. Now the actual connect request is sent. Later when the connection has been made the two applications are informed with another `sig_receiveEvent`. The parameter: 0 is a connect confirm and 1 is a connect indication. The ''B indicates a placeholder for a parameter which are not used.

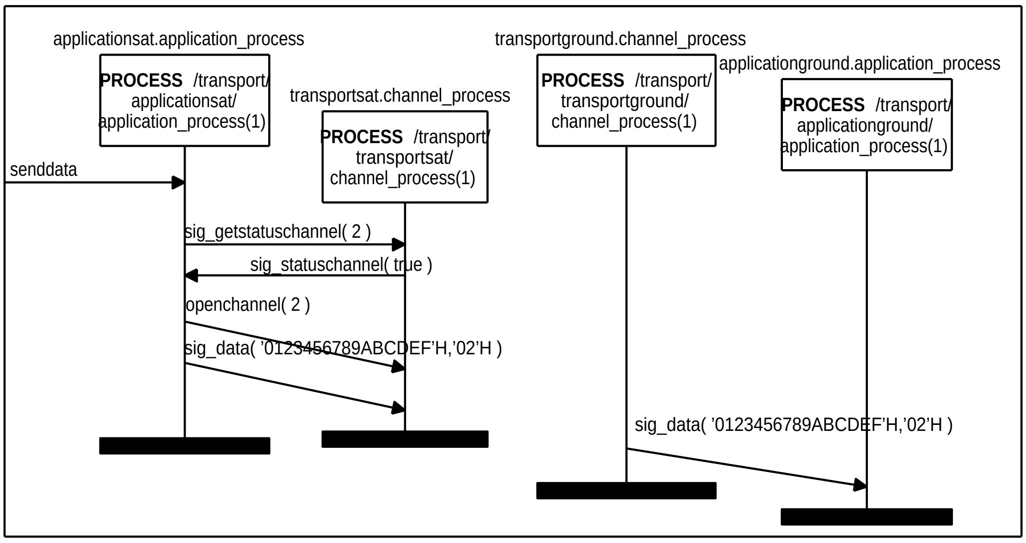

applicationsat.application_process
PROCESS /transport/ applicationsat/ application_process(1)
transportsat.channel_process
PROCESS /transport/ transportsat/ channel_process(1)
transportground.channel_process
PROCESS /transport/ transportground/ channel_process(1)
applicationground.application_process
PROCESS /transport/ applicationground/ application_process(1)
senddata
sig_getstatuschannel(2)
sig_statuschannel(true)
openchannel(2)
sig_data('0123456789ABCDEF'H,'02'H)
sig_data('0123456789ABCDEF'H,'02'H)

Figure 12.3: MSC for the send data functionality. At first the application checks the status of channel number two. Then the channel is initialized before sending data to it. Some time after the data is received on ground.

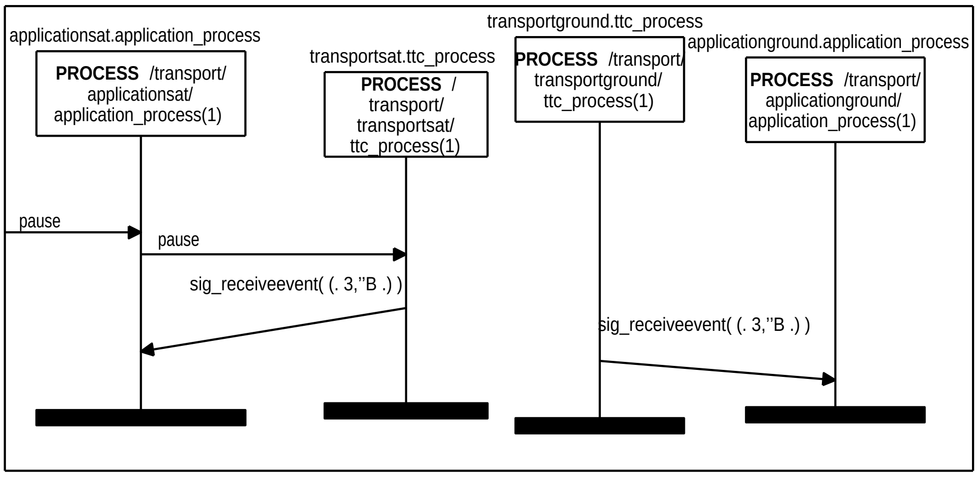

applicationsat.application_process
PROCESS /transport/ applicationsat/ application_process(1)
transportsat.ttc_process
PROCESS / transport/ transportsat/ ttc_process(1)
transportground.ttc_process
PROCESS /transport/ transportground/ ttc_process(1)
applicationground.application_process
PROCESS /transport/ applicationground/ application_process(1)
pause
pause
sig_receiveevent((. 3,''B .))
sig_receiveevent((. 3,''B .))

Figure 12.4: MSC for the pause functionality. The application sends `pause` to the TTC which responds with status disconnected when this becomes true. Also the ground application gets informed when the T55X layer becomes disconnected.

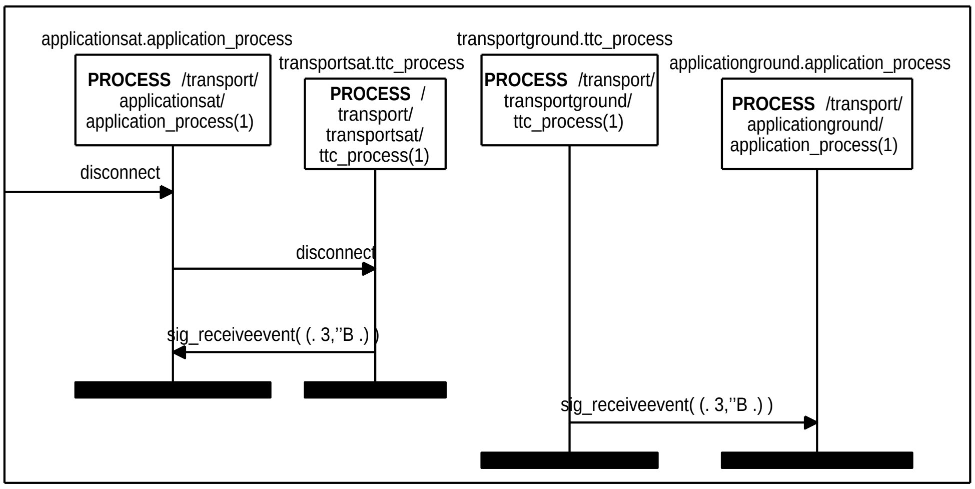

applicationsat.application_process
PROCESS /transport/ applicationsat/ application_process(1)
transportsat.ttc_process
PROCESS / transport/ transportsat/ ttc_process(1)
transportground.ttc_process
PROCESS /transport/ transportground/ ttc_process(1)
applicationground.application_process
PROCESS /transport/ applicationground/ application_process(1)
disconnect
disconnect
sig_receiveevent((. 3,"B .))
sig_receiveevent((. 3,"B .))

Figure 12.5: MSC for the disconnect functionality. The signals in the interface to the applications correspond to the pause functionality. The only difference is that it begins with a disconnect.

The next test specification is all functionalities combined in a specific trace in the model. Again all of the signals not concerning the interface of either application has been removed, the unedited MSC is in file `compleateSpecificTest.eps` in /MSC/functionality on the project CD-ROM. The MSC has been divided into three figures 12.6,12.7 and 12.8. Since each functionality has been described the only interesting functionalities will be highlighted. The entire test specification could be read while following the MSC but this is left as an exercise for the reader.

- 'Disconnect' - The flush functionality of the disconnect is shown in Figure 12.7. The important signals to observe is that data is sent on channel two. Then before finishing the transmission a disconnect occurs which flushes the buffers. Again the connection is established and another data transmission on channel two is attempted. By accepting this transmission it shows that the channels have been flushed.

- 'Pause' - The pause/resume functionality is shown in Figure 12.8. The things to notice is that data is being transmitted on channel two before the `pause` signal. Afterwards a connection is established and the transmission continues. The data is received in the last signal `sig_data`.

- Send data - In Figure 12.8 below the pause which couses both applications to receive a `sig_receiveEvent`, data is transmitted on channel while not connected. When the connection is reestablished the data is transmitted and received as the second last signal `sig_data`

- Highest priority data - Noticing that in Figure 12.8 data on channel two is sent before data on channel one, and by seeing that the first data to be received is the data which was on channel one shows that the highest priority is transmitted first.

Even though these tests of functionalities all have had the expected results nothing can be said about every test case. To test every case the exhaustive mode is needed which will be described in section 12.2 and section 12.3.

12.2 Verification

When conducting this test a few simplifications has been made.

- Both application layers are initialized.
- Filter conditions has been added, to maintain the queue length on processes equal to or below two.
- The number of channels are limited to three.

When conducting this test it became apparent that the models were too complex and an exhaustive simulation with all signals would take too much time and require too much machine power. The signal feed to the model was changed so that only one type of signal was tested at a time. These test completed with no errors. This form of verification cannot be used to guarantee that errors will not appear when all signals are added.
An attempt was made to simplify the model so that an exhaustive simulation could be possible. The following items was tried:

- Connect was limited to states were the TTC state machine was disconnected and relying on the TTC to inform about its state instead of using `getStatus`.
- Open/Closed was removed for a channel, making the channel inform when there was room on a channel.

With these simplifications the time to exhaustively test a signal decreased but not enough to test all signals at once. If applied to the entire model this approach could be used to reduce the number of states that the simulator must explore and finally make it possible to run a exhaustive simulation.

12.3 Validation

Validation is used to check that the model responds to stimuli as stated in the requirement specification. When informing ObjectGEODE two formalisms for expressing the requirement specification are used:

- Stop conditions - Expresses a condition which halts the system if true.

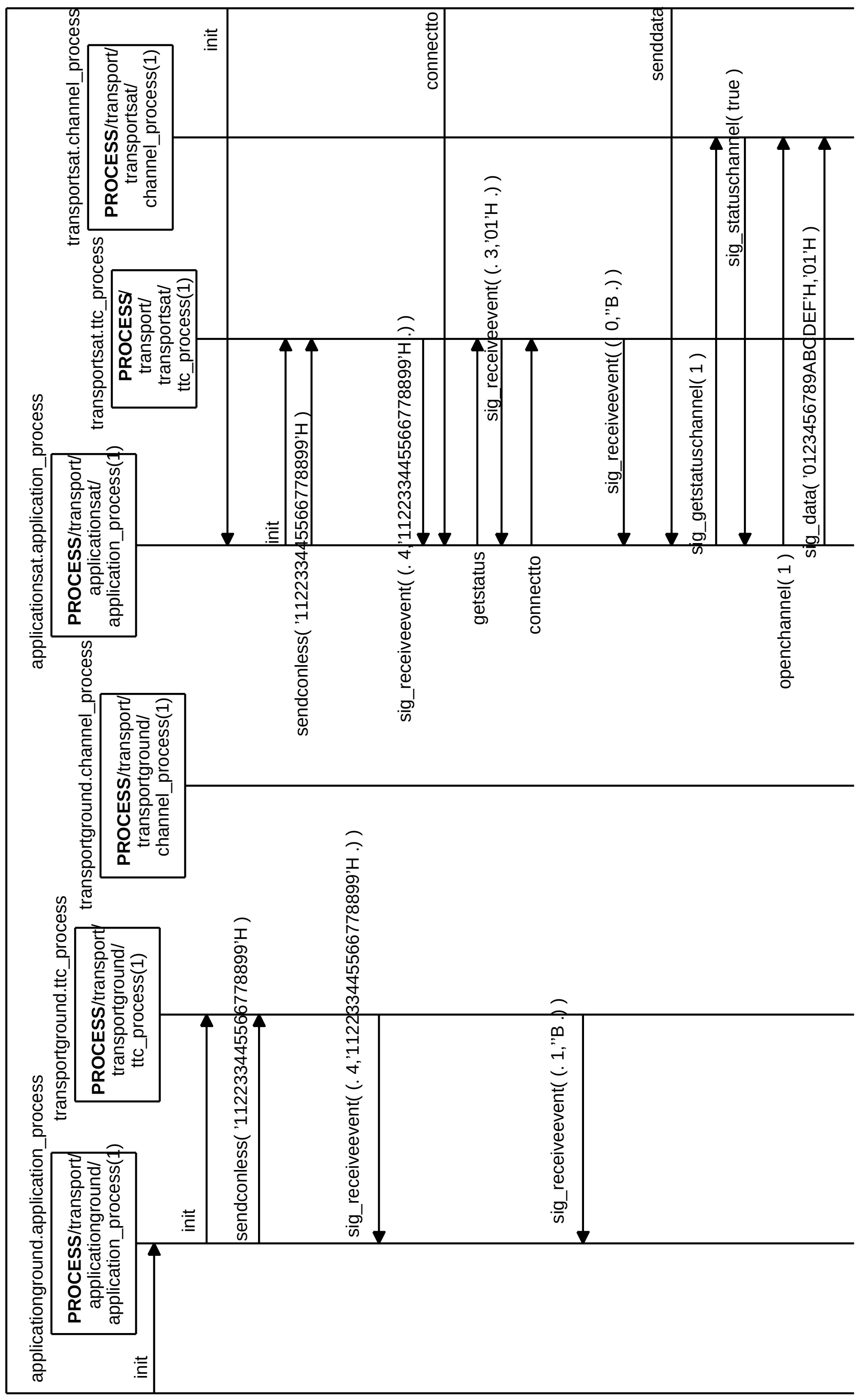

applicationground.application_process
PROCESS/transport/applicationground/application_process(1)
transportground.ttc_process
PROCESS/transport/transportground/ttc_process(1)
transportground.channel_process
PROCESS/transport/transportground/channel_process(1)
applicationsat.application_process
PROCESS/transport/applicationsat/application_process(1)
transportsat.ttc_process
PROCESS/transport/transportsat/ttc_process(1)
transportsat.channel_process
PROCESS/transport/transportsat/channel_process(1)
init
init
sendconless('112233445566778899'H)
sig_receiveevent((. 4,'112233445566778899'H .))
sig_receiveevent((. 1,''B .))
init
init
sendconless('112233445566778899'H)
sig_receiveevent((. 4,'112233445566778899'H .))
connectto
sig_receiveevent((. 3,'01'H .))
getstatus
connectto
sig_receiveevent((. 0,''B .))
senddata
sig_getstatuschannel(1)
sig_statuschannel(true)
openchannel(1)
sig_data('0123456789ABCDEF'H,'01'H)

Figure 12.6: MSC number 1 for the combined functionality test specification.

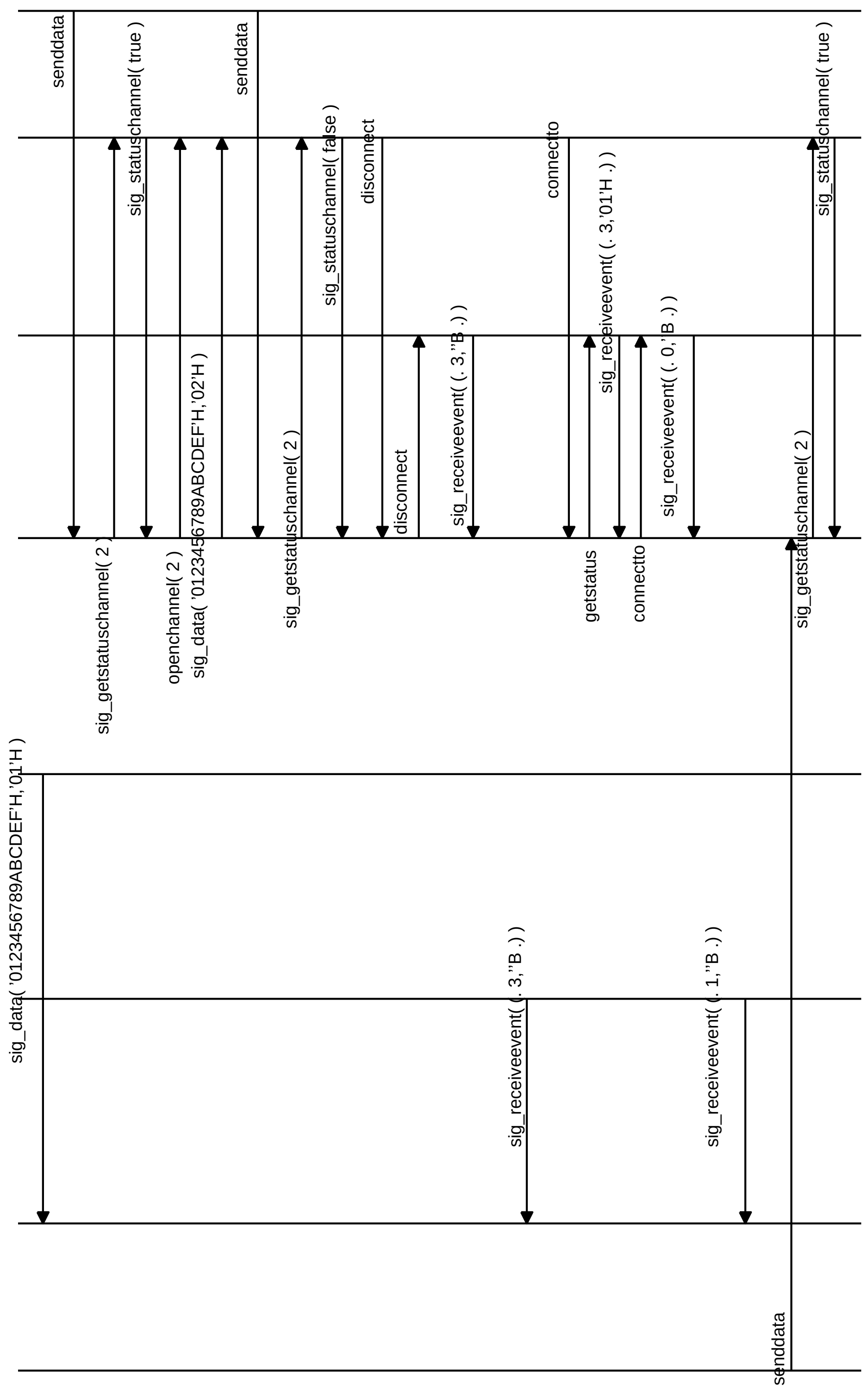

senddata
sig_getstatuschannel(2)
sig_statuschannel(true)
openchannel(2)
sig_data('0123456789ABCDEF'H,'02'H)
senddata
sig_getstatuschannel(2)
sig_statuschannel(false)
disconnect
disconnect
sig_receiveevent((. 3,''B .))
connectto
getstatus
sig_receiveevent((. 3,'01'H .))
connectto
sig_receiveevent((. 0,''B .))
sig_getstatuschannel(2)
sig_statuschannel(true)
sig_data('0123456789ABCDEF'H,'01'H)
sig_receiveevent((. 3,''B .))
sig_receiveevent((. 1,''B .))
senddata

Figure 12.7: MSC number 2 for the combined functionality test specification.

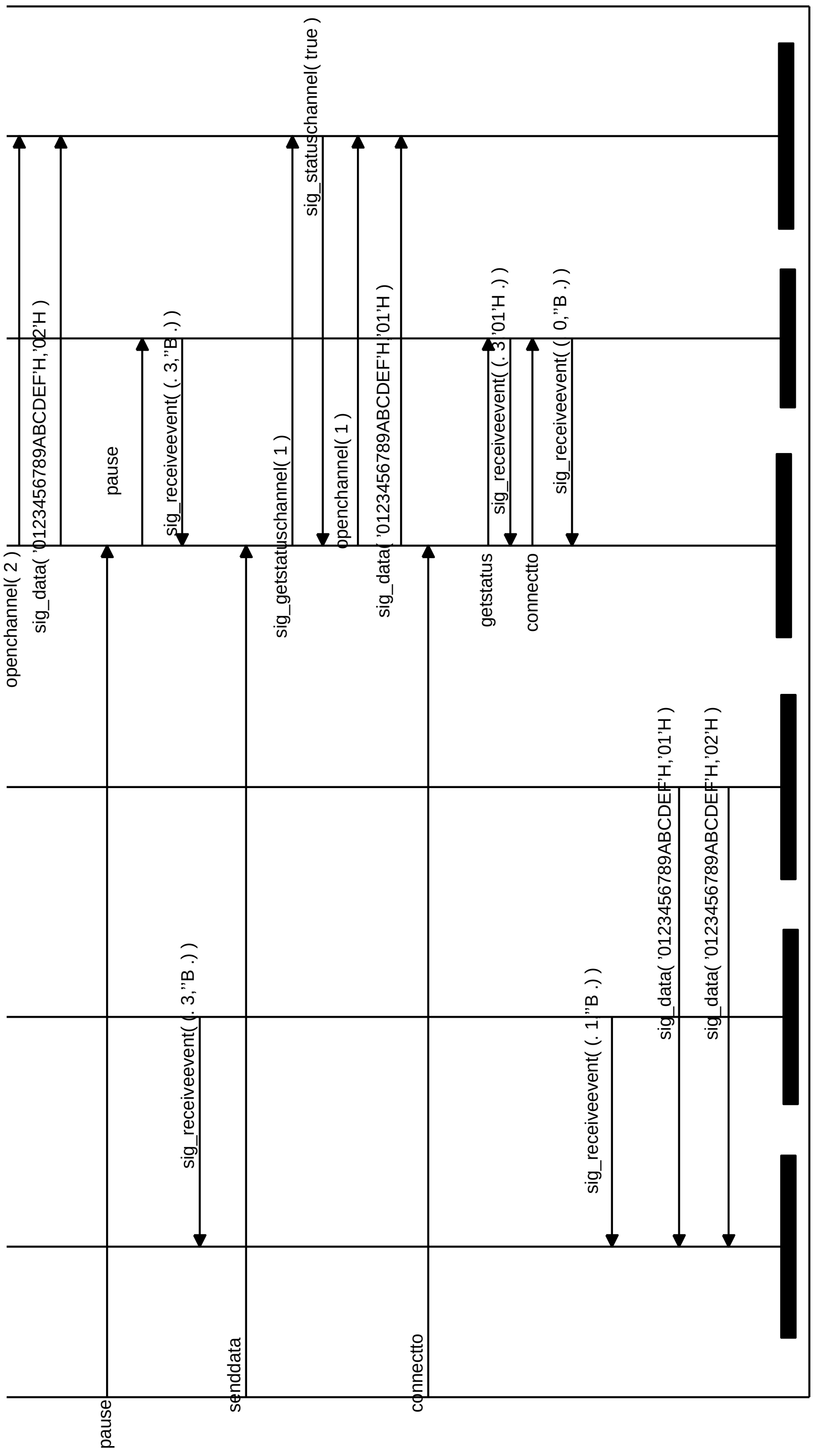

openchannel(2)
sig_data('0123456789ABCDEF'H,'02'H)
pause
sig_receiveevent((. 3,"B .))
sig_getstatuschannel(1)
sig_statuschannel(true)
openchannel(1)
sig_data('0123456789ABCDEF'H,'01'H)
getstatus
sig_receiveevent((. 3,'01'H .))
connectto
sig_receiveevent((. 0,"B .))
sig_data('0123456789ABCDEF'H,'01'H)
sig_data('0123456789ABCDEF'H,'02'H)
sig_receiveevent((. 3,"B .))
sig_receiveevent((. 1,"B .))
pause
senddata
connectto

Figure 12.8: MSC number 3 for the combined functionality test specification.

- MSC - Expresses a part of the model as a signal sequence. Parts can be put together to form the complete system.

From the previous experience it was decided that exhaustive simulation was not possible. This removes the possibility for validating the model.

Chapter 13

Summary

This part has described the design phase of the T55X layer in SDL.

First an overview was given of the layer, explaining the signals and signal structure. This was preceded by a more detailed description of the most important functionalities in the layer.

Next test specifications for simulation, verification and validation tests were stated.

A description of the completed test and the results of these were given. It was not possible to validate the model, but simulations suggested that the model was correct accroding to its specification. The next step is to further simplify the model so that an exhaustive simulation is possible within reasonable time.

Part IV

Implementation

This part describes the implementation and test of the system. The AX.25 protocol has been partly implemented and tested. The implemented modules are described and a list of missing parts from the AX.25 implementation is specified. The T55X layer is not implemented, but the structure of a potential implementation is described.

Chapter 14

AX.25

This chapter documents the implementation of the AX.25 protocol. The following topics will be covered: Modifications to the original AX.25 design, implementation environment and the actual implementation, including the different modules and common definitions. Furthermore test specification for the different modules will be presented. It is recommended as a minimum to have read Appendix B, but for the full understanding the AX.25 documentation found in [5] can be necessary. The c-program for the implementation can be found on the project CD-ROM in the folder 'sourcecode/'.

14.1 Modifications

In general there is no implementation specification available for the AX.25, as stated in [5], hence this chapter will cover a specific implementation that satisfies the requirements for this project. This section will cover the parts of the protocol that has been modified or left out entirely.

14.1.1 Segmenter

In this implementation the segmenter is divided into two parts: A segmenter module and a reassembler module. The purpose is to have as many independent modules as possible when distributing assignments in the implementation.

Segmenting connection-less (data in U-frames) is very poorly described in the AX.25 specification, because it requires the PID-field, which is not part of an U-frame. It was decided not to segment connection-less data, which means that the maximum length of connection-less data is 256 bytes. Since connection-less data is only used in beacon mode this will not be a problem, because the required length is 256 bytes, as stated in chapter 7.

14.1.2 Physical Layer

Since the hardware interface was not available from OSSS when the implementation started, it was not possible to use the correct device driver or hardware. To test the rest of the implementation it was decided to implement a physical layer that utilizes the TCP/IP protocol stack, so tests could be done on any network using the TCP/IP protocol stack. Unfortunately a group member decided to take a leave from his studies before finishing this implementation. Because of this situation the physical layer will not be documented further in this report and have not been fully implemented.

14.1.3 Management Data-link

This part has not been implemented because there was no time for it. Leaving this part out instead of another was decided because it is possible to test the actual data transmission functionality of the AX.25 protocol without this part.

Recalling from section B.1.1 in the AX.25 appendix the management data-link part is responsible for negotiating parameters between two AX.25 protocols. Instead of negotiating these parameters they are implemented as constants in this implementation.

Leaving this part out in the final implementation might not be possible because it makes it impossible for radio amateurs to contact the satellite if they are not using the same parameters.

14.1.4 Data-link

Because of time constraints, timers have not been implemented in the data-link module. The functionality of the timers are encapsulated in separate functions, so the only thing missing is implementing the timers using the facilities offered by the desired operating system.

14.1.5 Link Multiplexer

The link multiplexer is implemented without any multiplexing features, since there is only one data link available. Therefore the link multiplexer state machine is simply a dummy layer connecting the data-link state machine to the physical state machine. It will always allow the data-link state machine to seize control of the radio. The link multiplexer is included in order to avoid changes in the data-link state machine. (The data-link state machine could just as well access the physical layer).

14.2 Platform and Environment

Because there was little time for the implementation process it was decided to implement the protocol in a Solaris operating system running on a SPARC processor (the servers at the university). This is a well known platform for the group, and was chosen to avoid spending time on the practical problems (constant upload of new software, debugging, testing, etc.) involved in implementing software for an embedded micro-controller and for the group, new operating system.

The only compiler available for the on-board computer is an ANSI-C compiler, hence this implementation is also programmed in this language, so that most of the code can be reused in the final implementation.

Implementing the protocol for the satellite will be a matter of porting the source code to the operating system and the on-board computer. Some specific parts that will need to be ported is the thread concept and memory allocation.

14.3 Common Definitions

This section lists and describes the common data structures and constants in the implementation. All definitions are in the `ax25.h` file that can be found on the project CD in /c.

14.3.1 Data Structures

The common data structures in the AX.25 protocol are the frames send between modules and between two peers.

I-frames are defined as structs with the elements listed:

```
typedef struct {
  unsigned char address[14];
  unsigned char control;
  unsigned char pid;
  unsigned char *data;
  unsigned char dataLength;
  unsigned short int fcs;
} iFrame;
```

U-frames are also defined as structs, but with different elements:

```
typedef struct{
  unsigned char address[14];
  unsigned char control;
  unsigned char *data;
  unsigned char dataLength;
  unsigned short int fcs;
} uFrame;
```

Finally S-frames are defined as:

```
typedef struct{
  unsigned char address[14];
  unsigned char control;
  unsigned char *data;
  unsigned char dataLength;
  unsigned short int fcs;
} sFrame;
```

One may notice that the S-Frame struct is similar to the U-Frame struct. It is defined independently to differentiate between the different data frames.

14.3.2 Constants

As stated in section 14.1.3 the management data-link part has been left out, hence those parameters normally negotiated are defined as constants, which along with other constants are defined here:

- `N1=256 -` The byte length of the info field (`N1` is not very descriptive but is used in the implementation because it is used through out the documentation of the AX.25).
- `N2=10 -` Number of retries.
- `WINDOW_SIZE=7 -` Sliding window size.
- `MY_CALLNAME={'a','b','c','d','e'} -` Call-name used for source address.
- `DEST_CALLNAME={'b','b','c','d','e'} -` Call-name used for destination address.

14.4 Overview

This section will give a general overview of the modules in the implementation and how they interact. Figure 14.1 illustrates how the implementation is structured regarding interfaces and threads.

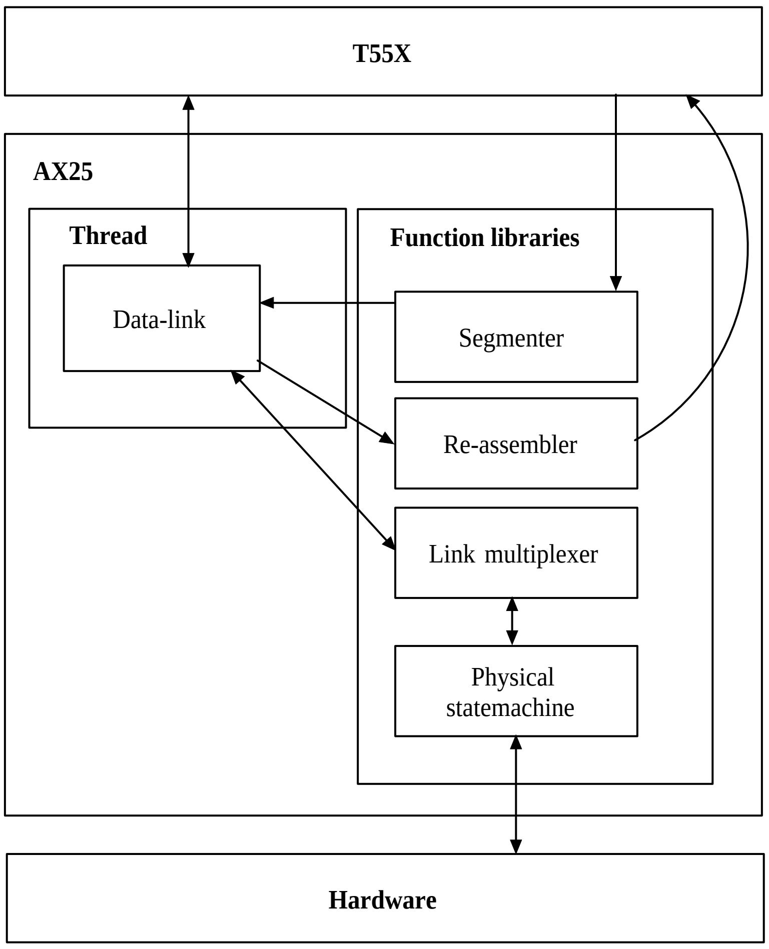


Figure 14.1: Overview of the AX.25 implementation.

The only thread in the system is the data-link state machine. It communicates with the T55X and the hardware through the function libraries available. This means that frames are send from the segmenter to the data-link state machine they will be stored in a queue and the thread will process them later.

In the following sections all the implemented modules are described.

ating segments two and one. The frames will look like this:

PID	Info		
0x8	2	3	4

PID	Info		
0x8	1	5	6

The last frame is generated outside the loop, because it has a unique length. Besides the `dataLength` parameter in the frame the last frame is generated as the other frames. In this example there is a single byte left, and the last frame will look like this:

PID	Info	
0x8	0	7

Finally the function will return a `DATA_OK` value.

`UnitDataRequest`

As stated in section 14.1 connection-less data can not be segmented, hence this function is rather simple. If the data is larger than `N1` it will return the `DATA_TOO_LARGE` value and if the data is smaller than one byte it will return the `DATA_ERROR` value. If the data was valid an U-frame is generated, send it to the data-link module and a `DATA_OK` value is returned.

14.4.1 Test Specification

The functions in this module is tested using the white-box technique, to cover all paths through the flowchart in Figure **??** and through the flow in the `UnitDataRequest` function.

During the test of the functions the `N1` constant is set to 256, which is the default value.

The data is considered as arbitrary, hence only the `length` parameter is specified. During the tests the data will be verified, but the content of the data is not important, because it has no influence on the flow in the functions.

The outputs that will be tested upon are:

- The return value of the functions.
- The frames send to the data-link module.

In the test cases that produces frames the frames will be verified to have the content defined in section **??**.

`DataRequest`

To test all paths through this function the following test cases are used:

Test Case	**Input**	**Output**	**Purpose**
1.	$Length = 0$	Return DATA_ERROR.	To test that the function will return the right value if the length is less than 1 byte.
2.	$Length = 256$	Non-segment frame with the entire data, and return DATA_OK.	To test that the non-segmenting part of the function works.
3.	$Length = ((N1-1)\cdot 128)+1 = 32641$	Return DATA_TOO_LARGE.	To test that the function will return the right value when data is too large.
4.	$Length = 1018$	4 frames with 255 bytes of data in the first three and only 253 bytes in the last frame and return DATA_OK.	To test the segmenting part with a special last frame.
5.	$Length = 4\cdot 255 = 1020$	4 frames with 255 bytes of data in each and return DATA_OK.	To test the segmenting part when data fits in N1-sized frames.
6.	$Length = (N1-1)\cdot 128 = 32640$	128 frames with 255 bytes in each and return DATA_OK.	To test that the function will have the expected result even at the maximum length of the data.

`UnitDataRequest`

To test this function three test cases are used:

Test Case	**Input**	**Output**
7.	$Length = 0$	Return DATA_ERROR.
8.	$Length = 256$	Send U-frame to data-link and return DATA_OK.
9.	$Length = 257$	Return DATA_TOO_LARGE.

14.5 Reassembler

The reassembler module is responsible for receiving segments from the data-link module and reassemble them into data.

14.5.1 Interfaces

The interface to this module from the data-link module is the two functions:

- `char DataIndication(iFrame* frame)` - Used to reassemble segments send in connection-oriented mode, and send the reassembled data to the T55X layer.
- `char UnitDataRequest(uFrame* frame)` - Retrieves the data in an U-frame and sends it to the T55X layer.

The module declares the following constants:

- `DATA_OK=1` - Data was send successfully.
- `OUT_OF_MEMORY=2` - Allocating memory for the data failed.
- `DATA_ERROR=3` - Segments were received out of order.

The interface to the T55X layer is:

- `void receiveDataFromAX(char* start,int length)` - Receives data send in connection oriented mode.
- `void receiveConLessDataFromAX(char* start,int length)` - Receives data send in connection-less mode.

14.5.2 Data Structures

The input to this module is U- and I-frames where the PID and the first byte in the INFO-field is formatted as described in section **??**.

The module has four static variables:

- `char* buffer`:Pointer to a dynamically allocated buffer to hold the reassembled data.
- `int offset`:Indicates the offset (where to put the next segment) in the buffer.
- `char reassembling`:Used to detect errors if a start segment is received before all segments in the previous data block was received.
- `remain`:Indicates how many segments remain before a complete data block is fully reassembled. Along with the remain information in a segment this variable is used to detect if segments are received out of order (an error in the order of segments should not happen in the reassembler, but error detection is done to prevent the protocol from crashing if it happens anyway).

14.5.3 Functions

In what follows the functions in the module are described.

`DataIndication`

This function is responsible for retrieving data from I-frames and reassemble it into the original data block. Furthermore the function will also free the memory allocated for the data in the I-frames and detect errors in the reassembling process.

The flow in the function is illustrated in Figure 14.2.

In the following two examples will be used to describe the functionality of this function. The I-frames generated from the two examples in section **??**, will be used as input in the examples, hence `N1`=3 in these examples as well.

Example 1 In the first example the input is a frame, that has $PID = 0xF0$, which means that it is a single segment and the first decision box will branch to the left. The data is send to the T55X layer, the memory allocated for the frame is freed and the function returns `DATA_OK`.

Example 2 All the frames in the second example has $PID = 0x08$, hence the first decision box will branch to the right and the number of remaining frames is stored in `thisRemain`.

Segment 1: Recollecting that the INFO-field in the first frame was '131', '1', '2', gives that it is the first segment ($MSB = 1$ in 131) and `thisRemain` is 3 ($131 - MSB = 3$). The next decision box will branch to the decision box directly below it. If the function is already in the reassembling state (`reassembling=1`) the memory allocated for the old buffer is freed. The `reassembling` state is set, the `offset` variable is reset and memory for `buffer` is allocated to hold the entire data.

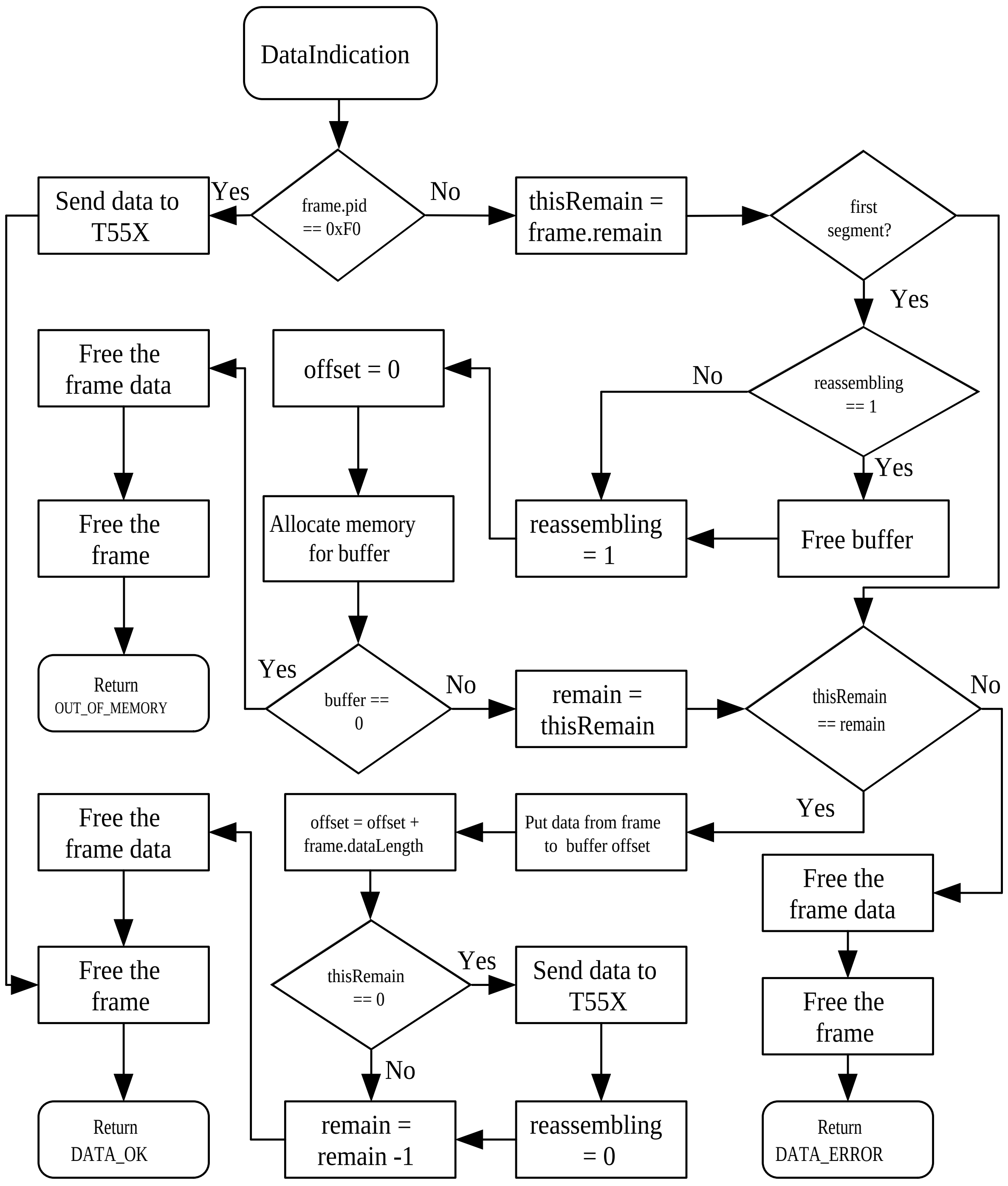


Figure 14.2: Flowchart of the DataIndication function

In this case the size of the buffer is:

$$(1 + \texttt{thisRemain}) * (N1 - 1) = 4 * 2 = 8$$

Even though the size of the original data is only 7 bytes the function needs to allocate 8 bytes, to comply with the worst case scenario where the last frame also contains 2 bytes, which is not known when the first segment is received.

If the function fails to allocate memory for the buffer (`buffer`=0) it will free the memory allocated for the frame and return a `OUT_OF_MEMORY` value.

When memory is allocated successfully the static `remain` variable is set to `thisRemain` to indicate how many segments are still missing in the buffer.

When reassembling the first segment the next decision box will always branch to the right where the data in the frame is copied to the buffer and the `offset` variable is updated according to the length of the data. The buffer and offset now looks like this:

1	2	0	0	0	0	0	0	`offset=2`

Since this is not the last segment (`thisRemain`!=0) the function will decrease the `remain` variable, free the memory allocated for the frame and return a `DATA_OK` value.

Segment 2 & 3: The next two segments come to the decision box where `thisRemain` is compared to `remain`. With the given input in this example this comparison is always true, but if for instance `thisRemain`=1 in the first of the two segments the comparison is false because `remain`=3-1=2 after the very first segment.

After the decision box the two segments are processed like the first segment and the buffer and the offset changes like this:

1	2	3	4	0	0	0	0	`offset=4`
1	2	3	4	5	6	0	0	`offset=6`

They will both return in the same way as the first segment.

Segment 4: When data is copied from the last frame to the buffer and the offset is updated they will look like this:

1	2	3	4	5	6	7	0	`offset=7`

This time `thisRemain`=0 and the function calls the `receiveDataFromAX` function in the T55X transport layer. The `offset` variable is used as the `length` parameter.

`UnitDataIndication`

This function retrieves data from U-frames and sends it to the T55X layer as connectionless data. After data is retrieved it will free the memory allocated for the data in the frame and return a `DATA_OK` value to the data-link layer.

14.5.4 Test Specification

The `DataIndication` function will be tested using the whitebox-technique to cover all paths through the flow chart in Figure 14.2. The `UnitDataIndication` function will be tested using the blackbox-technique, since it is has a simple sequential flow. Besides the test specifications for the two functions there is also a test specification for a test that will test the reassembler working togehter with the segmenter.

During the test of the functions the `N1` constant is set to 256, which is the default value.

For the same reasons as in the test of the segmenter, described in section 14.4.1, the data in the frames is considered as arbitrary.

There are two outputs that will be tested upon:

- The return value of the functions.
- The data send to the T55X transport layer.

`DataIndication`

In this test the varying input parameters are: PID, first byte in INFO-field and the order of the frames. In all test frames the length of the data is at its maximum, which is 256 bytes for single frames and 255 bytes for segmented frames.

To test all paths through this function the following test cases of the varying parameters are used:

- **Test Case 1:**

  Purpose: To test the part of the function that handles single segments.

  Input:

PID	First byte in INFO-field
0xF0	Data

  Output: Send 256 bytes of data to T55X transport layer and return `DATA_OK`.

- **Test Case 2:**

  Purpose: To test the part of the function that handles the first-, middle- and last segment, when segments come in the correct order.

  Input:

PID	First byte in INFO-field
0x08	131

PID	First byte in INFO-field
0x08	2

PID	First byte in INFO-field
0x08	1

PID	First byte in INFO-field
0x08	0

Output: Return `DATA_OK` three times, send $4 \cdot 255 = 1020$ bytes of data to T55X transport layer and finally return `DATA_OK` for the last frame.

- **Test Case 3:**

  Purpose: To test the part of the function that handles the situation where `reassembling` =1 when a first segment is received.

  Input:

PID	First byte in INFO-field
0x08	130

PID	First byte in INFO-field
0x08	1

PID	First byte in INFO-field
0x08	130

PID	First byte in INFO-field
0x08	1

PID	First byte in INFO-field
0x08	0

  Output: Return `DATA_OK` four times, send $3 \cdot 255 = 765$ bytes long data to T55X transport layer and finally return `DATA_OK` for the last segment.

- **Test Case 4:**

  Purpose: To test the part of the function that finds error in the order of segments.

  Input:

PID	First byte in INFO-field
0x08	131

PID	First byte in INFO-field
0x08	2

PID	First byte in INFO-field
0x08	0

Output: Return `DATA_OK` two times and then return `DATA_ERROR` for the last segment.

- **Test Case 5:**

  Purpose: To test the part of the function that handles memory allocation failures.

  Input: Any arbitrary frame with MSB in the first byte of the INFO-field set to '1' (memory is allocated when the first frame is received). To trigger a memory failure the function needs to be altered to do so in all cases.

  Output: Return `OUT_OF_MEMORY`

`UnitDataIndication`

- **Test Case 6:**

  Purpose: To test that the function sends the correct data to the T55X layer.
  Input: Any arbitrary U-frame with `dataLength`=256.

  Output: The 256 bytes of data is send to the T55X transport layer and return `DATA_OK`.

Segmenter and Reassembler

The purpose of this test is to do a blackbox test of the segmenter and reassembler working together. From a test program, data with varying length is send to the segmenter and send on to the data-link module in frames. From the data-link module the frames is send back to the reassembler where the reassembled data is send back to the test program. When data is received back at the test program it should be compared to be the same as the data send to the segmenter.

14.6 Data-link

The data-link state machine described in AX.25 is the part of the protocol that carriers out a lot of the essential data-link layer functionalities - e.g. flow control, retransmission, acknowledgment etc. The data-link state machine uses the link multiplexer in order to transmit data.

14.6.1 Interfaces

The interfaces available to upper layers are:

- `char DLDataRequest(iFrame* frame)` - Used to send data in connected mode received from the segmenter.
- `char DLUnitDataRequest(uFrame* frame)` - Used to send data connection-less received from the segmenter.
- `void DLConnectRequest()` - Used by upper layer to request a connection.
- `void DLDisconnectRequest()` - Used by upper layer to request termination of a connection.

The interface to the reassembler is:

- `void DLDataIndication(iFrame* frame)` - Used by the data-link module to send incoming data (in connected mode) to the reassembler.
- `void DLUnitDataIndication(uFrame* frame)` - Used by the data-link module to send incoming connection-less data to the reassembler.

The interface to the T55X layer is:

- `void DLErrorIndication(char error)` - Handles errors indicated by the data-link state machine. The `error` parameter indicate the type of error.
- `void DLConnectConfirm()` - Used by the data-link module to indicate to the T55X layer that AX.25 is now connected.
- `void DLDisconnectConfirm()` - Used to indicate to the T55X layer that AX.25 is now disconnected after a `DLDisconnectRequest`.
- `char void DLDisconnectIndication()` - Used to handle a disconnection that was not initiated by the T55X layer - e.g. if the peer disconnected or the connection was lost.
- `char void DLConnectIndication()` - This function is invoked if the peer requests an connection.

14.6.2 Data Structures

Several data structures are used in the data-link state machine. Most of them are simple definitions used to make the program more simple to read.

Error codes

- `DL_ERROR_A` - A frame has been received with F bit set to 1, but no frame has been sent with P bit set to 1.
- `DL_ERROR_C` - An unexpected UA-frame has been received.
- `DL_ERROR_D` - An UA-frame has been received without F=1 when a SABM or DISC frame was sent with the P bit set to 1.
- `DL_ERROR_E` - Unexpected DM-frame received.
- `DL_ERROR_F` - Indicates that data-link was reset
- `DL_ERROR_G` - RC to big in awaiting connection state
- `DL_ERROR_H` - RC to big in awaiting release state
- `DL_ERROR_I` - `N2` timed out - unacknowledged data
- `DL_ERROR_J` - `NR` sequence error
- `DL_ERROR_K` - Invalid frame received
- `DL_ERROR_O` - I-frame exceeded maximum allowed length
- `DL_ERROR_Q` - UI-response received, or UI-frame received with P bit set to 1
- `DL_ERROR_S` - I-response received
- `DL_ERROR_T` - `N2` timed out - no response to enquiry
- `DL_ERROR_U` - `N2` timed out - extended peer busy condition

Internal Variables

The data-link state machine consists of several internal variables that describes the current state of the state machine. The most important of them are mentioned below:

- `SRT` - Smoothed round trip time value
- `layer3Initiated` - Has layer 3 tried to initiate communication? Important when receiving confirmation from the peer, of an establishment of the data-link.
- `dlState` - The current state of the data-link state machine (e.g. disconnected, awaiting connecting, awaiting release, connected or timer recovery).
- `rejectException` - Has a reject exception been raised?
- `SREJEnabled` - Is selective reject enabled?

- `SRejectException` - Has a selective reject been raised?
- `peerRecieverBusy` - Is the peer receiver busy?
- `ownRecieverBusy` - Is this terminals receiver busy?
- `acknowledgePending` - Are any acknowledges pending?
- `VS` - Contains the next sequential number to be assigned to the next transmitted I frame.
- `NS` - Contains the sequence number of the I frame being sent.
- `VR` - Contains the sequence number of the next expected received I frame.
- `NR` - Contains the sequence number of the most recently received I or S frame.
- `VA` - Contains the sequence number of the last frame acknowledged by the peer.

I-Frame Queue

The data-link state machine uses an internal queue in order to be able to handle more than one I-frame at the time. The queue is a dynamic FIFO queue implemented as a doubly linked list. The queue has two functions:

- `void insert(listItem* *head, iFrame frame)` - Inserts a new iFrame in the list. The `head` parameter is a pointer to the first listitem in the linked list.
- `iFrame* pop(listItem* *head)` - Returns and removes the oldest iFrame in the list.

14.6.3 Functions

This section will describe the functions in the module.

`DLDataRequest`

This function receives I-frames from the segmenter and if possible transmit these frames using the link multiplexer. The function returns no value, so the sender has no insurance that the data will reach its goal. Confirmation of data being send should be implemented in upper layers. If the frame is lost during transmission, it will if possible be retransmitted. If an error-indication is received by the T55X layer (e.g. `DL_ERROR_I`) this can implicitly be considered as a failure of sending the data. However silence (no error indications) can not be considered as a confirmation that the data was send.

`DLUnitDataRequest`

This function sends data connection-less, meaning it just transmits the frame, and afterwards forget about it. It expects that the input data is already segmented into the uFrame format. If the data doesn't reach the peer, a retransmission is not invoked.

`DLConnectRequest`

This function attempts to establish a data link. It does not return any values, but if the attempt is successful the function `DLConnectConfirm()` will be invoked. If a data-link can not be established within a certain amount of time, a re-request will occur. After `N2` re-requests an `DL_ERROR_INDICATION(G)` will be received, and the function `DLDisconnectIndication()` will be invoked. If the request is successful the peer will receive a `DLConnectIndication()`.

`DLDisconnectRequest`

Requests a disconnection of the data link. This will delete any data that is not transmitted yet. No value will be returned by the function, but the function `DLDisconnectConfirm()` is invoked when the data-link has been destroyed.

The peer will receive a `DLDisconnectIndication()`.

14.6.4 Internal Functions

This section describe the internal functions of the module.

`void startT1`

Used to start the `T1` timer.

`void stopT1`

Used to stop the `T1` timer.

`void startT3`

Used to start the `T3` timer.

`void stopT3`

Used to stop the `T3` timer.

`void T1Expiry`

This function is invoked by the data-link state machine whenever the T1 timer has been started and the time interval defines for this timer has passed. Depending on the current state of the data-link state machine different actions will be taken according to the AX.25 specifications.

`void T3Expiry`

This function is similar to `void T1Expiry`. The only difference is that it is linked to the timer T3.

`void UICheck`

This function is invoked when UI-frames are received. The function does not return any values, but if there are any errors in the UI-frame it invokes `DLErrorIndication()`. If the UI-frame is correct `DLUnitDataIndication` is invoked.

`void checkNeedForResponse`

This function is used to determine if a response is needed whenever a frame is received. If the incoming frame is a frame with the p bit set to 1 a response is needed. If this is the case `enquiryResponse` is invoked. If a response frame with the p bit set to 1 is received `DLErrorIndication` is invoked.

`void clearExceptionConditions`

Resets the following variables - `peerRecieverBusy`, `ownRecieverBusy`, `rejectException` and `acknowledgePending`.

`void establishDataLink`

Used to establish a data-link. It creates a SABM-frame and sends it to the link-multiplexer. Also is stops T3, and starts T1 in order to be able to retry to establish the data-link if too much time passes.

`void selectT1`

This function is used to determine the next value of the timer T1. The value is determined from the value of `RC`.

`checkIFrameAcked`

Used by the data-link state machine to determine which actions should be taken upon reception of I frames. It determines which timers should be restarted and if new values should be selected. Also it updates the value of `VA`.

`void NRErrorRecovery`

When an acknowledgment error that can not be recovered occurs AX.25 solves this by resetting the data-link and reconnects. This function is responsible for indicating to upper layers through `DLErrorIndication` that a `NR` error has occurred. Afterwards the function invokes `establishDataLink`.

`void invokeRetransmission`

Invoked whenever retransmission of frames is necessary.

`void lmIFrameDataIndicate`

This function is always invoked by the link multiplexer. It is used by the data-link state machine to handle incoming I-frames. The function acts differently depending on the current state of the data-link, according to the specifications in the AX.25 protocol.

`void lmSFrameDataIndicate`

Similar to `void lmIFrameDataIndicate` - the only difference is that it handles S-frames instead of I-frames.

`void lmUFrameDataIndicate`

Similar to `void lmIFrameDataIndicate` - the only difference is that it handles U-frames instead of I-frames.

`void LMSeizeConfirm`

Invoked by the link multiplexer whenever the data-link state machine is allowed to seize control of the radio.

`iFrame createIFrame`

The function is used to allocate memory for a new I-frame and fill out the following fields: Source, destination, NS, NR, pbit, data, dataLength and the type of command.

`uFrame createUFrame`

This function is used to allocate memory for a new U-frame and fill out the following fields: Source, destination, frame-type, command-type, data and data-length. All the available frame types are described later in this chapter. The command-type indicates whether it is a command or response frame.

`sFrame createSFrame`

This function is used to allocate memory for a new S-frame and fill out the following fields: Source, destination, NR, frame-type, pbit and commandtype.

`void enquiryResponse`

This function is used whenever a command frame is received with the pbit set to 1. This means that the terminal have to response to the peer. The function sends back a RR-frame or RNR-frame to the peer - depending on the terminals own reciever is busy or not.

`char pBit`

The function returns the value of the p/f bit in the control field of a frame.

`char frameType`

The function returns whether the frame is a command/response frame. If the frame is not any of these, it returns an error.

14.6.5 Test Specification

The data-link module will be tested using the white-box-technique. It should be noted that testing the module by only trying all the inputs to the c-functions is not enough since most of the functions depends on external factors - such as the state of the data-link machine, `NR` etc. In the test specification these external factors will be considered as input to the function.

Since the c-functions rarely have any output but rather invoke other functions and/or change the state of the data-link, the output in the specification is the expected function calls and the state of the data-link right after the function has been called.

Since the amount of inputs to the functions are relative limited, the black-box test tries to cover all possible inputs.

Because of the extend of this test, the specification is available on the project CD-ROM in the two files `testspec.ps` and `testspec.pdf` in the /test/datalink/ library.

Chapter 15

T55X

The T55X layer has not been implemented due to lack of time, but the initial steps in the implementation process are covered in this chapter.

15.1 Overview

This section will provide an overview of the implementation of the T55X. Because of the limited resources on the satellite the software has has to be minimized. One thread should be capable of controlling the T55X layer. The SDL state machine TTC has been chosen to be the thread while the other two state machines are function libraries. The interfaces to other layers are shown in Figure 15.1.

15.2 Modifications

This section discusses the modifications that are necessary to implement the system. The system should follow the SDL model as close as possible, meaning that the functionality of the state machines must be kept intact. In fact some functionalities that were left out from the model verification is once again included.

The design of the structure of the signals in the SDL model, has been created using ObjectGEODE's capability to manage data types. An example is that there never had to be used pointers to describe data and buffers. The frame structure for signals has to be changed so that it always informs where the data is resident in memory, with a pointer and length instead of the ObjectGEODE data field.

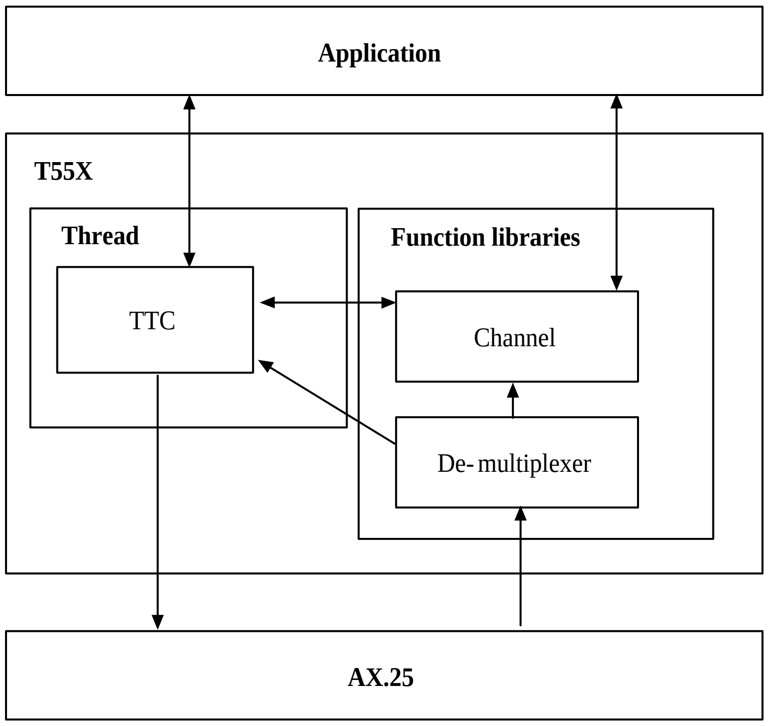


Figure 15.1: An overview of the interfaces and the task of the T55X layer

15.2.1 Channel

Functions which were left out of the design, but will be reintroduced in the implementation for the channel are:

- Left To Transmit
- Elements in Transmit Buffer
- Flush Entire Transmit Buffer
- Flush the First Element in Transmit Buffer
- Data in Receive Buffer

15.2.2 TTC

A single function which was left out of the design of the TTC will be reintroduced in the implementation of the TTC. This function is:

- Resume connection

15.3 Test Specifications

The test specification could be ported and reused from the design test specification. The main difference is that functions are used instead of signals. The reason that it is possible to reuse the test specification is that nothing should have been changed during the implementation.

Chapter 16

Test

This chapter documents the tests conducted on the implemented system accroding to the test specifications.

16.1 Segmenter

This section describes the tests performed on the segmenter module according to the test specifications stated in section 14.4.1. The source code used and the test results can be found on the project CD-ROM in folder “test/segmenter/”.

Both functions are tested with a test program (represents T55X module) and a test data-link module.

16.1.1 Test Program

The pseudo code for the test program is:

```
main(){
  Arraytype data
  data.length = 32768
  data = [0,1,2,3....254,255,0,1,2,3.....253,254,255]
  print "********* Test case 1 *********"
  print "Length"+test case 1 length
  i = DataRequest(data, test case 1 length)
  print "Return value: "+i
  .
  .
  .
  print "********* Test case n *********"
  print "Length"+test case n length
  i = DataRequest(data, test case n length)
  print "Return value: "+i
}
```

The `DataRequest` call can also be a `UnitDataRequest` call.

16.1.2 Test Data-Link Module

The `DLDataRequest` function called from the segmenter in the data-link module has the following pseudo code:

```
print "********* Frame number *********"
print "PID: "+frame.pid
print "Length: "+frame.dataLength
print "First byte: "frame.data[0]
print the info field
```

The `DLUnitDataRequest` function has the same pseudo code, except it will not write out PID and first byte information.

16.1.3 Result

All tests were conducted and they all had the expected result, which ensures that all paths through the functions works according to the specification.

An example of the test output for test case 2 looks like this:

```
************ Test Case 2 ************
Length=256
************ Frame 1 *************
PID: 240
Length: 256
First byte: 0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83
84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102
103 104 105 106 107 108 109 110 111 112 113 114 115 116 117
118 119 120 121 122 123 124 125 126 127 128 129 130 131 132
133 134 135 136 137 138 139 140 141 142 143 144 145 146 147
148 149 150 151 152 153 154 155 156 157 158 159 160 161 162
163 164 165 166 167 168 169 170 171 172 173 174 175 176 177
```

```
178 179 180 181 182 183 184 185 186 187 188 189 190 191 192
193 194 195 196 197 198 199 200 201 202 203 204 205 206 207
208 209 210 211 212 213 214 215 216 217 218 219 220 221 222
223 224 225 226 227 228 229 230 231 232 233 234 235 236 237
238 239 240 241 242 243 244 245 246 247 248 249 250 251 252
253 254 255

Return value: 1
```

All the results can be found in the `testfile` file in directory: /test/segmenter/ on the project CD-ROM.

16.2 Reassembler

This section describes the tests performed on the reassembler module according to the test specifications stated in section 14.5.4. The source code used and the test results can be found on the project CD-ROM in folder “test/reassembler/”.

Both functions are tested with a test program (represents data-link module) and a test T55X module.

16.2.1 Test Program

The pseudo code for the test program is:

```
main(){
  print "********* Test case 1 *********"
    print Test case 1 purpose
    print "Frame number 1"
    print "Length= "+Test case 1, Frame number 1 length
    print "PID= "+Test case 1, Frame number 1 PID
    print "First byte in info= "+Test case 1, Frame number 1 info[0]
    frame=new frame
    put values in frame
    returnvalue=DataIndication(frame)
    print "Return value= "+returnvalue
    .
    .
    .
    print "Frame number n"
    print "Length= "+Test case 1, Frame number n length
    print "PID= "+Test case 1, Frame number n PID
    print "First byte in info= "+Test case 1, Frame number n info[0]
    frame=new frame
    put values in frame
    returnvalue=DataIndication(frame)
    print "Return value= "+returnvalue
  .
  .
  .
  print "********* Test case n *********"
    print Test case n purpose
    print "Frame number 1"
    print "Length= "+Test case n, Frame number 1 length
    print "PID= "+Test case n, Frame number 1 PID
    print "First byte in info= "+Test case n, Frame number 1 info[0]
    frame=new frame
    put values in frame
    returnvalue=DataIndication(frame)
    print "Return value= "+returnvalue
    .
    .
    .
    print "Frame number n"
    print "Length= "+Test case n, Frame number n length
    print "PID= "+Test case n, Frame number n PID
    print "First byte in info= "+Test case n, Frame number n info[0]
    frame=new frame
    put values in frame
    returnvalue=DataIndication(frame)
    print "Return value= "+returnvalue
}
```

16.2.2 Test T55X Module

The `receiveDataFromAX` function called from the reassembler in the T55X module has the following pseudo code:

```
print "********* Receiving connection oriented data *********"
print "Length: "+length
print "Data: "+print out the data
```

The `receiveConLessDataFromAX` function has the same pseudo code, except from the first line:

```
print "********* Receiving connection less data *********"
```

16.2.3 Results

The test program was executed and all tests had the expected output according to the test specification. It implies that all paths through the functions works according to the specification.

An example of the test output for test case 1 looks like this:

```
************************ Test Case 1 ************************
Single Segment

Frame number 1:
Length=256
PID=0xF0
First byte in info is data
********** Receiving connection oriented data **********
Length: 256
Data:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109
110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126
127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160
161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177
178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194
195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211
212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228
229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245
246 247 248 249 250 251 252 253 254 255

Return Value=1
```

The output from the entire test is found in the `testfile` file in the /test/reassembler/ directory on the project CD-ROM.

16.3 Segmenter and Reassembler

This section documents the test conducted on the segmenter and reassembler working together as specified in section 14.5.4.

The modules are tested using a test data-link module and a test T55X module.

16.3.1 Test Data-link

The `DLDataRequest` function in this module calls the `DataInidication` function in the reassembler module with the same parameter as it received itself. The code is:

```
char DLDataRequest(iFrame* frame) {
  DataIndication(frame);
}
```

16.3.2 Test T55X

This module is responsible for sending, receiving and comparing data. The `main` function in the module has the following pseudo code:

```
Arraytype data;
data.length = 32768;
data = [0,1,2,3....254,255,0,1,2,3.....253,254,255];
loop 20000{
  len=random number between 0 - 32768;
  i=DataRequest(data,len);
  if (i==DATA_TOO_LARGE)print("Length: "+len+" Data was too large");
  if (i==DATA_ERROR)print("Length: "+len+" Data<1");
}
```

The loop is limited to 20000 iterations to limit the size of the testfile.

The pseude code for the `receiveDataFromAX` function, that receives data from the reassembler, is:

```
void receiveDataFromAX(char* start,int length){
  if (len==length){
    correct=2;
    loop length times
      if(tgs[j]!=start[j])correct=1;
  }
  else
    correct=0;
  printf("Length: "+len+" Correct: "+ correct);
}
```

With the given pseudo code the value of `correct` means:

- `correct`=0 - The length of the data was not correct.
- `correct`=1 - The data was not correct.
- `correct`=2 - The length and the content of the data was correct.

16.3.3 Results

The test was conducted and all 20000 test cases had the expected result, which raises the probaility that the implementated modules are correct according to their specification.

And example of 10 test outputs looks like this:

```
Length: 22529 Correct: 2
Length: 7704 Correct: 2
Length: 829 Correct: 2
Length: 27735 Correct: 2
Length: 14860 Correct: 2
Length: 31311 Correct: 2
Length: 32719 Data was too large
Length: 29186 Correct: 2
Length: 12811 Correct: 2
Length: 17962 Correct: 2
```

The test was also conducted where the loop was not limited to 20000 but iterated infinitely. And the test program was set to halt if `correct` was different from 2.

16.4 Data-link

This section will describe the test performed on the data-link module according to the test specifications in section 14.6.5. The source code for the test-program can be found on the project CD-ROM in folder "test/datalink".

16.4.1 Test Functions

To carry out the blackbox test, some extra functions are neccesary. Also some macros are defined in order to print out test results. These are described below:

Macros

- STOP_T1 - Prints that timer 1 has beens stopped.
- START_T1 - Prints that timer 1 has beens started.
- STOP_T3 - Prints that timer 3 has beens stopped.
- START_T3 - Prints that timer 3 has beens started.
- DLSTATE - Prints the currenct value of dlState.

- IFRAME_PUSHED_ON_QUEUE - Prints that an I-frame has been pushed onto the queue.
- DLCONNECT_CONFIRM - Prints that `DLConnectConfirm()` has been invoked.
- DLCONNECT_INDICATION - Prints that `DLConnectIndication()` has been invoked.
- UNIT_DATA_INDICATE - Prints that `DLUnitDataIndication()` has been invoked.
- PRINT_ERROR - Prints which error that has occoured when `DLErrorIndication()` has been called with.
- INVOKE_RETRANSMISSION - Prints that `invokeRetransmission()` has been invoked.
- DLDISCONNECT_INDICATION - Prints that `DLDisconnectIndication()` has been invoked.
- DLDISCONNECT_CONFIRM - Prints that `DLDisconnectConfirm()` has been invoked.

Functions

- `void setAcknowledgePending(char val)` - Sets the value of `acknowledgePending`.
- `void setNR(char val)` - Sets the value of `NR`.
- `void setOwnRecieverBusy(char val)` - Sets the value of `ownRecieverBusy`.
- `void setPeerRecieverBusy(char val)` - Sets the value of `peerRecieverBusy`.
- `void setRC(char val)` - Sets the value of `acknowledgePending`.
- `void setVS(char val)` - Sets the value of `VS`.
- `void setVA(char val)` - Sets the value of `VA`.
- `void setLayer3Initiated(char val)` - Sets the value of `layer3initiated`.
- `void setNS(char val)` - Sets the value of `NS`.
- `void setVR(char val)` - Sets the value of `VR`.
- `void printDlState()` - Prints the current state of the data-link state machine.
- `void setDlState(char state)` - Sets the state of the data-link state machine.

16.4.2 Test Program

An example of the test program for test case number 18 and 19 can be seen here:

Test code

```
  /* SABM frame test - start */
  printf("*** Test %d ***\n", testNum);
  printf("Recieving SABM command frame pf bit = 0 \n
          Expected result: UA_F0 (Expidited) +
          DL_CONNECT_INDICATION + T3 started +
          state -> CONNECTED\n\nResult:\n");
  setDlState(DISCONNECTED);
  lmUFrameDataIndicate(createUFrame(source, dest,
                                    SABM_P0, COMMAND_FRAME,
                                    0,N1));
  printDlState();
  printf("Recieving SABM command frame pf bit = 1 \n
          Expected result: UA_F1 (Expidited) +
          DL_CONNECT_INDICATION + T3 started +
          state -> CONNECTED\n\nResult:\n");
  setDlState(DISCONNECTED);
  lmUFrameDataIndicate(createUFrame(source, dest,
                                    SABM_P1, COMMAND_FRAME,
                                    0,N1));
  printDlState();
  printf("*** End Test %d ***\n",testNum);
  testNum++;
  /* SABM frame test - end */
```

Output

Below the output from the test code is shown:

```
*** Test 5 ***
Recieving SABM command frame pf bit = 0
Expected result: UA_F0 (Expidited) + DL_CONNECT_INDICATION + T3
started + state
-> CONNECTED

Result:
Uframe (UA_F0) (RESPONSE_FRAME) expidited unit data request
DL-CONNECT-INDICATION
Started T3
```

```
State: CONNECTED
Recieving SABM command frame pf bit = 1
Expected result: UA_F1 (Expidited) + DL_CONNECT_INDICATION + T3
started + state
-> CONNECTED

Result:
Uframe (UA_F1) (RESPONSE_FRAME) expidited unit data request
DL-CONNECT-INDICATION
Started T3
State: CONNECTED
*** End Test 5 ***
```

As it can be seen from these results, the function behaves as expected with the given input. Since the test is very long please refer to the `testfile` file on the project CD-ROM in folder "test/datalink/".

Chapter 17

Summary

This chapter describes the status of the implementation of AX.25 and T55X. Which parts are finished and which parts are missing.

17.1 AX.25

The following modules have been implemented and tested:

- Segmenter
- Reassembler
- Data-link

The only thing missing from these modules are the details about the timers. The functionality is implemented, but the actually timers are missing.

The following modules have not been implemented:

- Management data-link - The values handled by this state machine are defined as constants.
- Link multiplexer - Functions has been defined in order to maintain compatibility with data-link state machine.
- Physical state machine - Not implemented at all due to a decrease in group members.

Since all modules have not been implemented a test of the total system has not been done. Furthermore the implementation has been done on a Solaris-platform. The platform on the CubeSat will have both different operating system and different hardware. The implemented parts of AX.25 has not been tested on this platform.

The following things would need to be done in order to complete the implementation:

- Implement management data-link, link multiplexer and physical state-machine.
- Implement timers with the facilities provided by the operating system on-board the satellite.
- Port implementation to the on-board computer on CubeSat
- Integrate all the modules with each other.
- Test the total system.

17.2 T55X

The T55X layer has not been implemented, but the structure of a potential implementation has been analyzed and described.

Part V

Conclusion

This part covers the status of the project - which parts of the communication subsystem for the CubeSat project are done and which ones are incomplete. This will be useful for a potential new project group, that would finish the CubeSat communication subsystem. Finally it will describe the development process in this project.

Chapter 18

Project Status

The original purpose of having a complete communication subsystem ready by the end of this project period has not been reached, but the development is well under way. This section describes which parts of the original purpose that is fully developed and which have not been developed yet.

18.1 Completed

This section describes the tasks that have been completed.

Hardware

A supplier of the communication hardware was found (One Stop Satellite Solution - OSSS). This company is still in the process of developing the hardware, meaning that the hardware has not been purchased; but specifications for the modem have been available and analyzed. The analysis of the hardware included a link budget for the link between the ground station and the satellite.

Requirements and Interfaces

In cooperation with the other CubeSat project groups, a requirement specification for the communication software has been established. The main focus of this requirement specification was to establish a well defined interface to the communication subsystem.

AX.25 Protocol

An existing data-link protocol AX.25 was chosen as a basis for the communication protocol. AX.25 is a protocol widely used by the amateur radio community and several other CubeSat projects. The protocol provides the necessary functionality needed for this project, and supports half-duplex communication as needed by the hardware.

The segmenter and reassembler modules in the AX.25 protocol have been implemented using the C programming language on a Solaris / SPARC platform. The data-link module has also been implemented on this platform, but the timers in this module have not been implemented. The functions to invoke when timers expire have been implemented.

The implemented parts of AX.25 have been tested using white-box and black-box testing techniques. Test specifications were written and tests were conducted according to them. All tests had the expected result, hence the implementation is considered as correct according to the test specifications.

T55X Layer Protocol

Furthermore a combined transport and session layer, named T55X has been specified. This protocol has been designed and modeled in SDL using the ObjectGEODE tool. The purpose of using ObjectGEODE was to verify and validate the protocol.

Simulations has been conducted on the model and all errors detected in these tests have been corrected. Furthermore a planned verification of the model proved impossible to conduct as the model was to complex. For the same reason validation was not conducted either.

The T55X layer has not been implemented, but the structure of a potential implementation has been analyzed and described.

18.2 Pending Items

This section lists the parts of the project that are incomplete and needs to be done to have a complete communication system.

Hardware

- Consider other possible solutions than the one from OSSS or go ahead and buy the hardware from OSSS.
- Integrate the hardware with the other subsystems and with the mechanical structure.
- Purchase ground station hardware: Antenna, transceiver, rotor etc.
- Test the hardware.
- Design and implement a micro controller for the emergency beacon.

To complete these tasks it will of course require that a complete hardware solution is bought or developed. When the final hardware is ready it would be desirable to have a person, with experience in integrating and testing this type of hardware, to take over.

The micro controller for the emergency beacon will probably be a PIC similar to one of those used in the other subsystems, because they have already been tested for space. While the functionality is rather simple, completing this part should not be very time consuming.

AX.25 Protocol

- Complete implementation of data-link module.
- Port the currently implemented parts to the platform on board the satellite.
- Implement physical layer including device drivers.
- Combine all modules into a complete protocol.
- Test the complete protocol.

Our experience is that the documentation of the AX.25 is not very precise, hence this task could be very time consuming.

T55X Layer Protocol

- Simplify the model to complete design verification and validation.
- Implement the protocol.
- Test the implementation.

Along with the implementation of the AX.25 these tasks will be where most of the future work is needed.

Integrating and Testing the Complete System

- Combine AX.25 and T55X.
- Test the complete communication system.
- Integrate the communication protocol with the data handling system.

Our experience is that integrating and testing larger systems can be very complicated and time consuming. For instance, different interpretation of the interfaces can lead to a lot of problems.

Chapter 19

Project Course

This chapter describes the course of the project by looking at the development process, educational achievments and finally a retrospective view.

19.1 Development Process

When the project started the intention was to have a fully operational communication subsystem ready by the time this report was due. Since no student groups from the Department of Communication Technology have been involved in the AAU CubeSat project, it was decided to purchase the communication hardware from One Stop Satellite Solution (OSSS). At the time when this report was delivered, OSSS was still working on the hardware. During the project period much time and effort have been put into communication with OSSS about the hardware. But since OSSS is still working on the hardware, full specifications have not been available.

Since a lot of student groups have been working on the AAU CubeSat project, intensive communication between the groups has been necessary. Deciding who should be responsible for which parts of the satellite, including specific parts of the individual subsystems, took quite some time at the beginning of the project. After the responsibilities were settled, the interfaces to other subsystems had to be agreed upon. With two groups working on the communication software the work had to be divided. At first the two groups worked together on everything to ensure a common understanding and general agreement on what had to be developed. Later in the project the work was divided the following way:

- **01gr554** - Modeling and verification of the AX.25 protocol. Implementation of the software on the ground station.
- **01gr555** - Modeling and verification of the T55X layer. Implementation of the software on the satellite.

Along the way a lot of modifications have been made to the original design ideas, mainly due to feedback from other groups. Thus the design phase has taken considerably longer time than first expected. Furthermore the actual design was started later than expected, since much effort was put into discussing the needs, interfaces etc. with other groups. The first part of the project was furthermore delayed by the fact that the groups involved in the AAU CubeSat project was not used to this type of project. Most student projects at AAU do not require cooperation with several other groups, therefore some time was spent adjusting to this and finding common ground.

The design part of this project only included the T55X layer, since the AX.25 protocol was predefined. This meant that the implementation of AX.25 could be done in parallel with the design of T55X. In the end the implementation also suffered from the fact that a member of the group chose to take leave from his studies.

19.2 Educational Achievements

The bottom line is, the project period has definitely not been as we expected, but it has never the less been very educational. The main subjects we have obtained knowledge about are:

- Aerospace engineering.
- Wireless communication.
- Communication protocols.
- SDL for modeling and validation.
- Real life engineering project.

As part of the AAU CubeSat project we attended eight courses in aerospace engineering, such as: Space environments and orbit, spacecraft engineering, communication system etc.

Finding possible solutions for the hardware part of the system, necessitated that we studied the fundamentals in wireless communication, e.g. link budget.

One of the main parts in the project was to analyze, design, model, validate, implement and test communication protocols. More specifically the AX.25 data-link protocol was analyzed, implemented and tested. A higher level protocol, called T55X, was analyzed, designed, modeled and to some extend validated in SDL.

The tool used for SDL modeling and validation was ObjectGEODE. Understanding the syntax and functionality of this tool has also been one of the main parts in the project.

Being part of a real life engineering project has been a big challenge for us. We only had experience in working in groups up to seven people working closely together on the same specific project. The AAU CubeSat project involves about 60 people working on

9 different projects. Agreeing on simple things like mission took a whole month, and more specific items as interfaces was also not an easy task.

19.3 Retrospective View

Looking back at the development process, there are things that should have been done differently:

- Department of Communication Technology at AAU (KOM) should have been affiliated to the project.
- Better analysis of the data-link protocols, before deciding on AX.25.
- Validating an old and widely used protocol as AX.25 was a bad choice.
- Better organization in the CubeSat project.

It turned out that buying wireless communication hardware was not an easy task. Buying the system still requires knowledge about all the parameters for such a system. Our qualifications on this area were simply not sufficient, hence too much time was spent on this task. Furthermore the other subsystems had requirements to the communication subsystem which we had no control over. A solution to these two problems is to have a group from the KOM department to be responsible for the communication hardware.

The AX.25 data-link protocol was among other things chosen because it seemed well documented. However it turned out that the documentation was not very precise on some areas. If the project was to be done again we would consider to design a simpler data-link protocol.

Since the documentation of the AX.25 includes SDL diagrams it was decided that group 554 should model and validate the protocol in ObjectGEODE. However it turned out that converting the SDL diagrams in the AX.25 documentation to SDL in ObjectGEODE was almost a project in it self. Retrospectively it was a bad choice because AX.25 is an old an widely used protocol, hence it should not be necessary to model and validate it for a development project.

We find the organization in the CubeSat project a bit too unstructured. It was the intention that the students should feel that they decided everything on their own. This can of course raise the motivation but at the same time it can be frustrating that everybody has to agree on everything. A solution to this problem is to take more decisions from the top when things get out of hand.

Part VI

Appendices

This part contains the appendices which are: Calculations for the link budget, description of the AX.25 protocol, description of ObjectGEODE and a description of the OSI-model

Appendix A

Link Budget

A.1 Probability of Error

To calculate the probability of bit error an equation known as Ebno is used. The Ebno equation is:

$$E_b/N_0 =$$

$$EIRP + G/T + 196.15 - 20 \cdot log(d/1\ \text{km}) - 20 \cdot log((f/1\ \text{MHz}) - 10 \cdot log(B/1\ \text{Hz}) \tag{A.1}$$

EIRP, also known as "Equivalent Isotropically Radiated Power", is the power required by the transmitter output for a antenna that is omni directional such as the CubeSat antenna. EIRP is calculated by the following equation:

$$EIRP = P_t + G_t \tag{A.2}$$

Where P_t is the transmitting power and G_t the transmitter gain. The CubeSat antenna is an isotropical antenna with an estimated -3 dBW gain, and has 2 Watts of transmitting power, corresponding to a P_t value of 3 dBW. This gives us that

$$EIRP = 3\ \text{dB} - 3\ \text{dB} = 0\ \text{dB} \tag{A.3}$$

G/T is a measurement of the quality factor performance of the receiver. G/T is known by this equation:

$$G/T = G_r - 10 \cdot log(T/1\ \text{K}), \tag{A.4}$$

where G_r is the gain of the receiving antenna and T is the system noise temperature. The yagi antenna currently used have a gain of 16 dB, but to this we can add another

3 or 6 dB respectively if we use 2 or 4 yagi antennas in an array. The system noise temperature is 251 K, giving the following result with a single yagi antenna:

$$G/T = 16 \text{ dB} - 10 \cdot log(251 \text{ K}/1 \text{ K}) \text{ dB} \approx -8 \text{ dB} \tag{A.5}$$

$20 \cdot log(d/1 \text{ km})$, $20 \cdot log(f/1 \text{ MHz})$ and $10 \cdot log(B/1 \text{ Hz})$ are values to be subtracted, to take into account the distance d, the frequency f and the bit rate B, who all decreases the quality of the link. The bit rate B on the modem supplied to us by OSSS is 9600 bps, the frequency f is 433 MHz and the distance d is calculated using Pythagoras as shown in Figure A.1.

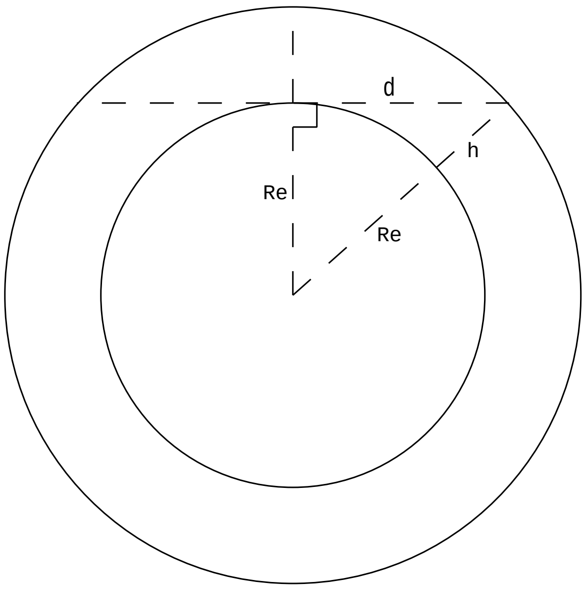


Figure A.1: Calculation of the maximum distance to the satellite is done using Pythagoras. Re is 6378 km and h is 600 km. This gives that $d = \sqrt{(Re+h)^2 + Re^2} = \sqrt{2 \cdot Re \cdot h + h^2} = \sqrt{2 \cdot 6378 \cdot 600 + 600^2} \approx 2830$ km

The values are:

$$20 \cdot log(433 \text{ MHz}/1 \text{ MHz}) = 52.7 \text{ dB}$$

$$10 \cdot log(9600 \text{ Hz}/1 \text{ Hz}) = 39.8 \text{ dB}$$

$$20 \cdot log(2830 \text{ km}/1 \text{ km}) = 69.0 \text{ dB}$$

These values are all to be subtracted from the other values and we can calculate the E_b/N_0 equation:

$$E_b/N_0 = 0 \text{ dB} - 8 \text{ dB} + 196.15 \text{ dB} - 69.0 \text{ dB} - 52.7 \text{ dB} - 39.8 \text{ dB} \approx 26.7 \text{ dB} \tag{A.6}$$

If setting up two or four antennas in an array we can add 3 or 6 dB, giving 29.7 dB and 32.7 dB respectively, but since the E_b/N_0 equation already gives a relatively high value this should not be necessary. A E_b/N_0 value of approximately 13 dB gives a probability of bit error around 10^-13. If it was possible to adjust the bit rate and transmitting power on the cubesat, we could use the link budget to adjust these to our liking and still have a E_b/N_0 value below 10 dB, which gives a probability of bit error below the 10^{-5} that Flemming Hansen has recommended. As this is not possible, the only use of the E_b/N_0 equation is to ensure that we are below the recommended probability of error.

Appendix B

AX.25

B.1 Introduction

This section will describe the AX.25 version 2.2 protocol. The protocol was developed in order to fulfill the amateur radio community's need for a protocol that would reliably accept and deliver data over a variety of communication links between two signaling terminals.

- The protocol works both in connection-oriented and connection-less environments.
- It supports both half- and full duplex operation.
- It permits the establishment of more than one link layer connection per device.
- It permits self-connection - meaning a device can establish a link to itself using its own address for both the source and destination frame.
- It supports flow control - meaning it is able to detect if the receiving end has a buffer overflow.
- It supports retransmission of corrupt frames.
- It supports n·8 bit data length per frame (default is 256·8 bit).
- It supports unlimited data sizes from upper layers, since it is responsible for segmentation itself.

B.1.1 AX.25 Model

The AX.25 protocol implements the first two layers of the OSI model - the physical layer and the data link layer. This layer can be divided into several distinct functionalities as shown in Figure B.1. The figure indicates a Data Link Service Access Point (DLSAP) at the upper boundary of Layer 2. This DLSAP is the point at which the data link layer provides services to the above layer. Several different data links can exist (in B.1 two data links are illustrated), each data link have their own DLSAP.

Several data link layer entities exist - these are the Link Multiplexer, Data Link, Management Data Link, Segmenter and Physical. Entities in the same layer, but in different systems that must exchange information to achieve a common objective, are called “peer entities”. Entities in adjacent layers interact through their common boundary.

A model of the AX.25 protocol is shown in Figure B.1. As shown it is possible to create more than one DLSAP, e.g. different applications using different DLSAP’s. The different blocks in the model communicate both horizontally and vertically.

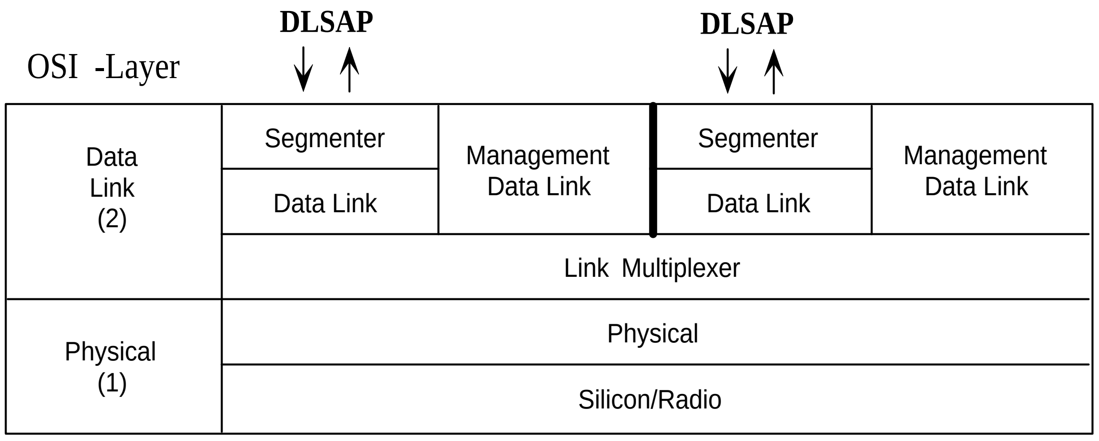


Figure B.1: AX.25 model

Segmenter

The Segmenter entity accepts input from the above through the DLSAP. If the unit of data to be sent exceeds the limits of a AX.25 frame, the segementer breaks the data unit into smaller segments for transmission. Incoming segments are reassembled for delivery to the higher layer through the DLSAP.

Data Link

The Data Link entity is the core of the AX.25 protocol. It provides all functionality necessary to establish and release connections between stations and to exchange information.

Management Data Link

The Management Data Link entity provides all logic necessary to negotiate operating parameters between two stations - e.g. timer settings, window size, duplex mode etc.

These parameters are set during a negotiation phase, which can occur at any time.

Link Multiplexer

The Link Multiplexer allows one or more data links to share a single physical (radio) channel. The Link Multiplexer gives each data link an opportunity to use the channel, according to the algorithm embedded within the link multiplexer.

Physical

The Physical entity manipulates the radio transmitter and receiver. Because different types of radio channel operations are used, the Physical entity exists in different forms. Each form hides the individual characteristics of each radio channel from the higher layer.

B.1.2 Service Primitives

Layer 3 requests services from the data link layer via command/response interactions known as service “primitives”. Similarly, the interactions between the data link layer and the physical layer occurs via these primitives. The primitives exchanged between two layers are of the following four types:

- **REQUEST:** Used by a higher layer to request a service from the next lower layer.
- **INDICATION:** Used by a layer to provide a service to notify the next higher layer of any specific activity that is service related. The INDICATION primitive may be the result of an activity of the lower layer related to the primitive type REQUEST at the peer entity.
- **RESPONSE:** Used by a layer to acknowledge receipt from a lower layer of the primitive type INDICATION. The AX.25 protocol does not use the RESPONSE primitive.
- **CONFIRM:** Used by a layer to provide the requested service to confirm that the activity has been completed.

The service primitives used to communicate between the different entities in AX.25 are shown in Figure B.2.

Example of the usage of these service primitives to achieve some higher functions can be seen in Figure B.3 and Figure B.4.

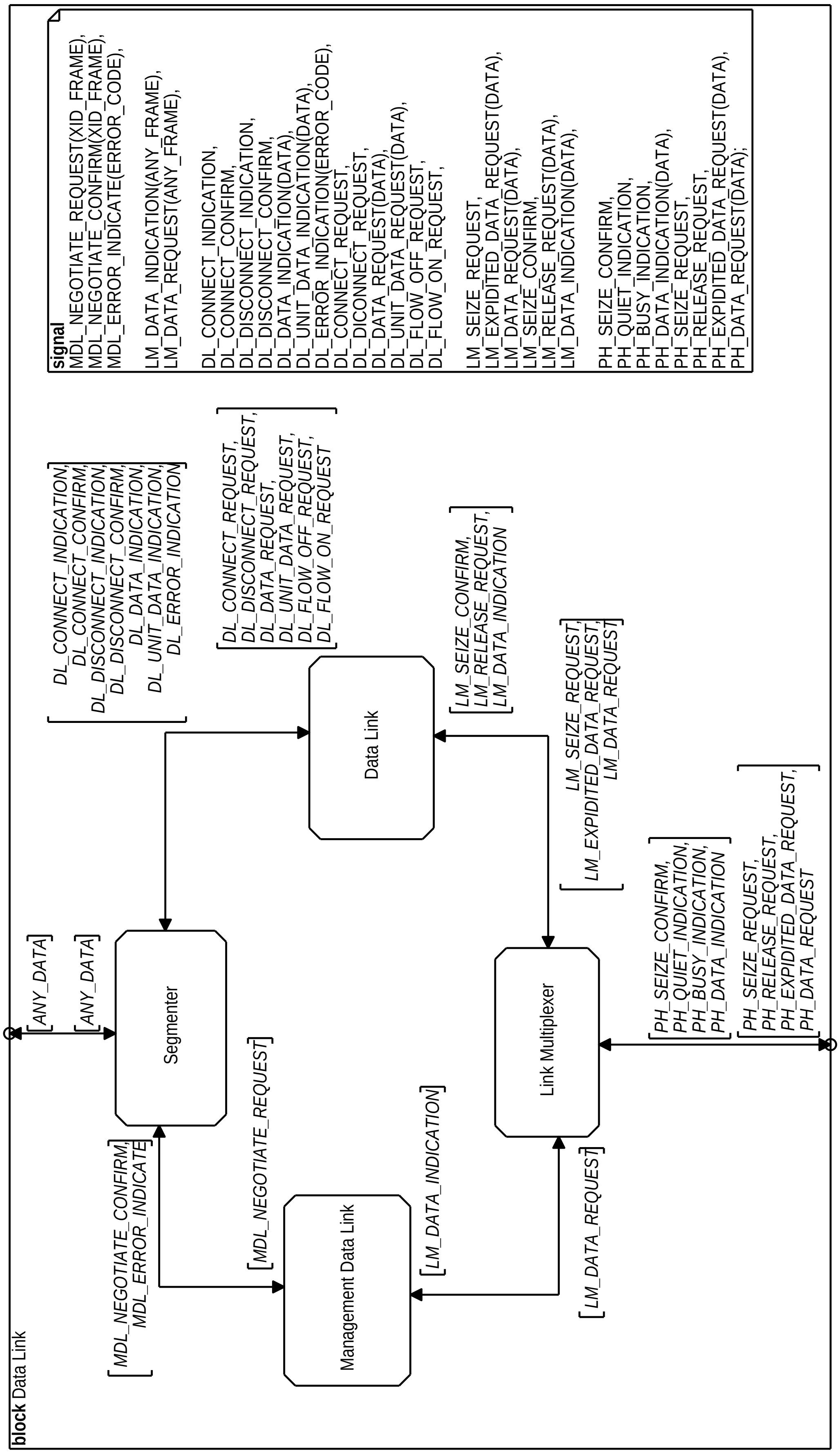

block Data Link
signal
MDL_NEGOTIATE_REQUEST(XID_FRAME),
MDL_NEGOTIATE_CONFIRM(XID_FRAME),
MDL_ERROR_INDICATE(ERROR_CODE),
LM_DATA_INDICATION(ANY_FRAME),
LM_DATA_REQUEST(ANY_FRAME),
DL_CONNECT_INDICATION,
DL_CONNECT_CONFIRM,
DL_DISCONNECT_INDICATION,
DL_DISCONNECT_CONFIRM,
DL_DATA_INDICATION(DATA),
DL_UNIT_DATA_INDICATION(DATA),
DL_ERROR_INDICATION(ERROR_CODE),
DL_CONNECT_REQUEST,
DL_DICONNECT_REQUEST,
DL_DATA_REQUEST(DATA),
DL_UNIT_DATA_REQUEST(DATA),
DL_FLOW_OFF_REQUEST,
DL_FLOW_ON_REQUEST,
LM_SEIZE_REQUEST,
LM_EXPIDITED_DATA_REQUEST(DATA),
LM_DATA_REQUEST(DATA),
LM_SEIZE_CONFIRM,
LM_RELEASE_REQUEST(DATA),
LM_DATA_INDICATION(DATA),
PH_SEIZE_CONFIRM,
PH_QUIET_INDICATION,
PH_BUSY_INDICATION,
PH_DATA_INDICATION(DATA),
PH_SEIZE_REQUEST,
PH_RELEASE_REQUEST,
PH_EXPIDITED_DATA_REQUEST(DATA),
PH_DATA_REQUEST(DATA);
[ANY_DATA]
[ANY_DATA]
Segmenter
[DL_CONNECT_INDICATION, DL_CONNECT_CONFIRM, DL_DISCONNECT_INDICATION, DL_DISCONNECT_CONFIRM, DL_DATA_INDICATION, DL_UNIT_DATA_INDICATION, DL_ERROR_INDICATION]
[DL_CONNECT_REQUEST, DL_DISCONNECT_REQUEST, DL_DATA_REQUEST, DL_UNIT_DATA_REQUEST, DL_FLOW_OFF_REQUEST, DL_FLOW_ON_REQUEST]
Data Link
[LM_SEIZE_CONFIRM, LM_RELEASE_REQUEST, LM_DATA_INDICATION]
[LM_SEIZE_REQUEST, LM_EXPIDITED_DATA_REQUEST, LM_DATA_REQUEST]
Link Multiplexer
[PH_SEIZE_CONFIRM, PH_QUIET_INDICATION, PH_BUSY_INDICATION, PH_DATA_INDICATION]
[PH_SEIZE_REQUEST, PH_RELEASE_REQUEST, PH_EXPIDITED_DATA_REQUEST, PH_DATA_REQUEST]
[MDL_NEGOTIATE_CONFIRM, MDL_ERROR_INDICATE]
[MDL_NEGOTIATE_REQUEST]
Management Data Link
[LM_DATA_INDICATION]
[LM_DATA_REQUEST]

Figure B.2: AX.25 SDL model

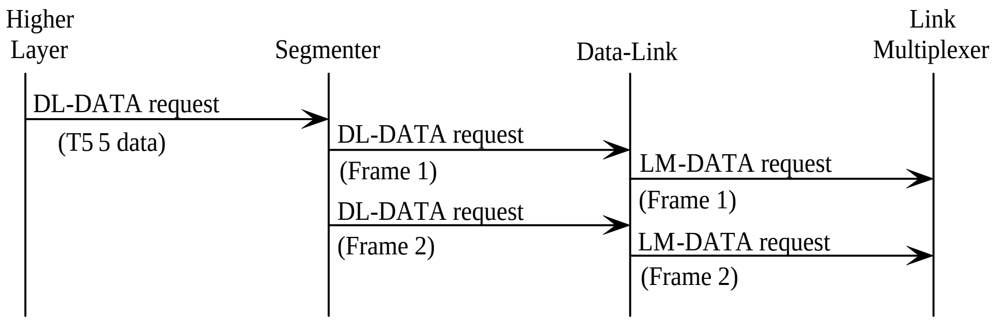


Figure B.3: Example of service primitives involved when sending data.

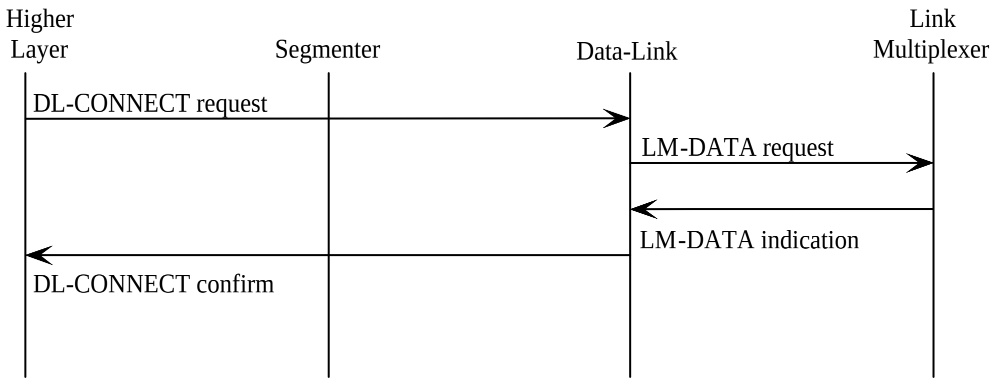


Figure B.4: Example of service primitives involved when creating a connection.

B.2 Frame Structure

Data link transmissions are sent in small blocks of data, called frames. In the AX.25 protocol there are three general types of frames:

- Information frame (I frame)
- Supervisory frame (S frame)
- Unnumbered frame (U frame)

A frame consists of several smaller groups, called fields. Figures B.5 and B.6 shows the construction of the three basic frame types.

Flag	Address	Control	Info	FCS	Flag
01111110	112/224 Bits	8/16 Bits	n*8 Bits	16 Bits	01111110

Figure B.5: U and S frame construction

Flag	Address	Control	PID	Info	FCS	Flag
01111110	112/224 Bits	8/16 Bits	8 Bits	n*8 Bits	16 Bits	01111110

Figure B.6: I frame construction

B.2.1 Flag Field

This field is one byte long, and indicates the beginning and the end of each frame. It consists of a zero trailed by six ones and a zero, ex. 01111110 or in hex numbers (7E). To avoid a flag sequence anywhere else in a complete frame, bit stuffing is used. When broadcasting a sequence of frames, two frames can share the same flag field making a flag indicating the end of one frame and the beginning of another.

B.2.2 Address Field

The address field is 14 or 28 bytes long. It identifies both the source and destination of each frame. It also contains the command/response information and facilities for data link repeater operation.

B.2.3 Control Field

This field is one or two bytes long and is used for identifying the type of frame being sent. Apart from that it contains several control attributes of the data link connection.

B.2.4 PID Field

This field is one byte long. The protocol identifier field identifies which kind of protocol is being used in the third layer, if any. The PID field occurs only in information frames (I and UI).

B.2.5 Information Field

This field contains the actual user data. The default length is 256 bytes and contains an integral number of bytes. These conditions apply prior to bit stuffing.

I fields are allowed in the following types of frames: I, UI, XID, TEST and FRMR.

B.2.6 Bit Stuffing

To ensure that a flag field does not accidentally occur anywhere else in a frame, bit stuffing is used. Every time there is a row of five ones, a zero is added at the end of the row. The zero is then discharged by the receiver.

B.2.7 Frame-Check Sequence

The frame-check sequence (FCS) field is a two byte number used to ensure that the frame was not corrupted during transmission. This number is calculated in accordance with recommendation to the ISO 3309. This field, in opposition to all other fields, is sent with the most significant bit first.

B.2.8 Invalid Frames

A frame is considered invalid if:

1. It consists of less than 17 bytes (Including flags).
2. It is not bounded by flags.
3. It is not byte aligned (an integral number of bytes).
4. It contains at least fifteen trailing ones without bit stuffing zeros (the frame is aborted).

B.2.9 Inter-frame Time Fill

When a TNC needs to keep its transmitter on while not actually sending frames, it should continuously send flags.

B.2.10 Address Field Encoding

As explained above the address field of every frame contain a destination, source and possibly two data link repeater subfields. Both subfields contains a call sign and a secondary station identifier (SSID). The call sign is made up of uppercase alpha and numeric ASCII characters. The SSID is a half byte integer uniquely identifying multiple stations using the same call sign. The HDLC address field is extended with an extra bit, extension bit. The call sign is shifted one bit left to make room for this extension.

Non-repeater Address-Field Encoding

When no data link repeaters are used, the address field is encoded as shown in Table B.1. The address field is separated in two subfields. The destination subfield is seven

Destination Address Subfield	Source Address Subfield
A1 A2 A3 A4 A5 A6 A7	A8 A9 A10 A11 A12 A13 A14

Table B.1: Non-repeater address-field encoding

bytes long, and consists of the call sign and SSID of the designated target. Like the destination subfield, the source subfield is also seven bytes long, and contains the call sign and SSID of the originator. This sequence of the address fields allows receivers to check whether the frame is addressed to them. Both these subfields are encoded in the same way, except that the source address field has the HDLC extension bit set.
For a more complete description of Address Field encoding see [5].

Appendix C

ObjectGEODE

ObjectGEODE is a tool which is used to design and simulate a model described in SDL. The simulation tool in ObjectGEODE can simulate in either exhaustive mode or non exhaustive mode.
Non exhaustive mode can be used to test the model with specific traces and random inputs and acts as preliminary debugging, whereas exhaustive mode can be used to check every possible trace in the model and will therefore act as final debugging. Before an exhaustive simulation can be conducted the model should follow these rules:

- A process queue must be restricted from growing towards infinity.
- No variables must have a infinite number of values

If one of these rules are broken the exhaustive search will never finish because new states will always appear. To fullfill the first rule, filter conditions are used. This restricts the simulator in ObjectGEODE from taking any transitions which makes a filter true. Another possibility is to create stop conditions which interrupts a simulation if true. An example filter is:

```
filter length (transportsat!ttc_process(1)!queue) > 2
```

The 'filter' tells the simulator that this is an filter condition. 'length' get the length of the transportsat's ttc_process' input queue. The '> 2' is the boolean condition. Meaning that all transitions adding to the input queue of the ttc_process are ignored.

C.1 Startup File

The startup file is used when the simulation facility is initialized. It is possible to state filters and stop conditions as well as feeds to the system. The startup file can be used is instead of entering all feeds, filter and stop conditions at each simulation start. The start up can be seen in (/cd/SDL/transport.startup)

C.2 State Graph

When using ObjectGEODE to simulate it translates the model into state graphs. The different states are connected with transitions. What characterizes a state is its input queue and its internal variables. For example in Figure C.1 a process is started and no transitions taken. This is depth 1 and the breadth is 1. From here there are three possible transitions each ending in three new states, here the depth is 2 and breadth is 3. When an exhaustive simulation is to be conducted all states has to be visited. A few different approaches to exploring the state graph is possible.

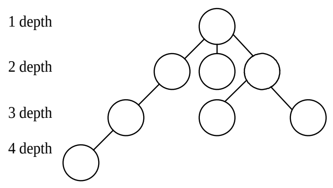


Figure C.1: A state graph example.

- Breadth first - The simulator explores all possible transitions in one depth before continuing to the next depth.
- Depth first - The simulator chooses a transition and continuing from this state until an edge is reached.

If an exhaustive simulation cannot be completed the result is not very useful because some parts of the state graph is not explored. For example if the depth first exploration continues to go deeper into a graph because of an variable that keeps incrementing then only a small percentage of the state graph might be explored.

C.3 Simulation

The three different types of simulation will be described this beeing simulation, verification and validation.

C.3.1 Manual Simulation and Random Simulation

Testing by manual simulating is, as mentioned before, non exhaustive. This means that the simulation will normally not cover all the different transitions in a model. It can be used to examine certain scenarios, for example parts of a test specification. Another use is to let ObjectGEODE choose which transitions to be taken and generate random inputs to the model checking how the model responds. This is very useful for detecting possible errors since the simulator may take transitions differently than the designer would. Also this is much faster than manually selecting the transitions.

C.3.2 Verification

Verification is used to detect scenarios which leads to exceptions, deadlocks and loss of signals. The trace coverage of the model is total meaning that if the verification is complete it is guaranteed that none of the errors mentioned before will occur.

C.3.3 Validation

Validation is used to check that the model responds to stimuli as stated in the requirement specification. When informing ObjectGEODE two formalisms for expressing the requirement specification are used:

- Stop conditions - Expresses a condition which halts the system if true.
- MSC - Expresses a part of the model as a signal sequence. MSC can be put together to form the complete system.

The MSC tree will be described in further detail in the following

C.3.4 MSC Tree

When describing the behavior of a process' specific interface, MSC's can be used to specify a signal flow at a given time. E.g., if `connect` always causes a `sig_receiveEvent` the connect behavior is described, this can be seen in Figure C.2 the connect behavior is described. If every functionality for a process is described in MSC's all signals to and from the process should be found in a MSC. To describe an entire system the MSC's has to be interconnected with each other. ObjectGEODE uses a hierarchially order for building a system. Figure C.3 shows that first an init behavior is expected and thereafter the connect behavior. The different types of compositions which connect to MSC are summarized:

- And - All leafs are visited one at a time.
- Or - One of the leaf are chosen.
- Repeat - Repeat its leafs 0-∞ times.

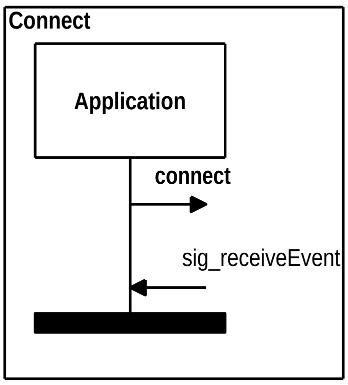


Figure C.2: MSC for the signal sequence when a connect is received

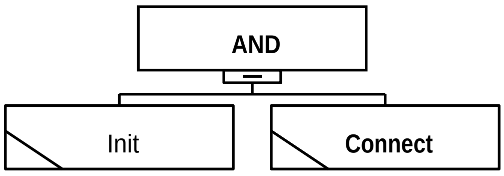


Figure C.3: MSC tree showing an AND composition between an init and connect

C.3.5 Validating

When a MSC tree model has been built two options exist for checking this with the SDL model.

- Verify - All possible traces are examined and if a signal pattern arrives which is not specified in a MSC it is considered to be an error.
- Search - Only checks whether the MSC tree is a possible behaviors for an interface.

The difference in the MSC tree for the two methods of testing is that for verify a complete model must be made whereas search is successful when the behavior is found once. Verify is more reliable since the complete behavior is described.

Appendix D

OSI Reference Model

The International Standards Organization (ISO) has developed a reference model for Open Systems Interconnection called the OSI-model. The OSI-model is a well known model for structuring interconnection of different types of communication system with the use of layering. The communication among application processes is partitioned into an ordered set of layers which is illustrated in Figure D.1. For further information about the OSI-model please refer to [4, page 28].

D.1 The Physical Layer

The physical layer is only concerned with the transmission of raw bits over a communication channel. Meaning this layer should make sure that when one side sends a bit, this should be received correctly by the other side. This typically involves specification of how a bit shall be represented, if signals can be send in both directions simultaneously, how an initial connection is established, how a connection is stopped etc.

D.2 The Data-link Layer

The data-link layer is concerned with using the facilities of the physical layer and transform these into a line that appears free of transmission errors to the network layer. This is done by having the sender break the data into data frames (usually from 0-1000 bytes) and then sequentially transmit these frames. The receiver then sends acknowledgment frames telling whether the frame was received successfully. Noise can garble the bits in a frame, but they normally contain a checksum so detection of garbled frames is possible. If necessary the data can be retrieved in several ways. One option is to implemented some form of forward error correction which can reconstruct destroyed data. Another option is to use the checksum to detect garbled frames, and then retransmit them. Another issue the data-link layer needs to deal with is how to keep a fast transmitter from drowning a slow receiver with data. Some regulation is needed in order to let the trans-

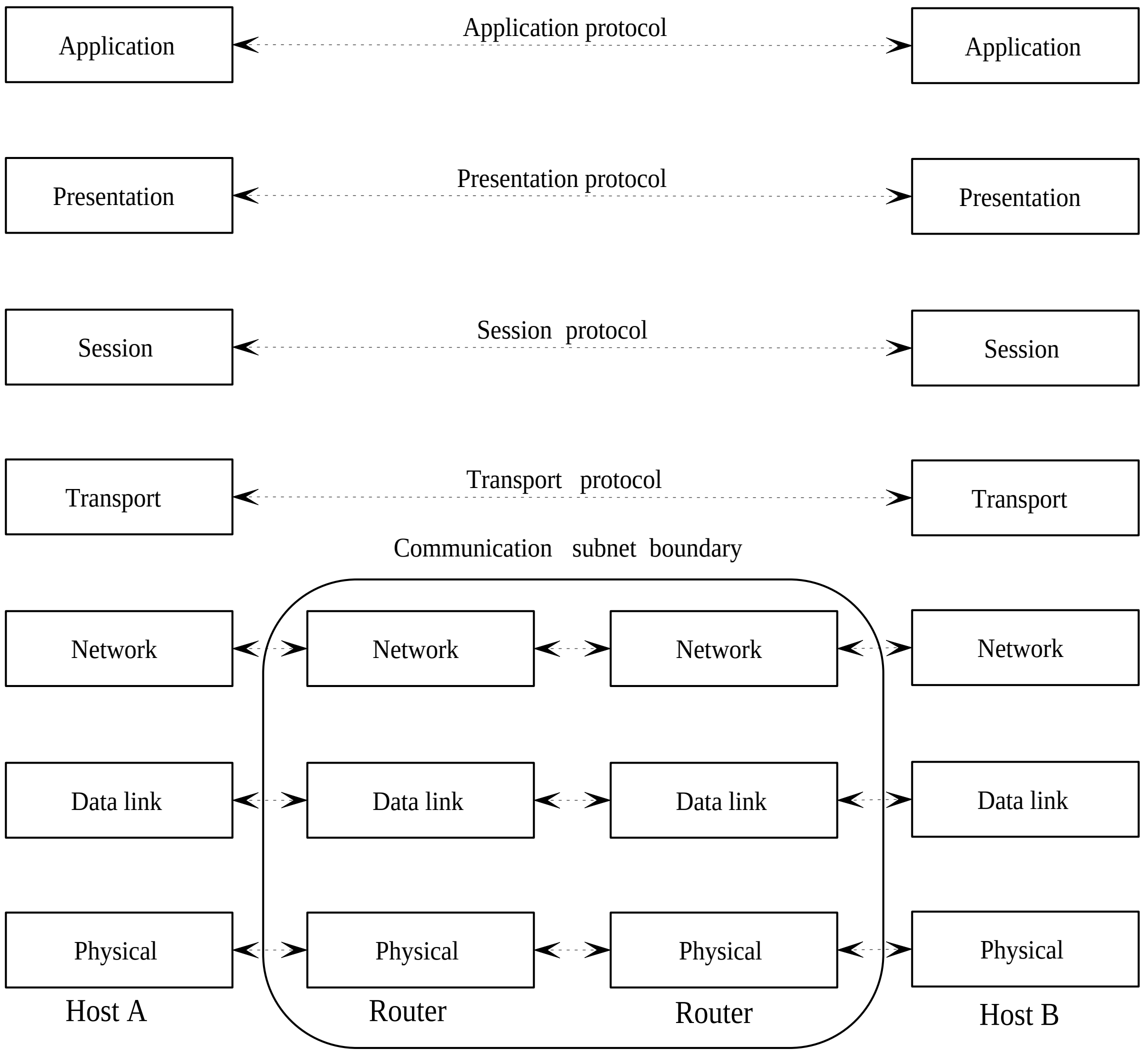


Figure D.1: Seven layer OSI reference model

mitter know how much buffer space the receiving machine has in the moment. Finally in broadcast networks there is an extra issue for the data-link layer to deal with: how to access a shared channel. A sub-layer, the medium access sub-layer, deals with this problem.

D.3 The Network Layer

The network layer controls how packets are routed from source to destination through a network. The route a packet follows can be based on static tables, determined at the start of each conversation or determined on a per packet basis. If too many packets are sent they can get in each other's way, forming bottlenecks. Such congestion must also be handled by the network layer. When a packet is send from one network to another many problems can arise. The addressing used by the networks may differ, the packet size may differ and so on. It is up to the network layer to handle all these problems so different networks can be interconnected.

D.4 The Transport Layer

The basic function of the transport layer is to accept data from the session layer, split it into smaller units, pass these to network layer and ensure that they arrive correctly at the other end. This must be done in an efficient way and must isolate the upper layers from the network. Under normal conditions the transport layer creates a distinct network connection for each connection required by the session layer. If high data throughput is required several network connections can be made, dividing the data among them to improve throughput. On the other hand the transport layer can also multiplex several transport connections onto one network connections, this is especially useful in creating and maintaining network connections, which requires many resources. The transport layer can offer different types of services, the most popular being an error-free point-to-point channel that delivers data in the order they were sent. Other services can be offered, for example broadcasting or transport of isolated messages with no guarantee about the order they arrive in. Since many connections can be made to one host, some way of telling which connection data belongs to is needed. The transport layer header is one place this information can be placed. Creating and deleting the network connections along with flow control is also the responsibility of the transport layer.

D.5 The Session Layer

The session layer allows users on different machines to establish sessions between them. A session allows ordinary data transport, like the transport layer, but can provide enhanced services useful in some applications e.g. logging into a remote timesharing system or file transfer between two machines. Another service in the session layer is synchronization. Consider the problem with a two hour file transfer between two machines with a mean time of one hour between crashes. After a crash the whole transfer would have to be repeated. To eliminate this problem the session layer provides the means to insert checkpoints into the data stream, so only data after the last checkpoint will have to be retransmitted in the event of a crash.

D.6 The Presentation Layer

Unlike the lower layers, which moves bits reliably from once place to another, the presentation layer is concerned with the syntax and semantics of the data being transmitted. A typical example of a presentation service is encoding data in a standard way. Different computers have different ways of representing e.g. integers, floating- point numbers, string etc. To make it possible for two machines with different representations to communicate, the data structures to be exchanged can be defined in a abstract way, along with a standard encoding to be used for the actual transfer. The presentation layer manages these abstract data structures and converts them to the appropriate representation.

D.7 The Application Layer

The application layer contains a number of commonly needed protocols. Consider for example a full screen editor that needs to work over a network with many different incompatible terminals. One solution to the problem is to define a virtual terminal that programs are written to deal with. To handle each actual terminal type a piece of software maps the functions of the virtual terminal onto the real terminal. Another example is transferring files between systems with incompatible file systems, handling this problem also belongs to the application layer. Other work for the application layer can be e.g. e-mail, remote job entry, directory lookup and a variety of other facilities.

Bibliography

[1] Group 554. *DL55X - The AAU CubeSat Data Link Protocol.* AAU, 2001.

[2] Stephen Biering-Sørensen. *Håndbog i Struktureret Program Udvikling.* Ingeniøren-Bøger, first edition, 1988. ISBN: 87-571-1046-8.

[3] Wiley J. Larzon and James R Wertz. *Space Mission Analysis and Design.* W.J. Larzon and Microcosm Inc., second edition, 1992. ISBN: 0-7923-1998-2.

[4] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall Inc., third edition, 1996. ISBN: 0-13-394248-1.

[5] Douglas E. Nielsen William A. Beech and Jack Taylor. *AX.25 Link Access Protocol for Amateur Packet Radio.* Tucson Amateur Packet Radio Corporation, 1997.