A RELIABLE TELEMETRY SOFTWARE DESIGN FOR A SATELLITE SYSTEM

Babudi Turcia Boshielo

THESIS PRESENTED IN FULFILLMENT OF THE REQUIREMENTS FOR THE

MASTER OF SCIENCE IN ENGINEERING SCIENCE

AT THE UNIVERSITY OF STELLENBOSCH

SUPERVISOR: MR J. TREURNIGHT

December, 2001

DECLARATION

I the undersigned, declare that the work contained in this thesis is my original work and has never been submitted, in part or in its entirety, at any University, for any requirement towards the achievement of any degree.

B.T Boshielo

ABSTRACT

One of the requirements for satellite telemetry systems is the provision of reliable telemetry data to allow accurate monitoring of the satellite status. This thesis focuses on the design of telemetry software that meets this data reliability requirement. An improved synchronization strategy to allow efficient ground reception of the telemetry data is implemented on SUNSAT's direct link. The data collection and transmission functions are also enhanced by the addition of the necessary redundant information to the data while meeting the real-time requirements of the system. To enhance the software quality a development methodology entailing structured programming practices and modular decomposition is proposed. It is shown that the resulting telemetry software fulfils the functional requirements.

OPSOMMING

Een van die vereistes vir die satelliet telemetrie stelsel is die verskaffing van betroubare telemetrie date om akkurate monitering van die satelliet status te verseker. Hierdie tesis fokus op die ontwerp van die telemetrie sagteware wat hierdie data betroubaarheid bevredig. 'n Verbeterde sinkronisasie strategie is geïmplementeer om meer effektiewe grond ontvangs te verseker van die telemetrie data op SUNSAT se direkte skakel. Die data versameling en transmissie funksies is ook verder verbeter deur die aanvulling van nodige oortollige informasie in die data terwyl die intydse vereistes van die stelsel steeds bevredig word. Om die sagteware kwaliteit te verbeter is 'n ontwikkelings metodiek voorgestel wat gestruktureerde programeering strukture en modulêre oplossings tot gevolg het. Die voltooide telemetrie sagteware het getoon dat dit al die vereistes bevredig.

ACKNOWLEDGEMENTS

I would like to thank the following people:

- Mr J. Treurnight, my supervisor, for his words of encouragement and repeatedly pointing me in the right direction.
- Mr Xandri Farr, my mentor, for his help and continued support throughout this work.
- My family, for their interest and always being there for me in all respects.
- All the people who have contributed to this effort in one way or the other.

TABLE OF CONTENS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

A/D	analog to digital
ADC	analog to digital converter
ADCS	attitude determination and control subsystem
BER	bit error rate
CCSDS	Consultative Committee of Standardization for Data Subsystems
CPU	central processing unit
CRC	cyclic redundancy check
EDF	earliest deadline first
ETX	end of text
IEEE	Institute for Electrical and Electronic Engineers
ISO	Internal Standards Organization
ISR	interrupt service routine
OBC	on board computer
OSI	open system interconnection
PC	personal computer
RAM	random access memory
RMS	rate-monotonic scheduling
RSW	received sync word
RTOS	real-time operating system
SSB	SUNSAT serial bus
SSW	stored sync word
SRS	software requirement specification
STX	start of text
SUNSAT	STELLENBOSCH University Satellite
TLM	telemetry
TMS	telemetry and modem subsystem
WOD	whole orbit data

SCOPE OF WORK

The work started with an analysis of the SUNSAT *(Stellenbosch University Satellite)* telemetry system and its operation with the intention of finding areas that require improvement and subsequently implementing the new developments on the telemetry software.

SUNSAT telemetry data can reach the ground station in four different ways, of which two are hardware paths, which are not discussed in this thesis because the main interest lies on the software implementation. The software telemetry paths include data collected by a telemetry micro-controller and send via two onboard computers, and data collected by the micro-controller and send directly to the ground-station.

The thesis looks at the performance of these two software telemetry paths, with **data reliability** as one of the key elements for analysis. The results of the analysis indicated that the OBC data path is more reliable due to the AX.25 protocol implemented on the channel. The direct link on the other hand did not prove to be reliable and a decision was taken to improve the communication on this link. The approach used for this task was to segment the work into three focus areas, namely, the ground-station synchronization, the communication protocol for the link and the telemetry micro-controller scheduler.

Each of the three sections mentioned, are tackled individually to find an improved implementation of each. Firstly an outline of the present functionality is given. Followed by the new design that is backed by theoretical work. The ground-station synchronization strategy was developed and tested. Finally the software requirements for the improved software covering all the three sections are laid out followed by the software design and implementation. The telemetry hardware for SUNSAT is used as a test platform for the final software produced.

CHAPTER 1

1 INTRODUCTION

The work contained in this thesis is introduced by providing a general description of the telemetry concept and then relating this description to the SUNSAT telemetry system. After formally stating the aim of the thesis, the document layout is outlined.

1.1 Concept Definition

The IEEE Standard 100 offers the following definition of ***telemetry***:

> *Telemetering (remote measurement) is measurement with the aid of intermediate means that permit the measurement to be interpreted at a distance from the primary detector. The distinctive feature of telemetering is the nature of the translating means, which includes provision for converting the measurand into a representative quantity of another kind that can be transmitted conveniently for measurement at a distance. The actual distance is irrelevant.*

Telemetry data is therefore data that is obtained through the above stated means. A telemetry system, also referred to as a data handling system gathers, processes and formats spacecraft housekeeping and mission data for down-link or use by an onboard computer [JW&WL, 1999, p395]. Data handling combines telemetry data from multiple sources on board the spacecraft.

A typical satellite telemetry system is illustrated in Figure 1.1. The system is divided into two segments: the space segment that performs the data acquisition, processing and transmission functions, as well as the ground segment that receives, decodes, displays and stores the telemetry data. Telemetry sensors measure data in the form of voltages, currents, charges, etc. The raw non-digital sensor outputs are converted into voltages in a standard range in a process called **signal conditioning** [JAK, 1999, p14]. Analog-to-digital converters are used to change the conditioned analog signals to their digital equivalent before processing. The data processing function involves data formatting, where information is added to the telemetry data to ensure data reliability, as well as data storage as whole orbit data (WOD), when the satellite is out of view of the ground station, for later retrieval.

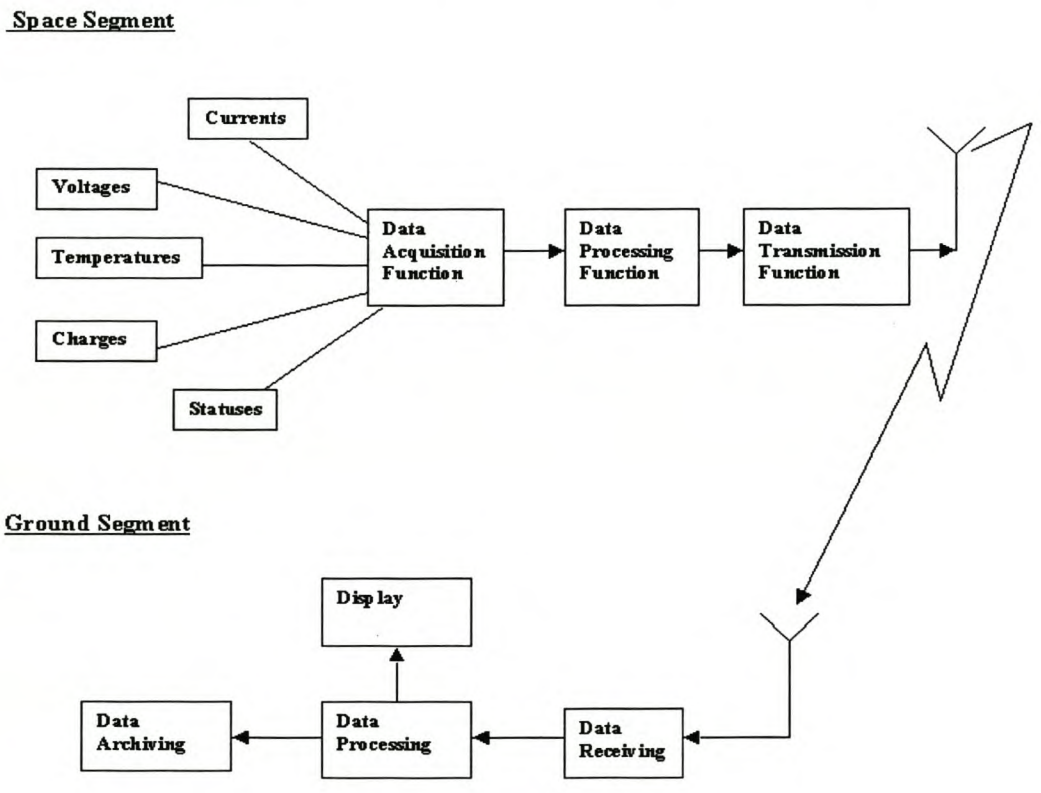

Figure 1.1: *A typical satellite telemetry system*

In the receiving function, the ground-station synchronizes on the data using synchronization information in the data if any was added. Decoding refers to the stripping off of the additional data so that only the useful information remains, which is then displayed and/or stored for ground station use.

1.2 SUNSAT Telemetry System

Before formally stating the aim of this thesis and describing the layout of the document it is necessary to have an understanding of the structure of the SUNSAT's telemetry system based on the broad telemetry concepts of section 1.1.

The structure of the telemetry system under study is shown in Figure 1.2. The 80C31 micro-controller is the heart of software telemetry. The system is a centralized system, with a number of subsystems to monitor. The **data acquisition units** combines the telemetry information from eight different satellite trays and presents it to the data collection function in digital format. The micro-controller collects data from the data acquisition units, then packages the data into telemetry frames, and makes it available to other subsystems as well as transmitting the data to the ground station.

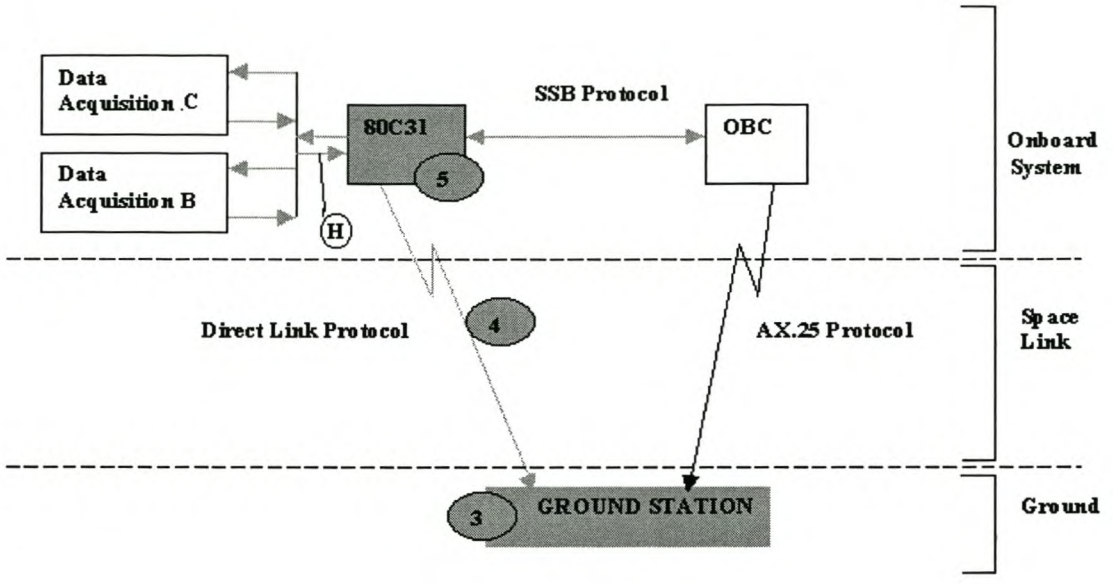

Figure 1.2: *Structure of Sunsat's telemetry system*

As illustrated in Figure 1.2, the 80C31 micro-controller can send data to an onboard computer (OBC) where it is then transmitted using the **AX.25 protocol**, or the data can be send directly to the ground using an asynchronous protocol, the **direct link protocol**. An alternative to accessing telemetry data is indicated as the circled letter H, which shows data that is sent to ground through hardware, bypassing the onboard processors. A description of the operation of the system is given in the Chapter 2.

1.3 Aim of the Thesis

The aim of the thesis is to design and implement telemetry software for the SUNSAT telemetry system. The design starts with the system analysis to determine areas of the software that can be implemented differently to benefit the entire telemetry process.

The shaded parts in Figure 1.2 indicate the areas on which improvements were made. Each circled number corresponds to the chapter describing the work on that part. The document layout follows in section 1.4.

1.4 Thesis Overview

The rest of thesis is structured as follows:

- **Chapter2: System Analysis**
  A detailed structure of the system, its functions, and interfaces are outlined, to provide an understanding of the telemetry system and the environment around it. The chapter further gives an account on the performance analysis of the telemetry system. The decisions on the improvements to be carried out are made. The decisions are supported by the need for reliable telemetry data.

- **Chapter 3: Ground-station synchronization**
  This chapter looks at frame synchronization strategies and suitable unique words for synchronization. A frame synchronization strategy is proposed and a unique word is chosen, to improve the ground station synchronization.

- **Chapter 4: Direct link protocol**
  The communication protocol to ensure reliable transfer of telemetry data from the spacecraft to the ground-station is proposed. This involves the design of a telemetry data format with the necessary redundant information added to the collected data to facilitate reliable transmission and successful management of the data at the ground.

- **Chapter 5: Task Scheduling for the 80C31 micro-controller**
  A task scheduler for the telemetry tasks is proposed. Depending on the nature of the tasks to be scheduled, the scheduler is designed to meet the system requirements.

- **Chapter 6: Telemetry Software**
  The propositions of chapter 4 and 5 are implemented in software that is targeted for the telemetry micro-controller. This entails a complete software process consisting of a requirement specification, software design, and software implementation and testing.

- **Chapter 7: Conclusion**
  This chapter concludes the work by summarizing the modifications required on the telemetry system and indicating what benefits are gained. Areas for further study are recommended.

1.5 References:

[JW&WL, 1999] James R. Wertz, "Space Mission Analysis And Design", third edition, Microsim Press, 1999.

[JAK, 1999] Jan-Albert Koekemoer, "Investigation of a Command and Data Handling Architecture for SUNSAT-2 Micro Satellite", Masters thesis, University of Stellenbosch, November 1999.

CHAPTER 2

2 SYSTEM ANALYSIS

In order to develop software for a hardware system, it is imperative to understand the structure and operation of that system. This chapter gives a brief overview of the SUNSAT telemetry system and its operation. Furthermore the weaknesses in the system performance are identified and a conclusion is drawn about the areas that will be improved to benefit the system.

2.1 SUNSAT Onboard Telemetry System

The following summary of SUNSAT onboard telemetry system was adapted from the thesis of I.de Swart [IdS, 1994].

2.1.1 System Overview

Refer to Figure 2.1 for the system structure. The telemetry system consists of two data acquisition modules (B and C) to provide redundancy. Each module is able to scan analog and status information from 8 different subsystems. The acquisition modules uses an 8-bit A/D converter each, to convert the analog data into digital format which can be input by a processor or transmitted through hardware. An 8-bit counter is used to generate the address signals for the acquisition modules. Acquisition module B has the added feature that the addresses can be generated by the telemetry system micro-

controller. A 32-bit hardware generated frame counter is used to count the amount of telemetry frames.

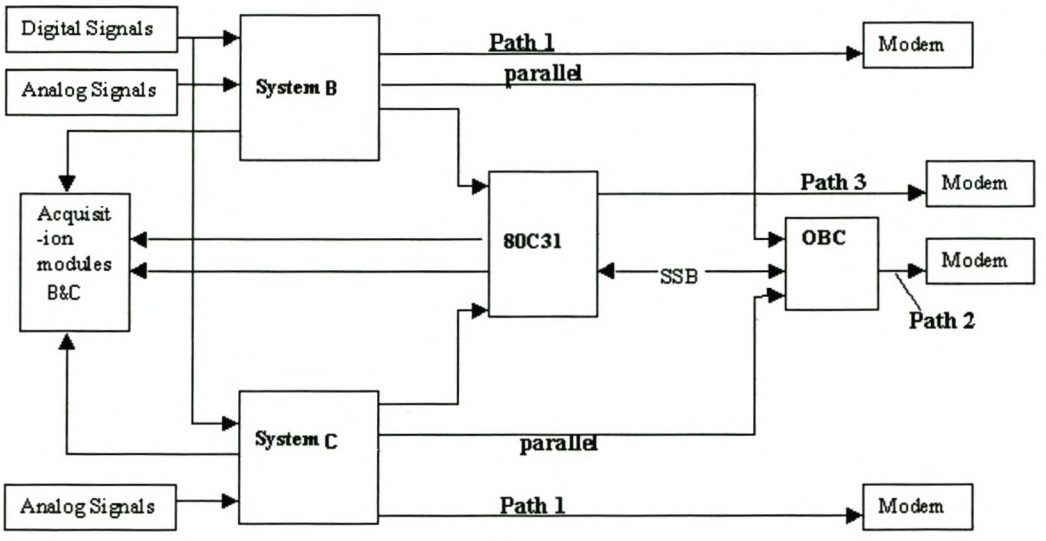

Figure 2.1: *Onboard telemetry system block diagram*

2.1.2 Hardware Interfaces

- The data acquisition modules
  -Sample analog channels and converts the analog data to digital format.
  -Generate addresses to indicate which channels to sample.

- The Sunsat Serial Bus (SSB).
  -It is a serial communication interface between the 80C31 and the OBC's.
  -Backup system for transferring telemetry data to OBC's.

- The OBC's.
  -Relay commands from ground station to the telemetry system and receives telemetry data from the micro-controller system.

- The modems.
  -The modems relay telemetry data from the subsystems to the ground station
  -Data is transmitted but is not received via these modems.

2.1.3 System Operation

Data is collected at a rate of 100 samples per second (1200-baud), as well as at a rate of 800 samples per second (9600-baud). The different channels of all the subsystems are combined into a single frame consisting of 256 bytes. A hardware generated frame counter is collected as the first four bytes in each frame (channels 0, 1, 2 and 3). If the channels are sampled continuously, the delay between each frame is 2.56s when sampled at 1200 baud or 0.32s when sampled at 9600 baud.

Telemetry data is collected and transmitted through the following routes:

- Telemetry data consisting of full frames is directly transmitted from the data collection hardware, either subsystem B or C, through the 1200baud modems. This telemetry path is shown as *path 1* in Figure 2.1. It is a hardware based telemetry mode.

- Any OBC can read telemetry data directly from subsystem B or C, with a parallel interface, and package the data using the AX. 25 protocol for transmission. This path is marked *parallel* in Figure 2.1.

- The last two telemetry routes indicated in Figure 2.1 as *path 3* and the *SSB path* are implemented by the i80C31 microprocessor. The controller collects the data when requested by tele-command from the OBC's and either sends the data to the OBC or sends it directly to the ground station through the 1200-baud modems.

The telemetry micro-controller (i80C31) operates in two distinct modes:

- Minimum telemetry mode

  This is a hardware telemetry mode whereby the data acquisition systems send telemetry data to the ground-station without the help of any of the onboard processors, neither the i80C31 nor an OBC. This data path is *path 1* in Figure 1.2.

- Full telemetry mode

  This is a software telemetry mode with the onboard processors, i.e the i80C31 and the OBC's communicating to carry out the telemetry functions.

  This mode is divided into two sub modes to make a distinction between telemetry data that is collected by the i80C31 and then send to the OBC's for transmission through the AX.25 protocol, and telemetry data that is collected by the i80C31 and transmitted through the direct link protocol.

  The direct link protocol uses the same data and frame format as the one for hardware telemetry. The direct link telemetry frame format is shown in Figure 2.2.

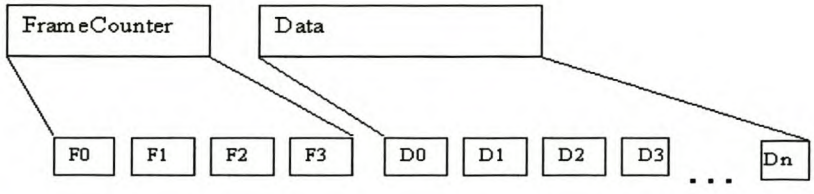

Figure 2.2: *Direct link telemetry frame format*

Different microprocessor functions are executed on requests from the OBC's. The OBC's communicates with the 80C31 using the SSB protocol. Further details about this protocol, the OBC commands and the 80C31 functions are described in appendix E.

2.2 Identification of Weaknesses in the System

Telemetry information helps to monitor the health and status of the satellite, it is therefore important that this information is received correctly and without errors so that the health and status of the satellite are monitored accurately. If errors are present there must be at least a way to indicate the presence of the errors if not correcting them, to prevent the use of erroneous information.

The three telemetry transmission paths shown in Figure 1.2 are summarized below to establish the data reliability of each path.

The hardware path (H) - this transmission path uses the same protocol as the direct link protocol. As a result the data received via this link will also have the same characteristics as the direct link data. But because the thesis is software development, this path does not form part of the work in this thesis and will not be discussed further.

The AX.25 protocol - this is a well-established protocol for data communications. Due to the use of the AX.25 protocol for the relaying of TLM data via the OBC's, this TLM path is regarded as a reliable link and no attempts are made to change the functionality of this communications link.

The direct link protocol - The micro-controller link called *path 3* in Figure 1.2 on the other hand uses no special protocol for its services. This makes the link an interesting focus

area for improvement. But before a decision is taken to improve the link, analysis of the data received on this link is carried out to investigate the link's reliability.

2.2.1 Direct Link Analysis Procedure

Some of the raw telemetry data files containing data received from the direct link were analyzed. A total of ten raw data files were used. A software module was written to

- Search for the frame-counter in each telemetry frame.
- Determine the continuity of the frame-counter to establish data loss.
- Count the number of bytes in each frame to determine shortages and additions.

2.2.2 Results

Table 2.1 below shows the results obtained from the analysis made:

Table 2.1: *Results of telemetry data analysis*

File Number	No. of successive bytes missing						No. of successive frames missing					
	1	2	3	4	5	6	1	2	3	4	5	>5
	No. of occurrences in file						No. of occurrences in file					
File 1	1	1	-	-	-	1	-	1	1	-	-	1(91)
File 2	-	-	-	-	-	-	-	-	1	-	-	1(1401)
File 3	5	-	-	-	-	-	-	-	1	1	-	1(245)
File 4	3	-	-	-	-	-	-	-	-	-	-	-
File 5	-	-	-	-	-	-	-	4	2	1	-	1(64)
File 6	3	-	-	2	-	-	1	2	-	-	-	3(9,80,311)
File 7	-	-	-	-	-	-	-	-	-	-	-	2(11,2)
File 8	1	1	-	-	-	-	-	-	-	-	-	1(211)
File 9	-	-	-	-	-	-	-	-	-	-	-	-
File 10	7	-	-	-	-	-	-	3	-	-	-	2(5,19)

Table 2.1 shows the number of occurrences of a given number of data bytes missing. For example, in File 8 there is one occurrence of a single byte missing, two occurrences of two successive bytes missing and one big gap of a missing frame block of size 211.

The results of the link study are summarized in Figure 2.3 to give a generalization of the communication's link performance.

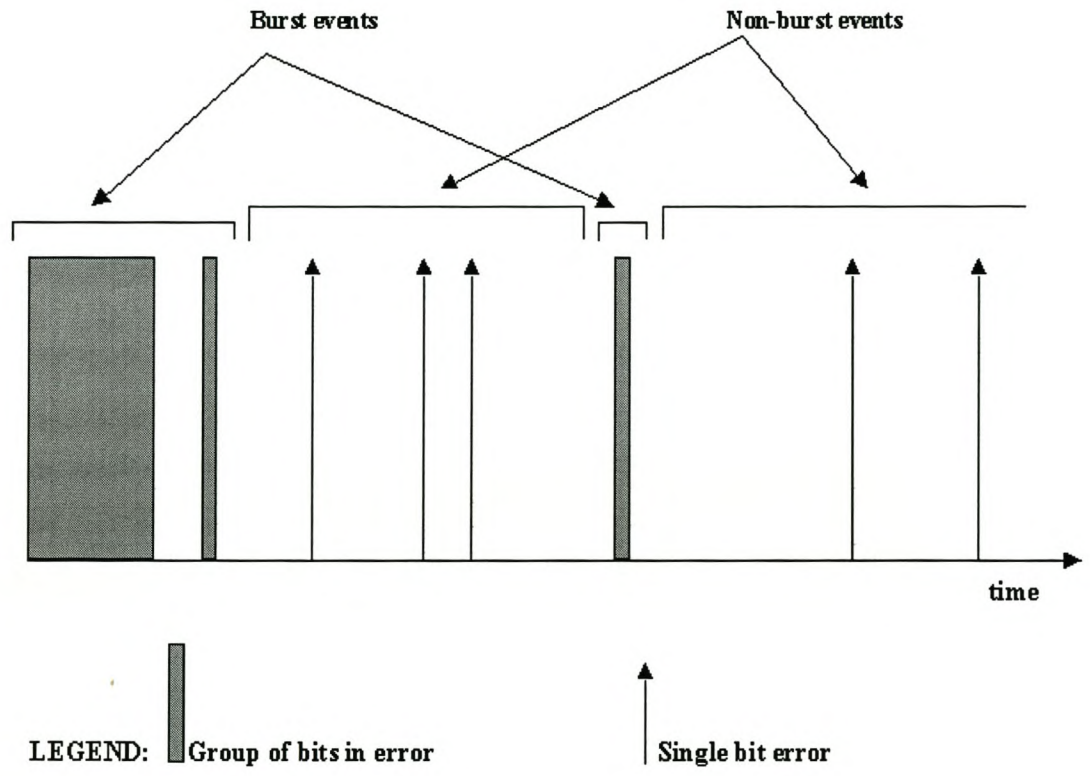

Figure 2.3: *Characteristics of the communications link*

The term *burst event* is defined to mean a time-duration when there are a number of bits in error either back to back or with smaller bursts of errors periodically occurring over a short time. It may be possible to have single bit errors occurring as well within the burst event. A *non-burst event* is defined to mean a time when there are only single-bit errors that occur in a random manner.

Therefore the result is a combination of block errors and single bit errors. Using this distinction between non-burst events and burst events, it is found that the link operates in the non-burst event mode for about 90% of the time. Bursts of errors occur mostly during the initial stages of communication and occasionally in the middle of communications where bit errors are more likely. The possible causes of these errors are unclear and may be a combination of the coding scheme used and the interference on the physical link.

- Out of the ten files analyzed, only two are received without problems. It can therefore be concluded that the communication link is not reliable. To make the link even worse is the fact that no error handling is implemented.
- Loss of blocks of data is common. This is probably the result of the receiving function's inability to synchronize with the incoming data, since the only synchronization information in the data stream is the four-byte frame-counter.

2.3 Conclusion

As a result of the above analysis and results, there is a good reason for the intended software to try to improve on the coding scheme used on the direct link channel. The main objectives of the thesis are thus stated as follows:

- To improve the ground-station synchronization on direct link data.
- To improve on the communication protocol used on the direct link channel.

- To implement the telemetry micro-controller scheduling to meet the new functionality requirements of the system.

The next three chapters look at each of these objectives separately.

2.4 References:

[IdS, 1994] I. De Swart, "The Design of a Reliable and Flexible Telemetry System for SUNSAT", June 1994.

[JAK, 1999] Jan-Albert Koekemoer, "Investigation of a Command and Data Handling Architecture for SUNSAT-2 Micro Satellite", Masters thesis, University of Stellenbosch, November 1999.

CHAPTER 3

3 GROUND STATION SYNCHRONIZATION

One of the requirements for efficient data acquisition at the ground station is proper synchronization of the satellite's telemetry system with the ground station receiver. This allows the receiver to determine the location of the packets in the received data sequence.

In this chapter the synchronization method used on SUNSAT is looked at. The section marked 3 in Figure 1.2 indicates this area of focus. An improved synchronization method is designed and implemented.

3.1 The Synchronization Algorithm Used for SUNSAT

In his paper on optimum frame synchronization, Massey recognizes that the most widely used method for frame synchronization is to insert a fixed binary pattern or **"sync word"** periodically in the data stream [JLM, 1972]. On the assumption that symbol synchronization has already been obtained, the receiver obtains frame synchronization by locating the position of the sync word in the data stream.

The current synchronization strategy implements the sync word concept described above. A frame consists of 256 bytes of which the first four constitutes a frame counter, which is used as a sync word. The counter is incremented by eight for every successive frame, when the sampling speed is 100 samples per second, which is the case.

Synchronization is achieved by searching for the frame counter in the received data sequence to mark the beginning of a frame. This frame format is illustrated in Figure 2.2.

The procedure is as follows [IdS, 1994]:

> The first 256 bytes are received and stored as the test frame. It does not matter where in the frame the reception starts. The following data bytes are received and compared to the previously stored bytes. The first byte received is compared to the first byte in the test frame; the second byte received is compared to the second byte in the test frame, etc. If four consecutive bytes are found of which the first byte is 8 higher than its corresponding byte in the test frame, then there is a good possibility that synchronization has been achieved.

The synchronization strategy did not prove to be a very effective way to synchronize the ground-station receiver with the data stream. The link performance study carried out showed that a lot of telemetry frames are lost during the process of trying to establish initial synchronization.

Ease of initial acquisition of the sync word pattern depends on the nature of the sync word used in the synchronization strategy [HS et al, 1969].

Synchronization words are chosen such that they have good auto-correlation and cross-correlation properties [HS et al, 1969]. The counter for every frame is the increment of the previous counter by 8. Even though the length of the counter seems to be sufficient, it is just an ordinary number with no special uniqueness characteristics. As a result, this contributes to the difficulty in achieving initial acquisition. The fact that it is not fixed makes the first sync acquisition latency worse because the sync algorithm has to wait for at least two frames before searching for the counter, by comparison of the two successive frames.

In the next section a suitable synchronization word is chosen, by looking at popular words used for synchronization purposes.

3.2 Synchronization words

The use of a sync word for frame synchronization has the advantage that frame acquisition can be essentially immediate because the sync word pattern is already known at the receiving end. Bernard Sklar [BS, 1998, p461] specified that a good synchronization word is one that has the property that the absolute value of its correlation side-lobes is small. He further stated that the sync word is supposed to be unique, distinct, and not too short to keep the probability of false detections low. Examples of such sync words are the Barker and Willard sequences.

John Fakatselis agrees that to avoid interference, the sync word must be chosen such that it has good auto-correlation and cross-correlation properties, and he mentioned some of the sequences that possess the properties of being good sync words [JF, 1996]. They include Maximum length sequences (m-sequences), Barker Codes, and Willard Codes.

In his paper on short pseudo-random number sequences, Carl Andren, made an in-depth study of correlation properties of Barker sequences [CA, 1997]:

- A Barker word has good auto-correlation side-lobes when preceded and succeeded by all zeros.
- His results shown that the 11 and 13 bit Barker words are good. For longer sequences the mentioned words can be used with added zeros.
- Cyclic shifts of these sequences and reversed sequences are also good.

The 16-bit code made from the 13 bit Barker word (1F35) with 3 leading 0s has a nearly balanced number of 1s and 0s and showed good correlation side-lobes. This code has

uniform side-lobes with a well-defined peak. The sequence is **0001 1111 0011 0101.** This sequence is proposed as the synchronization word for this application.

3.3 Testing the Sync Word of Choice

Firstly a file consisting of random data was generated for this test. The file contained 2 mega bytes of data. The purpose was to see how well the sync word stands out from the random data. How distinct the sync word is, determines how accurate the first sync will be because this tells how immune the sync word is from false detections. It turned out that for all the test cases there was no occurrence of the sync word in the random data.

Secondly the sync word was tested using real data. The raw telemetry data files obtained on SUNSAT's direct link were used, in which the sync word was searched to determine its number of occurrences in the raw data. This time the sync word occurred only once in six files of about 200 kilo bytes each. When one error was allowed in the sync word, the sync word occurred three times in the six files. And there were five occurrences when there was a two-error allowance for the sync word. It can therefore be stated that increasing the error allowance in the sync word results in increased false detection probability for the sync word.

These results prove that the 16-bit code (1F35) made from the 13 bit Barker word with 3 leading 0s is suitable for this application because of its uniqueness characteristics.

3.4 Performance analysis of Synchronization Words

The system performance can be measured in terms of two basic error modes:
Failure to detect a correct sync word and failure to reject an incorrect sync word. The likelihood of these events is termed the **miss probability** and the **false detection probability** respectively [WS&TS, 1968]. For analysis, suppose an N-bit frame that consists of an n-bit sync word followed by N-n random bits is transmitted. The random

bits are either information bits, which vary randomly, or noise of unused parts of the frame.

The equations for the miss and false detection probabilities will be used as they were stipulated in [WS&TS, 1968 & BS, 1998, p463]. If the bit-error probability of the system is p, then the probability **P** that an **n-bit** sync word contains **P** errors is the probability of detection when the detector tolerates up to k errors.

$$P = \sum_{i=0}^{k} \binom{n}{i} p^i q^{n-i} \qquad (1)$$

The miss probability $\mathbf{P_{miss}}$ is then given by

$$P_{miss} = 1 - P = \sum_{i=k+1}^{n} \binom{n}{i} p^i q^{n-i} \qquad (2)$$

The false detection probability $\mathbf{P_{false}}$ is given by

$$P_{false} = \frac{1}{2^n} \sum_{i=0}^{k} \binom{n}{i} \qquad (3)$$

The false detection probability $\mathbf{P_{false}}$ decreases with increasing sync word length while unfortunately the miss probability $\mathbf{P_{miss}}$ increase with increasing sync word length. On the other hand, a sync word too long is not a good idea as it adds to the overhead of the system, thus reducing the system efficiency. The miss probability can be reduced by a certain amount of error allowance k in the sync word pattern.

Figure 3.1 shows the performance of the chosen16-bit Barker code. The false detection probability is plotted against different values of bit error rate (BER), using equation 3.

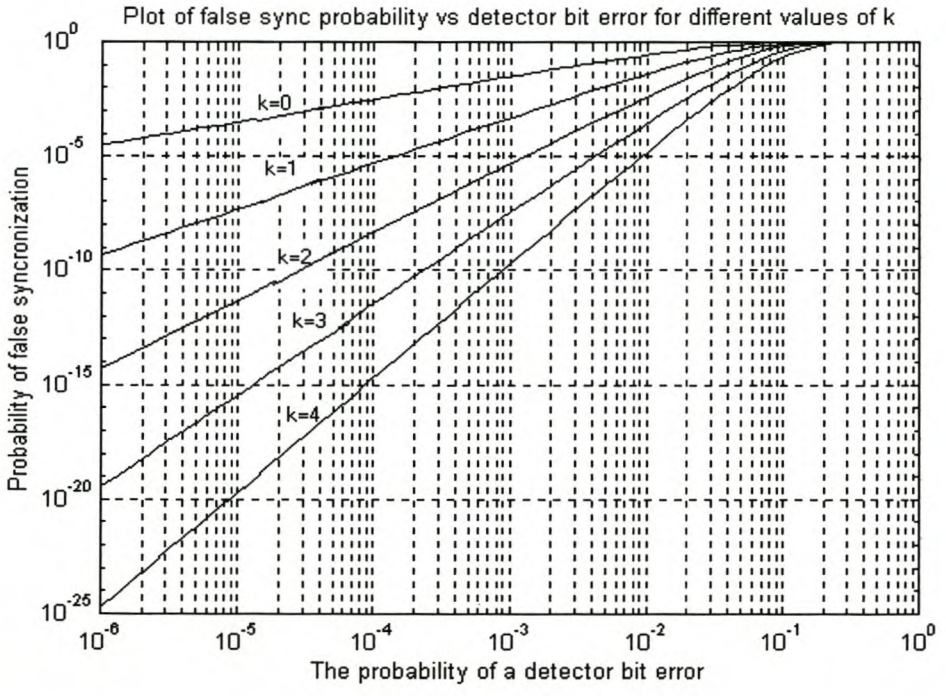

Figure 3.1: *False sync probability v/s bit error rate*

At a BER of 10^{-6} , if no error is allowed in the synchronization word, the false detection probability is 10^{-4} . There is an improvement of about five orders of magnitude when a single error is allowance in the sync word. Therefore depending on the value of k chosen, the false detection probability is reduced accordingly.

3.5 The Synchronization Approach

This section focuses on the design of the synchronization algorithm.

3.5.1 The Frame Synchronization Strategy

In order to improve the sync strategy of the link the following choices are made:

- The frame counter is supplemented with a ***16-bit synchronization word***. This sync word is the 16-bit Barker sequence described in the previous section. The telemetry frame will now have the structure of Figure 3.2:

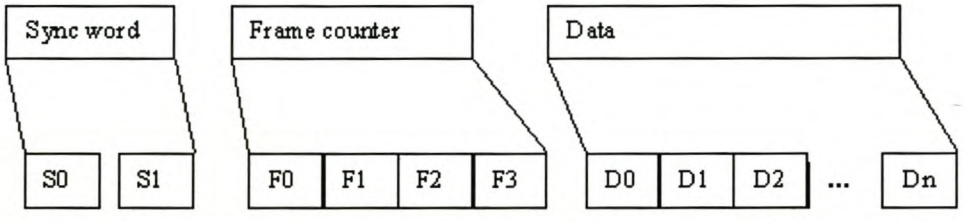

Figure 3.2: *Telemetry frame showing the added synchronization word*

- To avoid the risk of false synchronization in the initial acquisition, **a *two-frame strategy*** is implemented. That is, at least two successive synchronization words must be detected before lock-state is declared.

- Synchronization recovery:
  The sync recovery characteristics are required to be rapid. Apart from reducing the miss probability, the aperture technique also influences rapid resynchronization.

Experimentation with different aperture widths has shown that the false sync probability within the aperture region increases with the aperture width. It is therefore important to restrict the window size so that the false sync probability is kept low. An ***aperture size of four bytes*** is allowed in this application.

- The link analysis has revealed the presence of bit errors in the received data, which will prevent recognition of the sync word pattern if occurred within the sync word. Figure 3.1 has shown that an error allowance in the sync word reduces the false detection probability. Experimentation with error allowance values between 0 and 5 has shown that a value of **k = 2** is sufficient.

The frame synchronization strategy is based on separating the sync algorithm into three distinct modes:

- In **search** mode, the synchronizer looks for the sync pattern in the incoming data stream.

- One strategy that aids the first sync-acquisition is **confirmation**. This involves the verification of the sync word for some N number of times after its first recognition. In this mode, after a pattern is tentatively identified in the search mode, a **window** is set at a predicted time of reoccurrence of the sync pattern, and the masked sync pattern is checked for several frames. If the pattern recurs in the sync window for a prescribed number of frames, the synchronizer advances to lock.

- In **lock** mode, the synchronizer continues to look for the frame sync pattern in the sync window, and will only revert to a previous mode if the sync pattern fails to occur in the window for a given number of frames.

There are three distinct problems with synchronization [HS et al, 1969] and the solutions implemented to address these problems are described below:

- *The first synchronization might be false due to lack of spacing of the data as compared to the sync word.*

  The Barker sequence chosen for the sync word will solve this problem. The frame counter can be used as additional synchronization information to supplement the sync word.

- *There is a possibility that a bit error occurs in the sync word resulting in failure to synchronize on that message.*

  The bit error allowance enables the recognition of the sync word pattern even if some allowed number **k** of bit errors have occurred in the sync word. The window strategy will reduce the false alarm probability and significantly narrow down the time interval over which the sync word is searched.

- *There is a possibility that either a data word or part of the message might look like a sync message, resulting in false synchronization unto that message.*

  The confirmation strategy helps to reduce the probability of falsely synchronizing on a word that just happen to look like the sync word and thus saves time that may be wasted trying to re-sync because the initial detection was false.

3.5.2 The Design of the Synchronization Procedure

The synchronization algorithm is structured as a state machine. Figure 3.3 represents the state transition diagram of the synchronization procedure.

- **Search-state**

  The software searches for the sync sequence in the data stream. This involves comparing every successive two bytes received, (referred to as received sync word (RSW) with the locally stored sync word (SSW)), until a match is obtained. The received sync word must have no bit errors in order for the software to leave the search-state.

- **Confirmation-state**

  On finding the valid sequence, the confirmation-state is entered. Prior to locking up, a second sync word confirmation must be performed. This prevents false lock up caused by accidental coincidence of the sync word with random information. A window is targeted wherein the comparison is restarted. Any invalid sequence in the confirmation-state returns the program to search-state.

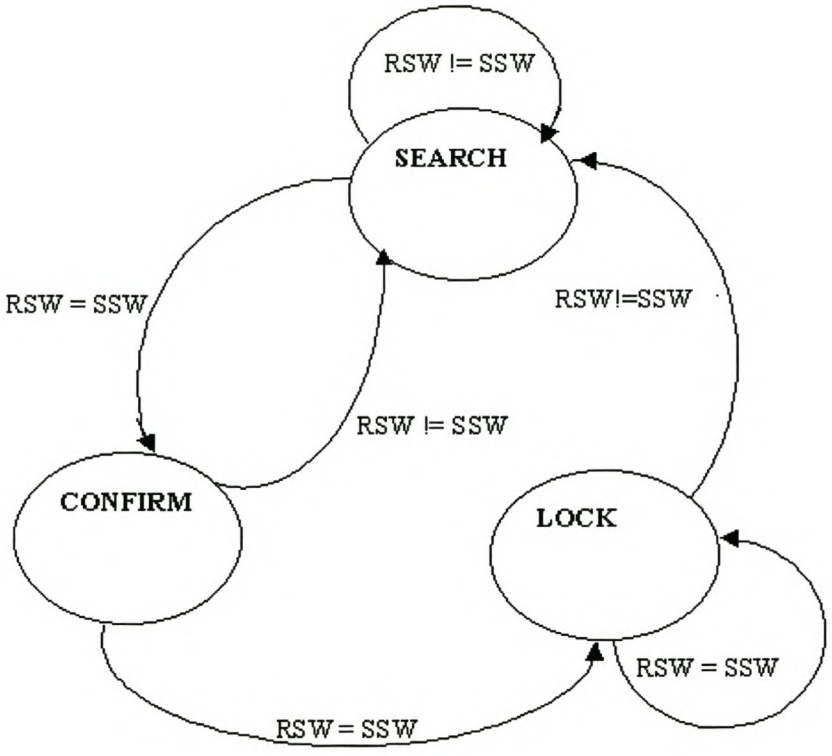

Figure 3.3: *Synchronization State transition diagram*

- **Lock-state**

If a second valid header is seen after entering the confirmation-state, then a sync condition is declared and the program proceeds to a Lock-state/Steady-state. In this state, comparison of the sync word continues within a defined window. Any invalid sequence in the lock-state returns the program to the search-state.

3.6 The Implementation and Testing of the Sync Algorithm

3.6.1 Software development

Appendix A. contains the source code listing of the synchronization procedures implemented in Java programming language.

To develop the synchronization software, two PC's connected together by a serial cable were used. One computer was sending serial data that resemble the telemetry data, i.e. frames that contain a sync word, a frame counter and random data. The second computer ran the synchronization software included in appendix A.

The first step was to generate the file on which to synchronize on. The file consists of frames of bytes. Each frame is 258 bytes long. The first two bytes in the frame are a synchronization word (the 13 bit Barker sequence, 1F35, with three leading zeros. Byte_3 to byte_6 represent the 4-byte frame counter, incremented by 1 for every frame. The remainder of the frame is a series of 252 random bytes.

3.6.2 Testing and Results

Logic state analyzer results are included to illustrate the effectiveness of the synchronization procedures:

- When perfect frames were send, perfect synchronization was achieved on the very first frame coming in and the computers remained synchronized for the rest of the communications.

  Figure 3.4 shows a series of telemetry frames being received. The sync algorithm starts in the search-state in the middle of the first frame. When the second frame is received the algorithm switches to the confirmation-state after which the lock-state is entered at the end of the second frame.

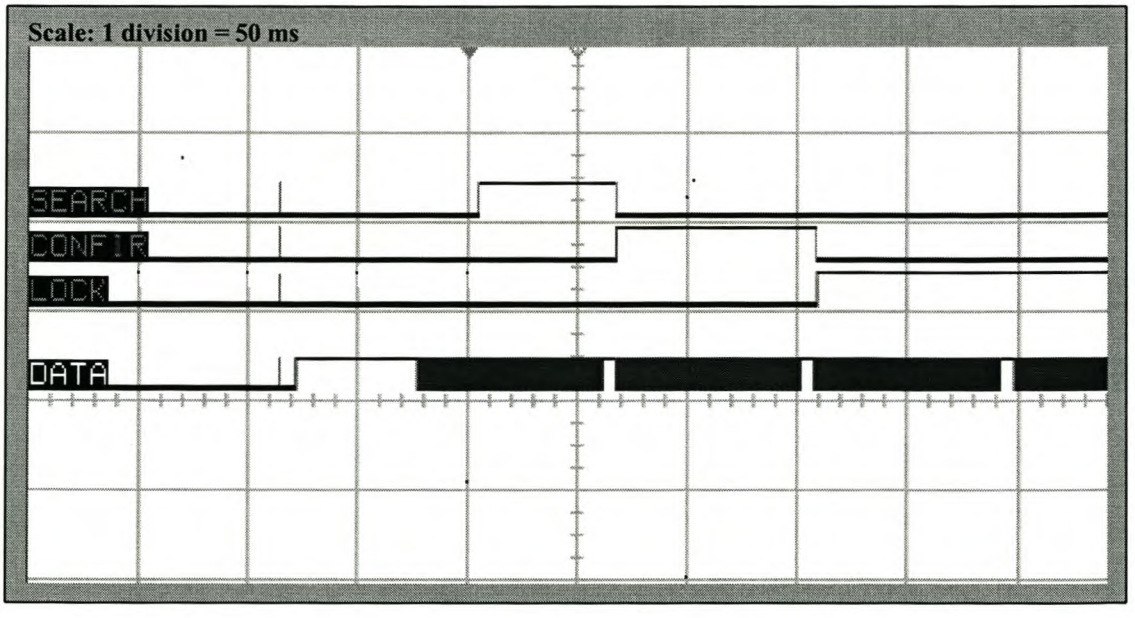

Figure 3.4: *First synchronization*

- Then corrupted data was sent to test how quick the synchronization algorithm returns back to synchronization if there is synchronization loss. The software has the ability to get back to serach-state on the very next frame after a synchronization loss. Figure 3.5 shows how immediate is the transition of the algorithm from the lock-state to the search-state and subsequently returning to the lock-state at the end of the second frame after a sync loss.

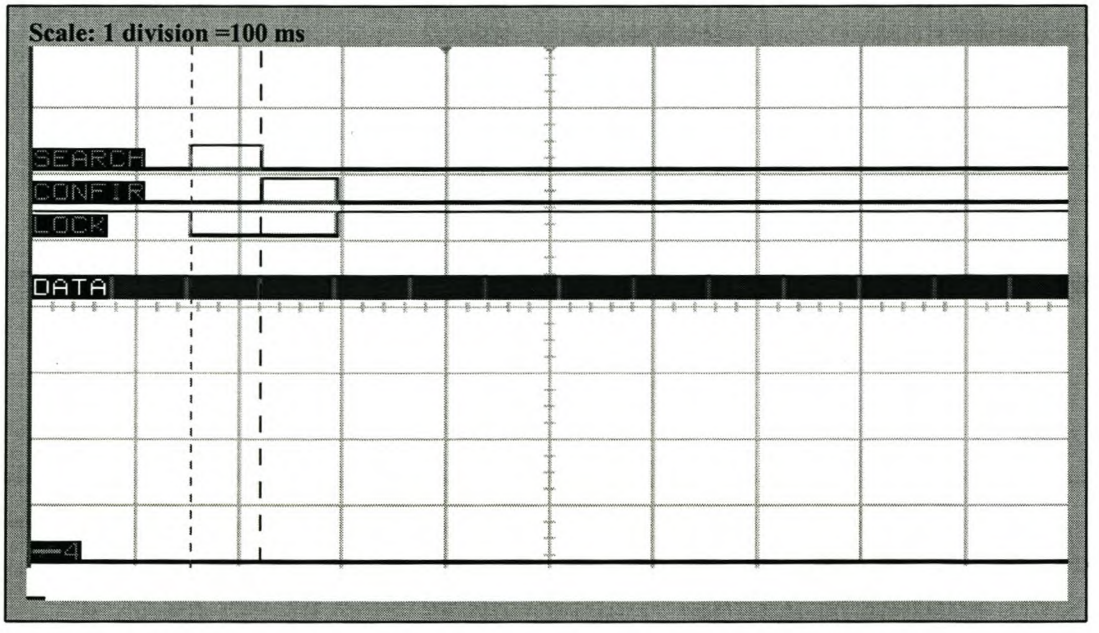

Figure 3.5: *Synchronization loss*

- Figure 3.6 illustrates the effort of the sync algorithm during bad communications. In the second set of pulses, it is shown that if the confirmation-state fails to detect a second sync word, the algorithm jumps back to the search-state to re-initialize the sync algorithm.

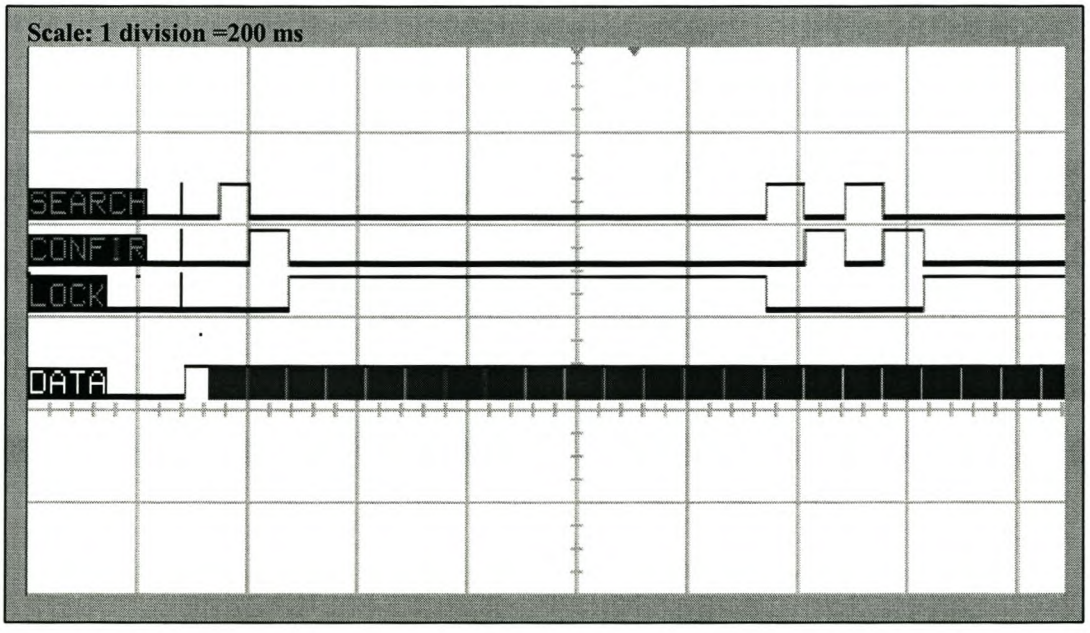

Figure 3.6: *Performance in the presence of errors*

3.7 Conclusion

The shortcomings of the current ground synchronization were identified. It was further shown that to improve the synchronization performance, the use of a synchronization word with the essential characteristics is necessary. A new synchronization strategy was designed using the 13-bit Barker sequence with three leading zeros. The results of the sync algorithm shown the quick initial acquisition and re-sync capabilities of the sync algorithm.

3.8 References:

[H.S et al, 1969] H. Saki, T. Murayana, M. Hashimoto, H. Kanzaki, and Y. Sakamoto, "Burst Synchronization in TDMA Systems", Intelsat/IEEE International Conference on digital satellite communications, Vol 25-27, Nov 1969.

[W.S&T.S, 1968] W. Schrempp and T. Sekimoto, "Unique Word Detection in Digital Communications", IEEE Transactions on Communication Technology, Aug 1968.

[J.L.M, 1972] J.L. Massey,"Optimum Frame Synchronization", IEEE Transactions on Communications, Vol 20, No. 2, April 1972.

[J.F, 1996] John Fakatselis," Processing Gain for Direct Sequence Spread Spectrum Communication Systems and PRISM®", AN9633, August 1996, (http://www.sss-mag.com/pdf/pggap.pdf.)

[C.A, 1997] Carl Andren, "Short PN Sequences for Direct Sequence Spread Spectrum Radios", 4 Nov 1997, (http://www.sss-mag.com/pdf/pggap.pdf.)

[I.dS, 1994] I. De Swart, "The Design of a Reliable and Flexible Telemetry System for SUNSAT", June 1994.

[B.S, 1988] Bernard Sklar, "Digital Communications, Fundamentals and Applications", 1988, Prentice Hall.

CHAPTER 4

4 DIRECT LINK PROTOCOL

In this chapter the communication protocol for the direct link is proposed. This section is labeled 4 in Figure 1.2. The purpose of the protocol is to permit the on-board data system to transmit data bytes over a space-to-ground communications channel in a way that enables the ground system to recover the individual data bytes with high reliability and provide them for ground-station use.

Firstly the layering concept is used to summarize the structure of the current protocol, which then serves as a base for the proposed protocol.

4.1 Protocol layers

In 1979 the International Standards Organization (ISO) published a seven-layer Open Systems Interconnect (OSI) model for organizing emerging network communication protocols [Z&H, 1980]. This model separates various functions of communication protocols into layers so that there is an agreed concept of what functions are to be performed and what the interfaces are between these functions.

The layering concept is illustrated in Figure 4.1, but only the data link and physical layers are shown. Furthermore, because a key design goal of telemetry is efficient use of

limited space link resources [CCSDS, 1987], the telemetry data units will be structured differently from those of OSI protocols.

Encapsulation is the idea that protocols at succeeding lower levels of the OSI model wrap or encapsulate data coming down from higher levels with information that the lower protocols need to perform their functions [GGP, 1993]. The encapsulation of the telemetry data is illustrated in Figure 4.1. Typically, each protocol layer adds a header containing information that will be used (and removed) by its counterpart at the receiving end.

The current protocol is operating only on the physical layer, where the collected telemetry data is send as raw data without any additional information. This has a negative impact on the reliability of the data due to lack of coding strategies like adding synchronization information and error control.

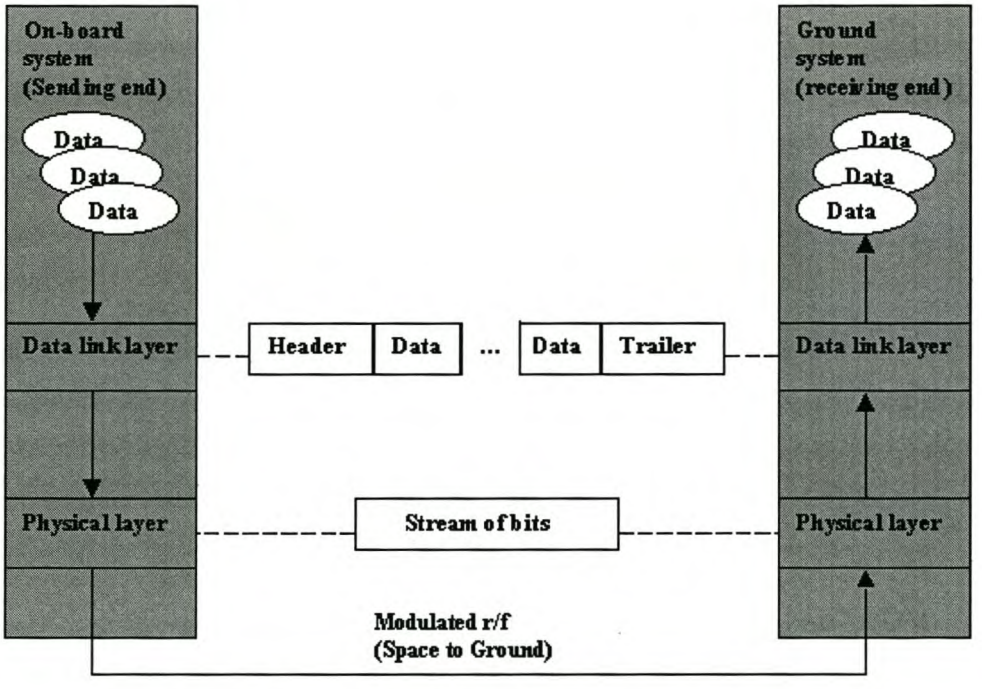

Figure 4.1: *Telemetry protocol layers*

In an attempt to improve the data reliability the data link layer as illustrated in Figure 4.1 is added to the protocol structure. The succeeding sections of the chapter decide on the specific information that will be contained in the header and trailer fields added to the telemetry data in the data link layer.

4.2 Protocol Elements

Five elements in a protocol definition were specified in [GJH, 1991], and four of these essential elements are used here to specify the communication protocol. Namely, services provided, execution environment, description of message used to implement the protocol and message encoding.

4.2.1 The Services Provided

- The protocol should provide a reliable data transfer, by ensuring that transmission errors are detected. Messages with transmission errors are discarded at the receiving station, and no acknowledgements are send back.

- The services specified in this protocol are unidirectional services: one end (the spacecraft) can send, but not receive, data through the protocols providing the service, while the other end (on the ground) can receive, but not send.

- These services are also unconfirmed services: the sending end (spacecraft) does not receive confirmation that data it sends has been received. This is a consequence of the design of the space link protocols, which avoid the delays involved in confirmed services [CCSDS, 1996,p2-2].

4.2.2 The Execution Environment of the Protocol

The environment in which the protocol is executed consists of a data collecting and transmitting hardware and a data receiving hardware as well as a transmission channel. That is a point to point protocol. The channel is a satellite link with a distance of approximately 450km to 860km. Therefore the minimum time for a message to travel from the aircraft to a ground receiver is 1,5 sec and the maximum time is 2.9 sec. Transmission is character oriented at 1200 baud. The data flow diagram of Figure 4.2 shows the path of the telemetry data from the data sources onboard the satellite, until it is received and decoded at the ground station.

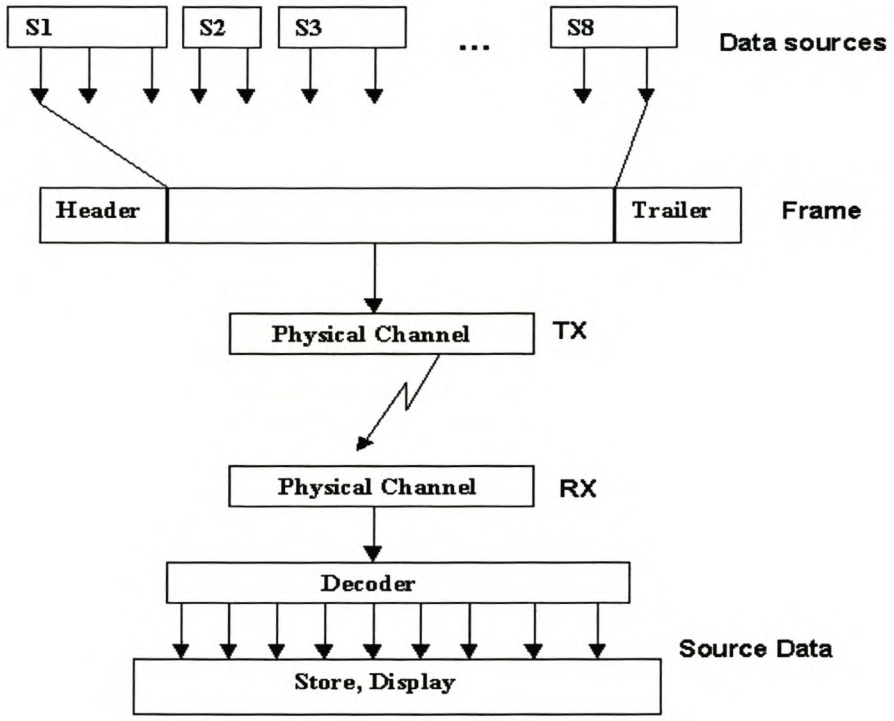

Figure 4.2: *Telemetry data flow*

Telemetry collected from different data sources on the eight satellite subsystems, represented by S1 to S8, is packaged into a frame by adding a header and a trailer and transmitted through a space link and are received and decoded at the ground station. The long distance traveled by the data directly influences the nature of the protocol.

4.2.3 The Description of Messages Used to Implement the Protocol

The protocol defines only two kinds of messages, WOD-frames and direct-frames. Raw data bytes coming directly from data sources (a total of 252) are packed into a frame referred to as a **direct-frame**. Alternatively the data can be read from an external storage place (a total of 256, including a four-byte frame counter) where **WOD-frames** are stored. The frame is a data structure that provides an envelope for transmitting data packets over the noisy space-to-ground channel. It carries additional information in the form of headers and trailers. The contents of these headers and trailers are described in section 4.2.4. All frames are of fixed length.

4.2.4 The message encoding

Every message contains two parts: a message part and a control part. The control part contains redundant information to help the ground management of the data. The frame format is shown in Figure 4.3. It consists of the following fields:

Figure 4.3: *Frame structure*

- **Sync-Word - SW**
  This is a 16-bit Barker code appended to the beginning of every frame, used by the receiving station to synchronize on incoming frames.

- **Frame counter - FC**
  The frame counter holds a 4-byte number to keep count of the frames being sent. The counter provides time information on when a frame was collected.

- **Frame ID - ID**
  The frame ID indicates the type of the frame being received. A "0" identifies a WOD-frame and a "1"identifies a direct-frame.

- **Data field - D0...Dn**
  Telemetry data are contained in this field. A total of 252 bytes are transmitted.

- **CRC field - CRC**
  The CRC result is calculated from the rest of the information contained in the previously mentioned fields and is the last to be transmitted. It is a 16-bit value that is calculated while the bytes are being sent using the encoding procedure outlined in section 4.3.1.

4.3 Error detection

No communications medium is completely error free, so error detection is the key to reliable communications [GGP, 1993]. If an error occurs and is detected, the receiver will have an option not to use erroneous data and steps can be taken to correct it. Error-detection schemes are based upon the transmission of redundant information and the detection of inconsistencies in the received data. The most common error detection

strategy uses a **checksum.** This checksum is appended as the last byte of a data frame and is defined to be the two's compliment of the sum of the values of all other bytes in the record modulo 256. That is, the sum of all bytes in a frame modulo 256 added to the checksum byte's value should yield a value of zero. If it does not, either the checksum byte or some other byte in the frame must have been corrupted. This is, of course, not a foolproof method of detecting corruption. If two bytes in the frame are changed in such a way as to compliment each other the corruption will remain undetected by the checksum [SG, 1999].

Improved error detection schemes uses cyclic redundancy checking (CRC). A CRC computes a result based on both the value and position of individual bits in a block of data using polynomial arithmetic. Many different CRC polynomials are possible; these generator polynomials are designed and constructed to have desirable error-detection properties [TR, 1986]. The major difference between the CRC is in their length. Longer polynomials provide more assurance of data accuracy and are fully usable over larger amounts of data; however, longer polynomials also produce longer remainder values, which add additional error-checking overhead to the data.

For the 261-size frame in this work, the 16 bit CRC seems to be sufficient. Two popular CRC polynomials are CRC-CCITT and CRC-16. Faster CRC computations uses the table look-up strategy. However where the CCIT needs 256 distinct entries in the table, the CRC-16 can work with a much smaller table of 16 entries [SG, 1999]. Therefore the CRC-16 polynomial is chosen to allow faster computations with a look-up table while saving on the limited 2 kilo-byte code memory of the 80C31 processor.

4.3.1 Encoding procedure

The encoding procedure accepts an (n -16)-bit frame, excluding the CRC field, and generates a systematic binary (n, n -16) block code by appending a 16-bit CRC field as the final 16 bits of the data block. The divisor polynomial used in this calculation is the CRC-16 polynomial, $x^{16}+x^5+x^2+1$.

4.3.2 Decoding procedure

The receiver can compute the CRC of the data bits, and compare it to the transmitted CRC bits. Less obviously, but more elegantly, the receiver can simply compute the CRC of the total block, with n + 16 bits, and verify that a result of zero is obtained.

4.3.3 Channel error performance

Figure 4.4 compares the error detection performance for a communication link, with no error detection, the case where a single bit error can be detected and the case where three or more errors are detected.

The undetected error rate is reduced tremendously when a three-bit error detection scheme is implemented, as compared to the case when no effort is made to detect errors. The CRC error detection will have even a lowered undetected error rate because of its ability to detect various combinations of errors.

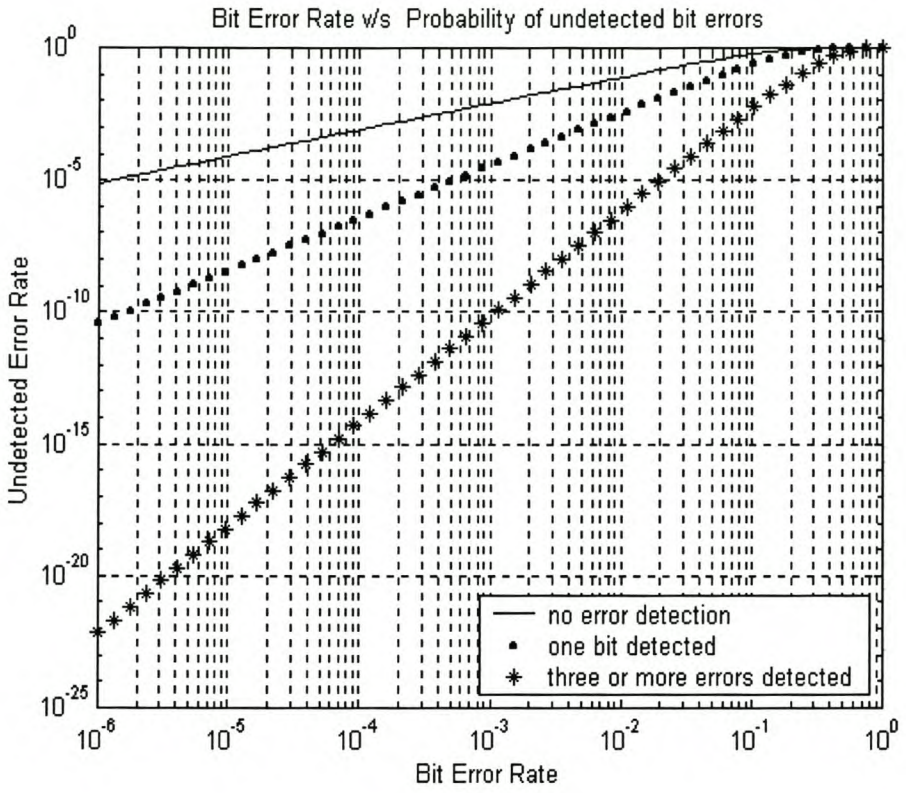

Figure 4.4: *CRC performance*

A 16-bit cyclic redundancy code detects any contiguous burst of errors shorter than the polynomial, any odd number of errors throughout the block, any single- and double-bit errors anywhere in the block, and most other cases of any possible errors anywhere in the data. It ensures detection of 99.998% of all possible errors. So every possible arrangement of 1, 2, or 3 bit errors will be detected. This level of detection assurance is considered sufficient for data transmission blocks of 4 kilobytes or less [TR,1986]. For larger transmissions, a 32-bit CRC is used.

4.4 Conclusion

A frame format for use on the direct link was produced. Because of the need for error handling a CRC field is incorporated in the frame for error detection. The synchronization word chosen in chapter 3 is also included to mark the start of every frame. The implementation of this protocol is reflected in the software in chapter 6.

4.5 References:

[TR, 1986] T. Ritter, "The Great CRC Mystery". Dr. Dobb's Journal of Software Tools. February. 11(2): 26-34, 76-83, 1986. (http://www.io.com/~ritter/ARTS/CRCMYT.HTM.)

[GGP, 1993] G.G. Preckshot, " Data Communication Systems In Nuclear Power Plants" , May 28, 1993.

[Z&H, 1980] Zimmermann, Hubert, "OSI Reference Model—The ISO Model of Architecture for Open SystemsInterconnection," IEEE Trans. on Communications, Vol. COM-28, No. 4, April 1980, pp. 425–432.

[CCSDS, 1987] "Packet Telemetry Services, CCSDS 103.0-B-1, Blue Book", December 1996. (http://www.ccsds.org.)

[SG, 1999] Scott Gasch, " Algorithm Archive (Work in progress)", 1999 (http://www.softlab.od.ua/algo/other/alg/alg.html.)

CHAPTER 5

5 TASK SCHEDULING FOR THE 80C31 MICRO-CONTROLLER

The task scheduler on SUNSAT 's telemetry micro-contoller is an event driven scheduler that waits for OBC requests and immediately responds to the request. These events (interrupts) are asynchronous and are disabled when requests are being serviced. The extension of the telemetry tasks in this thesis includes tasks that are time-driven. That is tasks that are executed at a given time period apart. This behavior is not accommodated in the current scheduler.

The aim of this chapter is to design a task scheduler with the ability to schedule both synchronous and asynchronous tasks.

5.1 Scheduling in Real-time Systems

Scheduling involves allocating resources and time to activities so that a system meets certain performance requirements. The system can have the properties of either a real-time system or a non real-time system. Real-time systems depend not only on the results of a computation, but also on the timeliness of the results. Timing constraints for

tasks can be arbitrarily complicated but the most common timing constraints for tasks are either *periodic* or *aperiodic* [SR, 1993]. An aperiodic task has a deadline by which it must finish or start, or it may have a constraint on both start and finish times. In the case of a periodic task, the *period* of the task becomes important. A period might mean "once per period T", or exactly T units apart.

The timeliness of tasks allows the classification of a real-time system as **hard real-time** or **soft real-time**. A system where the violation of a deadline causes it to fail, is known as a hard real-time system, while the missing of a deadline on a soft real-time system only leads to degraded system performance [PAL, 1997]. In other words a hard real-time task must be completed always by a specified deadline. Soft real-time applies to those systems that are not hard real-time, but some sort of timeliness is implied. That is, missing a deadline will not compromise the system's integrity.

5.2 System Characteristics

In his work on the flight software development for SUNSAT, N. Steenkamp maintains that, because of the structure and mission of SUNSAT the software for the satellite is considered to be a *soft* real-time system [NS, 1999]. As a result the system under consideration is regarded as a ***soft real-time system.***

In the current telemetry application, when an OBC is collecting telemetry data from the TMS, it is interrupted every 10ms when the system is running at 1200 Baud or every 1.25ms when running at 9600 Baud [JAK, 1999] which is a heavy interrupt load on this processors. The idea of implementing a micro-controller with data collection capabilities was aimed at relieving the OBC 's from this interrupt load. However, because of the lack of reliability of the direct link channel the 80C31 has not been utilized to its capabilities.

With the new improvements of the link, the 80C31 can be used as the default data collection and transmission system. As a result, other than collecting data continuously at the maximum sampling rates of 1200baud and 9600baud, a timer can be used to space the telemetry data as it is required. The OBC can command the 8031 to collect WOD or to send telemetry frames with a given spacing between the frames other than the continuous sampling used by the OBC's. This result in a new definition of tasks: **Short tasks** meaning those that are requested and executed once, and **Long tasks** as those that are requested and then repeated at a time tick until they are stopped by a command.

Thus the system consists of a set of pre-defined, specific tasks that are both ***periodic*** and ***aperiodic*** in nature. The system is both ***event-driven*** and ***time-driven***. It reacts to external events in the form of serial interrupts, which convey messages on which the system must react. An event can also be internal in the form of a timer tick to signal the start time of a specific task.

A listing of the tasks to be handled by the system is given in Table 2 in section 5.4. The tasks are classified in terms of periodicity and urgency. In the next sections a scheduling method for these tasks is defined in order that the overall system performance objectives are met. There are varieties of scheduling algorithms that have been developed, depending on the end application. One algorithm is chosen to satisfy the response requirements of the system.

5.3 Scheduling paradigms

Various scheduling designs relevant in the embedded real-time world are discussed in [SKJS, 2001]. They include static scheduling, round robin scheduling, static cyclic scheduling, dynamic scheduling and synchronous scheduling.

Dynamic scheduling can be **preemptive** or **non-preemptive**. A non-preemptive (cooperative) scheduler trusts each task to voluntarily release control of the processor on a periodic basis.

Most widely used real-time scheduling schemes are **Earliest Deadline First** and **Rate Monotonic scheduling** [AS, 1998].
Earliest Deadline First (EDF): By this scheme a task with the closest deadline to complete is given the highest priority to run. The priority decreases with the passing of the deadline and increases with the approach of the deadline. Hence it is a dynamic priority, scheduling scheme.

Rate Monotonic Scheduling (RMS): In this scheme the priorities of the tasks are proportional to their rate of execution. It is a static priority, scheduling scheme, if the periods of the tasks are known and do not change.

Paul Pop [PP, 2000] recognizes that event-triggered systems typically require *preemptive priority-based scheduling*, where the appropriate process is invoked to service the event. In the event-triggered approach the activities happen when a significant change of state occurs. The significant events are brought to the attention of the CPU by the interrupt mechanism. Paul further recognizes that time-triggered systems typically require *non-preemptive static cyclic scheduling*, where the process activation or message communication is done based on a schedule table. In the time-triggered approach the activities are initiated at predetermined points in time.

Because the system-tasks are both event triggered and time triggered, a scheduling algorithm that caters for these two task types is designed. That is an algorithm with the properties of both preemptive priority-based scheduling and non-preemptive static cyclic scheduling, according to the system requirements.

5.4 Choice of scheduling strategy

5.4.1 Approach

Due to the nature of the TLM system and its expected functionality presented in section 5.2, a **non-preemptive, dynamic scheduling** algorithm is used.

The non-preemptive scheduling is chosen because of the soft real-time nature of the system. Furthermore non-preemptive schedulers are easier to implement and analyze. It has advantages of saving extra context switches and save overheads involved in supporting mutual exclusion over preemptive scheduling [SK&JS, 2001]. The problem with non-preemptive scheduling is that it can cause some tasks to miss their deadlines as non-preempted tasks can have varying release times, but this does not threaten the system performance in this case as there are no strict deadlines to be met.

The dynamic scheduling [FB et al, 1998] policy is based on priority. The tasks are dynamically chosen based on their priority from an ordered prioritized queue. The priorities are assigned statically based on different criterions like, deadline, criticality, periodicity etc. Table 5.1 illustrates the system tasks with their classification and the resultant allocated priorities. A description of all the tasks is given in Appendix B.

Table 5.1: *Task characteristics and priority allocation*

TASK	APERIODIC	PERIODIC	URGENT	PRIORITY
1. SendSerial B	X			2
2. SendSerial C	X			2
3. SendParallel B*	X	X		1
4. SendParallel C*	X	X		1
5. Stop	X			2
6. SendParallel B	X	X		1
7. SendParallel C	X	X		1
8. TcmUpdate	X		X	3
9. TcmRequest	X			2
10. TcmReseted	X			2
11. RamTest	X			2
12. SendDirect	X	X		1
13. GetWod	X	X		1
14. SendWod	X	X		1
15. GetDwell	X			2

The tasks marked with a (*) operate at 9600-baud. All the tasks are aperiodic because they are initiated by serial interrupts. The tasks that are marked to be periodic are those tasks that are recursive. i.e. after the initial serial invocation, the tasks are repeated at specified periods apart.

As each task occurs in the system the existing schedule is checked to see if the new task can be added to the schedule so that it's deadline can be met. The rate monotonic scheme described above is used. Priorities are static with the "short tasks" assigned a high priority and the "long tasks" with the lowest priority.

The advantage of priority is that, the latency (response time) for important tasks can be reduced, by giving those tasks a high priority. Two or more tasks can have the same

priority. A priority 1 task is run only if no priority 2 tasks are ready to run. Similarly, we run a priority 3 task only if no priority 1 or priority 2 tasks are ready. If all tasks have the same priority, then the scheduler reverts to a round-robin system.

5.4.2 Functional Description of the Scheduler

As a minimum requirement, a real time operating system (RTOS) must provide three specific functions: task dispatching, task scheduling and inter-task communications [Lap97, p.142]. The three basic functions are discussed and related to their functionality in the TLM system scheduler design.

- **The task dispatcher**

  Task dispatching refers to the ability to transfer execution from one process to another process [NS, 1999, p.30]. There are two elements in this action: halting execution of one task, and initiating execution of a second. Because of the co-operative nature of the scheduler, a task cannot be interrupted by another task until it releases control of the processor. The complexity of the dispatching process in non-preemptive scheduling depends on whether the tasks are independent and whether there are resource constraints [PP, 2000]. The tasks in this application are independent but memory management is required to curb information overriding.

- **The scheduler**

  The interrupt-driven tasks are enabled inside the ISR to catch the requests even if the processor is busy. The scheduler schedules only the time triggered tasks. It is the so-called *tick scheduler* [PP, 2000]. The tick scheduler is activated periodically by the timer interrupt and decides on activation of processes, based on their priorities. The scheduler makes use of the *list scheduling based algorithm* [PE et al, 2000].

List scheduling [PE et al, 1998] is based on a priority list from which processes are extracted in order to be scheduled at certain moments. The *ReadyList,* contains the processes that are eligible to be activated. Based on the co-operative scheduling algorithm the scheduler arranges tasks for execution as a set of *ready* tasks. A task is said to be ready if it requires to be executed. The dispatcher is then signaled to begin execution of the new task.

- **Inter-task communications**

Inter-task communications between tasks is facilitated by the use of semaphores to achieve synchronization. The tasks are synchronized to ensure that every second task is initiated only when the first executing task has completed. Each task has a pre-amble in which a global semaphore is set and a post-amble in which the semaphore is cleared. The dispatcher monitors the state of this semaphore and initiates the running of a new task when the semaphore is cleared.

5.4.3 Task states and transitions

Tasks may be in one of three "task states": **Ready**, **Running**, and **Blocked**. The task state transition is illustrated in Figure 5.1. A task may be in one and only one of these states at any given time.

Ready: All tasks that are not **blocked** or **running** and that are ready to run are **ready**.

Running: Only one task is **running** at a time. This is the task that is currently being executed by the processor.

Blocked: A **blocked** task is a task that is waiting for an event like a time-out, message or a signal.

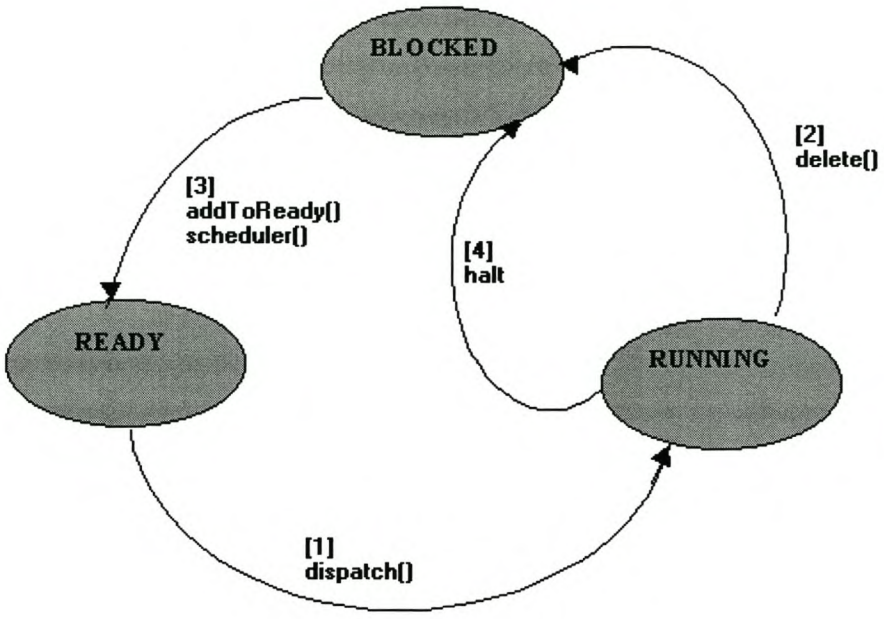

Figure 5.1: *Task State transition diagram*

Transition	Description
[1]	A task in the **ready** state is selected for execution. The scheduler changes the task's state to **running** and resumes execution of the task. The previously running task is switched to the blocked state. This happens only when the previously running task has run to completion.
[2]	A task in the **running** state is changed to the **blocked** state. When a running task completes a task is returned to the blocked state and is deleted from the ready list.

[3] A task in the **blocked** state is changed to the **ready** state. When a task is waiting for an event (signal, time-out, message, or semaphore) it is in the blocked state. When the event occurs, the task is switched to the ready state.

[4] A task in the **running** state is changed to the **blocked** state. This occurs when a low priority task halts for a given time during its run to allow a high priority task to run if any. It resumes when the time expires.

5.4.4 The scheduling algorithm

The scheduling algorithm used to determine the next task to run is as follows:

i. The scheduler performs non-preemptive multitasking **using** an event-driven task-switching scheme with priorities. Basically, the task with the highest priority that is ready gets to run.

ii. If a request for a task is received, the requested task is added to the ready list inside the ISR. The ready list holds tasks arranged in order of increasing priority from the top of the list. The highest priority task is called to run if there is no other task running, or a long task is on hold. All short tasks must run to completion before the next task is called to run.

iii. The tasks with the lowest priority can run for a very long time, and as a result provision is made for such tasks to release the processor at strategic points during their execution, to allow some urgent tasks to run if any have been scheduled.

5.5 Conclusion

Real time scheduling concepts were investigated and the real-time nature of the telemetry system established as well as the scheduling requirements for the system. Then a real time scheduler with the ability of scheduling both synchronous and asynchronous processes was proposed. The implementation of the scheduler as well as its performance is outlined in chapter 6, where it is then integrated with a number of TLM tasks and its performance is evaluated.

5.6 References:

[FB et al, 1998] Felice Balarin, Luciano Lavagno, Praveen Murty, and Alberto S. Vincentelli. “Scheduling for Embedded Real-Time Systems.” IEEE Design and Test of Computers, 1998.

[AS, 1998] Atul Sharma, "NEELPROS: A Predictable Real-time kernel layer Design for Multimedia", 1998

[PAL, 1997] Philip A. Laplante. "Real-Time Systems Design and Analysis: An Engineer's Handbook." IEEE Press, second edition, 1997.

[HK, 1997] H. Kopetz, “Real-Time Systems-Design Principles for Distributed Embedded Applications”, Kluwer Academic Publishers, 1997.

[PE et al, 1998] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, P. Pop, “Process Scheduling for Performance Estimation and Synthesis of Hardware/Software Systems”, Proceedings of 24th Euromicro Conference, 1998.

[PE et al, 2000] P. Eles, A. Doboli, P. Pop, Z. Peng, “Scheduling with Bus Access Optimization for Distributed Embedded Systems”, IEEE Transactions on VLSI Systems, 2000.

[AD&PE, 1998] A. Doboli, P. Eles, “ Scheduling under Control Dependencies for Heterogeneous Architectures”, International Conference on Computer Design (ICCD), 1998.

[CL&JL, 1973] C. L. Liu, J. W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environ-ment”, Journal of the ACM, V20, N1, 46-61, 1973.

[PP, 2000] Paul Pop, " Scheduling and Communication Synthesis for Distributed Real-Time Systems", 2000.

[SK&JS, 2001] Sanjeev Khushu and Johnathan Simmons, "Scheduling and Synchronization in Embedded Real-Time Operating Systems" CSE 221, March 5, 2001

[NS, 1999] Niki Steenkamp, "Development of the On Board Computer Flight Software for SUNSAT 1", 1999, masters thesis, Univ of Stellenbosch.

CHAPTER 6

6 TELEMETRY SOFTWARE

This chapter focuses on the software development process that has been followed to produce the telemetry software. This process is divided into four phases, namely: the software requirement specification (SRS) phase, the software design phase, the implementation phase and the integration and testing phase.

6.1 Software Requirement Specification

In a software process, requirements specification is the first major activity following the completion of a statement of need resulting from the predevelopment process [JP &WP, 2000, p117]. The chapters prior to this chapter formed this predevelopment process by analyzing the telemetry system and deciding on new implementations. The main focus of a requirement specification is on *what* a software system should do, to satisfy the informal requirements provided by the predevelopment process. The IEEE std. 830-1993 for developing a SRS [JP&WP, 2000] is used to produce an SRS for the telemetry software. This SRS is contained in Appendix B.

6.2 Software design

Software design immediately follows the SRS process. The software design process involves determining *how* requirements are realized as software structures. This process is characterized by selecting software interfaces, algorithmic explanation of functions and module interconnections. The software design process is detailed in Appendix C. The task scheduler is designed first, followed by the telemetry functions to be scheduled.

6.3 Software implementation

Implementation comes as the third phase of the software development process. In this process the source code is developed, with the design being verified by verifying functional correctness and validating the source code. Appendix D contains the final source code of the telemetry system.

6.3.1 The software development environment

The software is written in C programming language. C provides the ability to write separate software modules with well defined interfaces. This allows for data encapsulation and information hiding which promotes modular development. As a result different functions can operate independently from each other. Abstract data typing ability allows the creation and manipulation of complex data types that better represent the relevant information.

The Keil C51 Compiler is used for the source generation as well as proper compiling and linking. The Keil environment is well suited for code generation for hardware platforms.

6.3.2 The implementation setup

The setup used for the development of the software is shown in Figure 6.1. It consists of a micro-controller device and a communication interface.

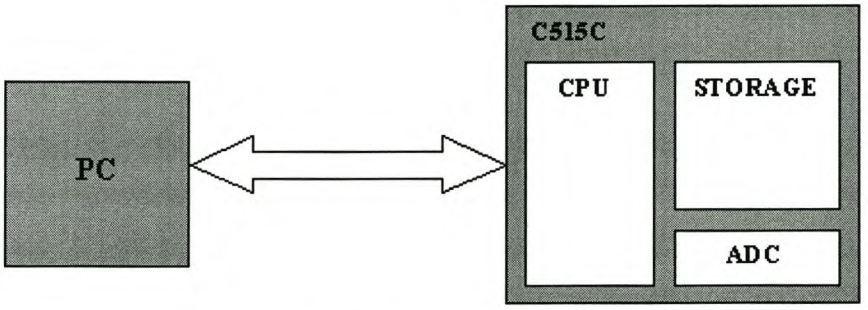

Figure 6.1: *The software implementation setup*

The Siemens C515C micro-controller was used because of the following features, which are required for the telemetry processes:

- It is fully compatible to the architecture of the standard 8051 micro-controller family, which includes the 80C31 micro-controller used for telemetry.
- 256 bytes on-chip RAM, 2KB on-chip XRAM and up to 64KB of external data memory.
- Three 16-bit timer/counters.
- 10-bit A/D converter with multiplexed inputs and built-in self-calibration.
- A full duplex serial interface.

Timer 1 was used for baud rate generation and timer 2 for the time spacing between recurrent tasks. To emulate TLM data, internal analog inputs were introduced to the eight analog inputs of the C515C one at a time where it is sampled with 8-bit accuracy. The data can be stored in external data memory as WOD.

The communication interface is a PC, used to emulate the OBC. The PC is connected to the micro-controller serially, and sends commands using the SSB protocol. The PC was also used for easier debugging of the software during the development phase. Test statements are inserted at target places inside the software to monitor the state of variables during program execution.

6.4 Integration

This is the final phase of the software development process and involves porting of the developed telemetry software into the telemetry system hardware. The integration setup is illustrated in Figure 6.2. The setup consists of the SUNSAT telemetry board, a communication interface and a data interface.

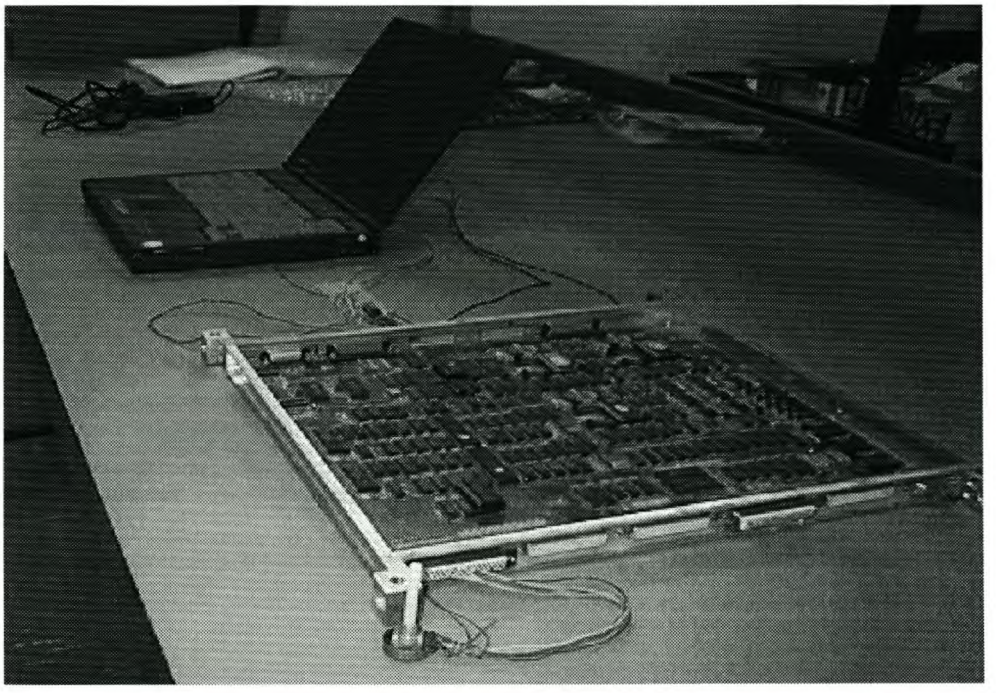

Figure 6.2: *The integration setup*

The same communication interface that is used in the implementation phase is used here with the same functionality as described in section 6.3.2.

Instead of the data being generated internally on the microprocessor, it is now coming from external sources. Multi-turn potentiometers are used to generate the analog inputs for analog to digital conversion. These analog inputs enter the telemetry system through a plug on the telemetry board where they can be sampled for A/D conversion.

The telemetry micro-controller is connected to a 32 kilo-byte external data memory for telemetry data storage. The memory space is organized as shown in Figure 6.3.

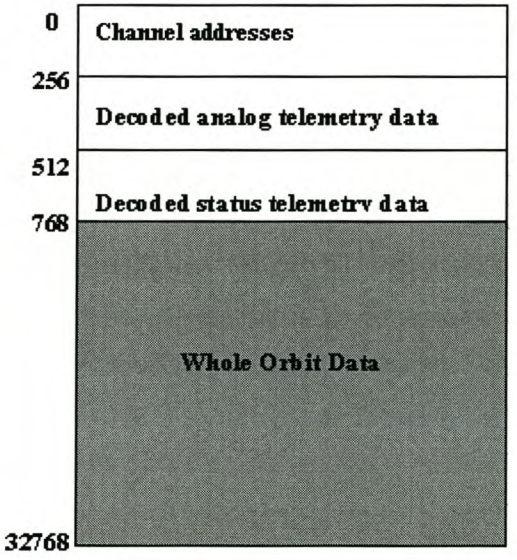

Figure 6.3: *Memory map for external data memory*

The first 768 of external data memory is reserved for storage of ordinary telemetry data in the current telemetry application and are allocated as shown. The remaining memory space indicated by the shaded area is used for WOD storage in this telemetry software.

6.5 Software Testing

Software testing is a continuous process that forms an integral part of the software process. It is carried out at different levels throughout the entire software life-cycle. The PC mentioned in section 6.3 has been particularly helpful in the testing of the software. Four levels of software testing are identified in [JP&WP, 2000, p.444]. They are: component, integration, system and acceptance testing. The first three levels of tests have been covered in this work.

6.5.1 Component level testing

At *component level,* tests verify the implementation of design of a software element (e.g., function, module). The tests that were performed on this level are functional, stress, and structure tests. The specific tests that were carried out to test the functionality and structure of modules are presented below in terms of what was tested, the action taken to perform the test (how) and the results of the test:

ProcessByte (unsigned char rec)

What : correct transition order and recognition of errors in the request frame

How : one or more fields in the request frame were sent with error. For example, incorrect STX or length, different from the actual length of the frame. All the fields were tested.

Result: In all the test cases the request was not carried through.

AddToReady(unsigned char request)

What : requests are actually added to the ready list

How : every time an element is added to the list, the list elements were displayed.

Result: the correct task value was added to the list always.

Sort( )

What : sorting ability

How : every time the list elements are sorted, display them before and after the sorting takes place.

Result: the list elements are always sorted with the highest priority task or the earliest task at index 0.

Dispatch( )

What : ability to execute the right task

How : display the list elements before and after the a call for dispatch( ).

Result: Element at index 0 is chosen and replaced by ' 0'.

delete( )

What : ability to delete the run task from the list.

How : display the list elements before and after the a call for delete( ).

Result: Element at index 0 is removed from the list (the elements are shifted out so that the element at index 1 now occupies index 0.

"Long tasks" (*sendDirect( ), getWod( ) or wodRequest( )*)

What : repetition after a given interval, stopped by command or on completion, interruptible.

How : request a "long task" and either wait for it to complete or send a command to stop it or send request for a different task while it is still busy.

Result: the "long task" stops when completed or stops on request and always gives chance to another task if requested whilst the recursive task is still busy.

The telemetry functions:

What : ability to get data from the data interface, send data on the SSB, store data in external memory, read data from external memory.

How : display the data collected, the data stored in external memory and the data read from external memory.

Result: all the tests resulted in the corresponding data expected from each function.

6.5.2 Integration level testing

At ***integration level*** hardware and software elements are combined and tested until the entire system has been integrated.

Integration was done by porting small pieces of code, until the complete code was ported. The series of tests performed in the component level were repeated here to ensure that the software meets the requirements.

6.5.3 System level testing

Tests at this level ensure that the software as a complete entity complies with its operational requirements. This is the level where performance tests were carried out to measure execution times and response times. Logic state analyzer results to illustrate the software performance are provided:

- In Figure 6.4 a request frame reaches the TLM system via the SSB. As the request is being received it is processed at the same time as shown by PROC. At the end of processing, the request is added to the ready list, shown by ADD. Then the task is executed as shown by Stask that denotes a short task. The message processing lasted for 12.1 ms.

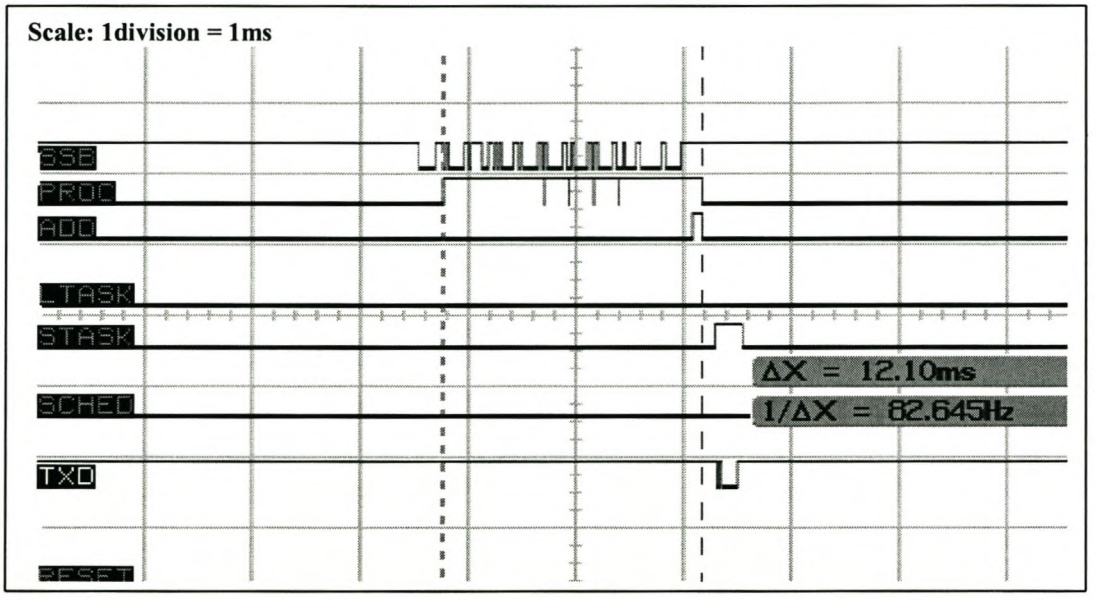

Figure 6.4: *Processing a task request and response*

- In Figure 6.5 a long task denoted by Ltask is rescheduled after a given period. The scheduler monitors the time and adds the task to the list. The long task in this case is *sendDirect( )* which sends one full frame repeatedly through the TXD pin of the microprocessor. The duration of *sendDirect( )* is 2.54 sec.

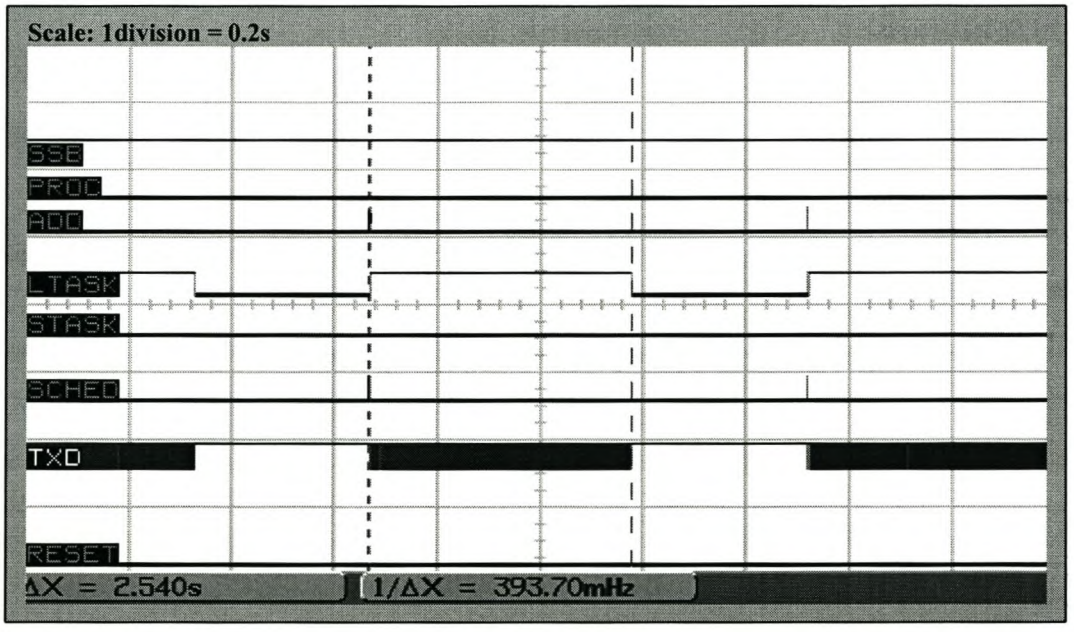

Figure 6.5: *Scheduling of recursive tasks*

- The software allows for the interruption of long tasks. In Figure 6.6 a request is received on the SSB while *wodRequest( )* is executing. The request is processed and added to the list. When *wodRequest( )* releases the processor after sending a full frame, the new task is serviced and in this case it was task 5 that stops the sending of WOD frames, and *wodRequest( )* was subsequently stopped.

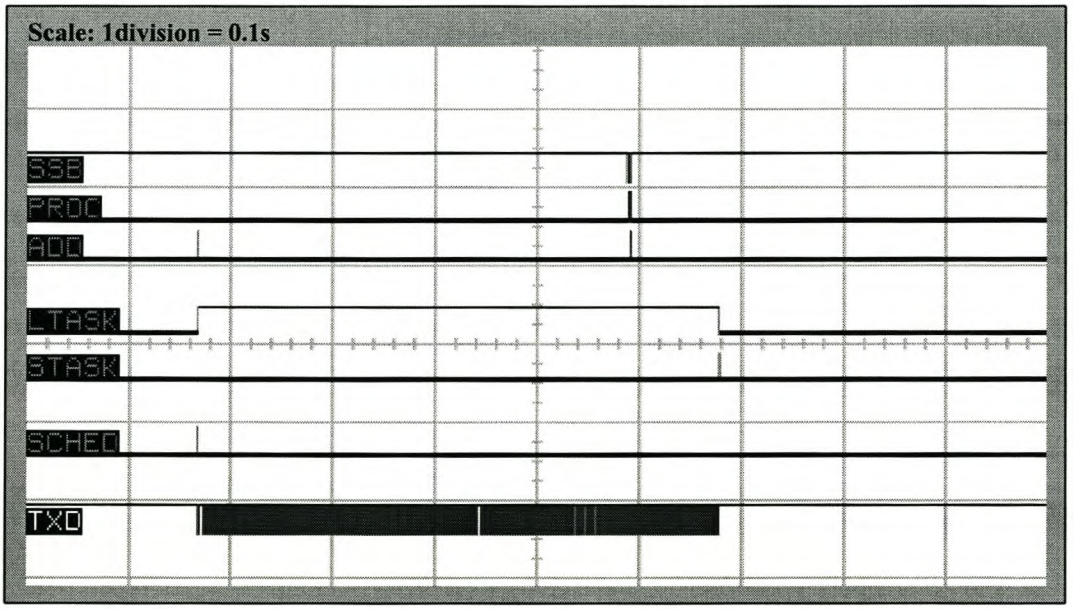

Figure 6.6: *Interruption of a long task*

A summary of the specific tests that were performed together with their results are presented in Table 6.1.

Table 6.1: *Performance results of the telemetry software*

Interrupt execution time:

Start of the ISR ← **174us** → end of the ISR function

Scheduler response time:

Scheduler( ) call ← **18us** → start of scheduler( ) function

Scheduler execution time:

Start of scheduler( ) ← **500us** → end of scheduler( )

Execution times for the different functions:

SendSerial( ) ← **2.6s** →

GetWod( ) ← **2.7s** →

WodRequest( ) ← **2.8s** →

SendDirect( ) ← **2.54s** →

GetCRC( ) ← **80us** →

Due to the fact that a co-operative scheduler is used, and that there are no strict timing requirements for the system, the results obtained are sufficient for the application.

Some of the performance metrics used for RTOS are *interrupt latency* and execution *determinism.* Interrupt latency is the measure of the time it takes before an interrupt is serviced from when the interrupt actually occurs, while determinism means the variance in code execution each time a specific code fragment runs. The two concepts have not been considered because they have minimal effect on system efficiency in co-operative multitasking. The reason for this is that tasks may have varying release times that in turn affect the response times of other tasks.

6.6 Conclussion

The telemetry software was developed by carefully following the different phases involved in software development. The requirement phase produced software requirements based on the predevelopment study that was carried out in the chapters prior to this one. Then a design phase followed to produce the design details of the different software functions to meet the telemetry requirements. In the testing phase the functionality of the different software functions was tested against the specified requirements. Performance tests resulted in measured values of different attributes to display the performance of the system software.

6.7 References:

[JP&WP, 2000] James F. Peters and Witold Pedrycz, "Software engineering, An Engineering Approach", John Wiley & Sons Inc, 2000

CHAPTER 7

7 CONCLUSION AND RECOMMENDATIONS

This chapter concludes the document by summarizing the work done in the thesis in relation to the main objectives. Then the required system modifications and benefits are outlined as well as the impact of the modifications on design.

7.1 Overall Results

Due to the data reliability requirement of telemetry data, the system analysis carried out in chapter 2 resulted in an attempt to improve the coding scheme used on the direct link channel with the aim of making the data more reliable. The main objectives of the thesis were then laid out in section 2.3, and are mentioned here again:

- An improved ground-station synchronization strategy for direct link data.
- An improved communication protocol for use on the direct link channel.
- A telemetry micro-controller scheduling design to meet the new functionality requirements of the system.

The following results based on each objective are presented:

7.1.1 Ground Station Synchronization

- To improve the reliability of the synchronization strategy, the use of a synchronization word with the essential sync word characteristics was identified. A new synchronization strategy was designed using the 13-bit Barker sequence with three leading zeros.

- The actual performance results obtained from the new strategy have shown that initial synchronization can be achieved on the very first frame. There is also rapid resynchronization capability whereby the search for a sync word takes place in the same frame where the sync loss occurred. Performance is further enhanced by the implementation of the two-frame strategy, the sync word error allowance and the windowing strategy.

- Because hardware telemetry uses the same data format as used on the TLM micro-controller, the sync word implementation in hardware must be considered so that there is ease of synchronization on hardware telemetry data.

7.1.2 Direct Link Protocol

- A frame format for use on the direct link was proposed. Because of the need for error handling a CRC field is incorporated in the frame for error detection. The synchronization word of choice was also included in the frame format to mark the start of every frame. A frame ID field differentiates between direct TLM frames and WOD TLM frames.

- The importance of the CRC implementation cannot be overemphasized because of the nature of the path traveled by the TLM data, in terms of distance and link errors. It has been shown in Figure 4.4 that the probability of error for the system is reduced tremendously when CRC error detection is used.

- The additions of the redundant information means increasing the 256 byte frame size in the current protocol by 2 bytes(sync word) plus 1 byte(frame ID) plus 2 bytes(CRC), which results in a 261 byte frame. This is equivalent to an efficiency reduction from 98% to 96%, which is a very small price to pay considering the data accuracy and reliability that comes along with it.

7.1.3 Telemetry Micro-controller Scheduling

- Real time scheduling concepts were investigated and the real-time nature of the telemetry system established as well as the scheduling requirements for the system. Then a real time scheduler with the ability of scheduling both synchronous and asynchronous processes was proposed.

- Because of the improved data reliability, the TLM micro-controller can now be used on a full time basis to relieve the OBC 's of the high interrupt load when they are collecting telemetry data. Functions like collection of WOD and sending data directly to ground can be implemented on the telemetry micro-controller rather than depending on the OBC 's, and as a result the OBC 's can be dedicated to other important functions.

- If the micro-controller is to be used for WOD collection, there may be a need for it to have access to bulk memory storage because the present 32 kilo-byte memory can only store about 120 WOD frames, if this storage is not enough.

7.2 Recommendations For Further Study

Having studied the telemetry system and using the knowledge and experience gained through the direct involvement with the system it is possible to make the following recommendations for further study:

- The presence of errors of both a block and random nature has been revealed in section 2.2. The CRC error detection scheme used in this thesis is not enough. A study of different error coding schemes with error correction capabilities should be undertaken to determine a suitable scheme for the telemetry channel coding while taking into consideration the processor limitations, bandwidth and coding overhead as well as coding delay. The CCSDS standard for telemetry channel coding suggests the use of Reed Solomon codes, convolutional codes and Turbo codes for telemetry data which can be investigated for suitability.

- One of the important requirements for space applications is flexibility. The design of the scheduler in this thesis is very inflexible. If there are new functions to be added to the telemetry software then the whole software has to be written all over again which is a waste of valuable time and resources. A flexible scheduler with the ability of adding new tasks for scheduling without changing the whole software should be investigated.

APPENDIX A:

A. GROUND SYNCHRONIZATION SOURCE CODE

```
/*-------------------------------------------------------------------------------------------------------------------------

 Name   : Synchronization.Java
 Date   : 27/09/2001
 Author : Boshielo B.T
 Desc   : This program receives data serially from a micro-controller that sends telemetry
          frames. The program performs the synchronization algorithm that searches for the sync word in
          the data bytes, and then looks for a second match before declaring a synchronization state
          (Lock) in which the algorithm stays until a mismatch occurs.

 -------------------------------------------------------------------------------------------------------------------------*/

import IO;
import java.io.*;
import java.net.*;


public class Synchronization{

public static void main(String[]args){


        IO.writeByte((short)0x3FB,(byte)0x80);     //Select baud registers(Baud rate=9600)
        IO.writeByte((short)0x3F8,(byte)0x0C);
        IO.writeByte((short)0x3FB,(byte)0x07);     //8 bits/char,1 stop,no parity
        IO.writeByte((short)0x378,(byte)0x00);

        new Receive().start();

 }
}

class Receive extends Thread {

  private byte RXBYTE;
  int search = 1;
  int lock = 3;
  int ptr1 = 0;
  int fptr = 0;
  int c = 0;
  int bcounter = 0;
  int fcounter1 = 0;
  int fcounter2 = 0;
```

```
int confirmation = 2;
boolean shift = false;
boolean confirm = false;
boolean seccheck = false;

byte[] buf1 = new byte[2];
byte[] fbuf = new byte[4];
int SSW = 0x1F35;                              // sync word
int RSW;
byte rxbyte;
int state = search;                            // initialize state to search
byte inbyte;
int W=4;                                       // initialize window size to 4

int value = 0;

Receive(){}

public void run(){

try{
        while (true){

                byte status = IO.readByte((short)0x3FD);
                 if(((status)&(0x01))==1){                    // Test if the byte has been received
                        inbyte=IO.readByte((short)0x3F8);    // read byte from receive register


//*****************************************************************************//
// LOCK STATE:
 /* In this state the frame counter is used to Lock the synchronization. Actually, the
   sync word should be used for the Lock and the frame counter be used as an
   additional checking measure. If a match is obtained, the state change to lock */

//*****************************************************************************//

  if (state==lock)
  {
        confirm=false;
        int size=0;
        int miss=2;
        bcounter++;

         if ((bcounter>=1)&&(bcounter<=4))
         {
             fbuf[fptr]=inbyte;
             fptr++;
             if (fptr==4)
                 {
                   fptr=0;
```

```
            String fcounter=Integer.toString((int)fbuf[0])+Integer.toString((int)fbuf[1])
            +Integer.toString((int)fbuf[2])+Integer.toString((int)fbuf[3]);
        }
}

if ((bcounter>=254-W)&&(bcounter<=254+W+1))
{

  if (shift)
  {
            buf1[0]=buf1[1];
            buf1[1]=inbyte;
            ptr1 = 2;
            c=c+1;
            if (c==W*2)
            {
                    state=search;
                    shift = false;
                    bcounter=0;
                    RSW=0;
                    c=0;
             }
  }
  else
  {
             buf1[ptr1]=inbyte;
             ptr1++;

  }
    if (ptr1==2)
    {
                shift = true;
                ptr1=0;
                RSW = (buf1[0]*256) + buf1[1];
                String xor = Integer.toBinaryString((RSW ^ SSW));
                size = xor.length();
                miss = 0;

                (int i=0;i<size;i++)
                {
                        int k = xor.charAt(i);
                        if (k==49)miss++;
                 }
      }
          if ((miss<=1)&&(miss>=0))
          {
                shift = false;
                state = lock;
                bcounter=0;
                RSW=0;
```

```
                                        c=0;
                                        fptr=0;
                        }
                }
        }

//*************************CONFIRMATION STATE*******************************//

 /* In this state the frame counter is used to confirm the synchronization. Actually, the
    sync word should be used for the confirmation and the frame counter be used as an
          additional checking measure. If a match is obtained, the state change to lock */

//****************************************************************************//
 if (state==confirmation)
 {
          confirm=true;
          int size=0;
          int miss=2;
          bcounter++;

   if ((bcounter>=1)&&(bcounter<=4))
          {
                  fbuf[fptr]=inbyte;
                  fptr++;
                  if (fptr==4)
                  {
                          fptr=0;
                          fcounter1=fbuf[0]+fbuf[1]+fbuf[2]+fbuf[3];
                          String fcounter=Integer.toString((int)fbuf[0])+Integer.toString((int)fbuf[1])
                          +Integer.toString((int)fbuf[2])+Integer.toString((int)fbuf[3]);
                  }
          }

          if ((bcounter>=254-W)&&(bcounter<=254+W+1+4))
          {

                  if (c==W*2)
                    {
                          state=search;
                          shift = false;
                          RSW=0;
                          bcounter=0;
                          c=0;
                    }
             if (shift)
             {
                    buf1[0]=buf1[1];
                    buf1[1]=inbyte;
                    ptr1 = 2;
                    c=c+1;
```

```
}
else
{
        buf1[ptr1]=inbyte;
        ptr1++;
}

  if (ptr1==2)
  {
                shift = true;
                ptr1=0;
                RSW = (buf1[0]*256) + buf1[1];
                String xor = Integer.toBinaryString((RSW ^ SSW));
                size = xor.length();
                miss = 0;

                for (int i=0;i<size;i++)
                {
                        int k = xor.charAt(i);
                        if (k==49)miss++;
                }
  }

        if (seccheck)
                {
                        fbuf[fptr]=inbyte;
                        fptr++;
                        if (fptr==4)
                        {
                                fptr=0;
                                fcounter2=fbuf[0]+fbuf[1]+fbuf[2]+fbuf[3];
                                if (fcounter2==fcounter1+1)
                                {
                                        c=0;
                                        RSW=0;
                                        bcounter=3;
                                        state = lock;
                                        seccheck=false;
                                }
                                else
                                {
                                        c=0;
                                        RSW=0;
                                        bcounter=0;
                                        state = search;
                                        seccheck=false;
                                }
                        }
                }
```

```
                    if ((miss<=0)&&(miss>=0))
                    {
                                shift = false;
                                c=0;
                                seccheck=true;

                    }
          }

 }

//*************************************SEARCH STATE*****************************//

 /* two successive bytes are received and compared with the two bytes of the sync word.
    A k number of errors is allowed. If a match is obtained the state changes to confirm */

//*******************************************************************************//
 if (state==search)
 {
          bcounter++;

          if (confirm)
          {
            confirm=false;
          }

          if (shift)
          {
                    buf1[0]=buf1[1];
                    buf1[1]=inbyte;
                    ptr1 = 2;
          }
          else
          {
                    buf1[ptr1]=inbyte;
                    ptr1++;

          }

             if (ptr1==2)
             {
                          RSW = (buf1[0]*256) + buf1[1];
                          shift = true;
                          String xor = Integer.toBinaryString((RSW ^ SSW));
                          int size = xor.length();
                          int miss = 0;

                          for (int i=0;i<size;i++)
                          {
                                    int k = xor.charAt(i);
```

```
                                if (k==49)miss++;
                        }

                        if ((miss<=1)&&(miss>=0))
                        {
                                shift=false;
                                ptr1=0;
                                 state=confirmation;
                                RSW=0;
                                bcounter=0;
                        }
                }
        }
      }
    }
  }
}
catch(Exception e){
        e.printStackTrace();

                }
        }
}

//*************************************THE  END*************************************//
//*************************************THE END*************************************//
```

APPENDIX B:

B. SOFTWARE REQUIREMENT SPECIFICATION

SRS.1 Introduction

This section defines a requirement specification for the improved telemetry system software. The aim is not to come up with a completely new specification but to incorporate new features in the existing telemetry micro-controller software. These new features have been outlined in chapters 3,4 and 5 of the thesis.

SRS.1.1 Purpose

The telemetry software carries out the collection, decoding, packaging, storing and transmission of telemetry data to both the OBC's and the ground station. The purpose of the additions and changes made to this software is to:

- Ensure a reliable communication channel for data transportation.
- Increase the validity of the telemetry data when it reaches its destination.
- Improve the telemetry micro-controller task scheduling to effectively include the additional functionality to cater for the new direct link protocol as well as the new tasks proposed for the system.

SRS.1.2 Scope

The telemetry system consists of two segments: The **space segment** that collects telemetry data, packages it into frames and sends it to the radios for transmission, and the **ground segment** that receives the telemetry data (see figure 1.2). The space segment consists of the OBC telemetry application and the telemetry micro-controller application. This specification is limited to the description of the telemetry micro-

controller functions and the task scheduler. The software runs on the 80C31 micro-controller found on the space telemetry system. The ground segment software does not form part of this specification.

SRS.2 Overall Description

SRS.2.1 Operating modes

The main functions of the system, assumptions, constraints, and dependencies of the telemetry software associated with this system are described in this section of the SRS.

Details of the operation of the current system are given in chapter 2. Refer to Figure 1.2 and Figure 2.1 for the next system operation description.
The telemetry micro-controller (i80C31) operates in the following modes:

- Minimum telemetry mode
  This data path is not indicated in Figure 1.2 as it does not form part of the work in this thesis.

- Full telemetry mode
  This mode is implemented in this software. It is a software telemetry mode with the onboard processors, i.e the i80C31 and the OBC's communicating to carry out the telemetry functions.

  This mode is divided into two sub modes:

  i. Telemetry data is collected by the i80C31 and then send to the OBC's for transmission through the AX.25 protocol.
  ii. Telemetry data that is collected by the i80C31 and transmitted using the direct link protocol proposed in chapter 4.

SRS.2.2 Constraints

Any improvements on the system are achieved through the use of software only without changing the hardware.

SRS.2.4 Assumptions and dependencies

Because the system is reactive, meaning that each task is carried out on requests from the OBC, the functioning of the system is dependent on the commands received from the OBC's. If the request frames are in error then the software will not respond to the requests. Furthermore it is assumed that the data collection hardware functions as expected. The speed of collection of a frame for instance is dependent on the conversion speed of the A/D converters.

SRS.3 Specific Requirements

SRS.3.1 Interface Requirements

SRS.3.1.1 Hardware Interface

The hardware interfaces are discussed in section 2.1.2, but are mentioned here for convenience.

- The Combiner modules
- The Sunsat Serial Bus.
- The OBC's.
- The modems.

SRS.3.1.2 Software Interface

The software interface requirement for the TLM software is a software module running on the OBC to send commands to the 80C31 module.

SRS.3.1.3 Communication Interfaces

- The SUNSAT serial bus provides a communication interface to the OBCs. This communication takes place in the form of serial interrupts.
- The data acquisition modules communicate with the 80C31 through the end of conversion signals from the ADC's which signal that data is ready for sampling. This communication takes place in the form of external interrupts from the ADC's.

SRS.3.2 Functional Requirements

SRS 3.2.1 Information flow

Table 5.1 contains all the telemetry functions to be performed by the telemetry software. Figure B.1 indicates some of the software functions as well as the flow of data information and request information between the functions. The OBC sends requests and receives data packets from the telemetry controller. Functions that require telemetry data outputs channel addresses to the data acquisition subsystems and then gets the data, packs it and presents it to the OBC or the modems.

The software functions represented in Figure B.1 are described in section 3.2.2.

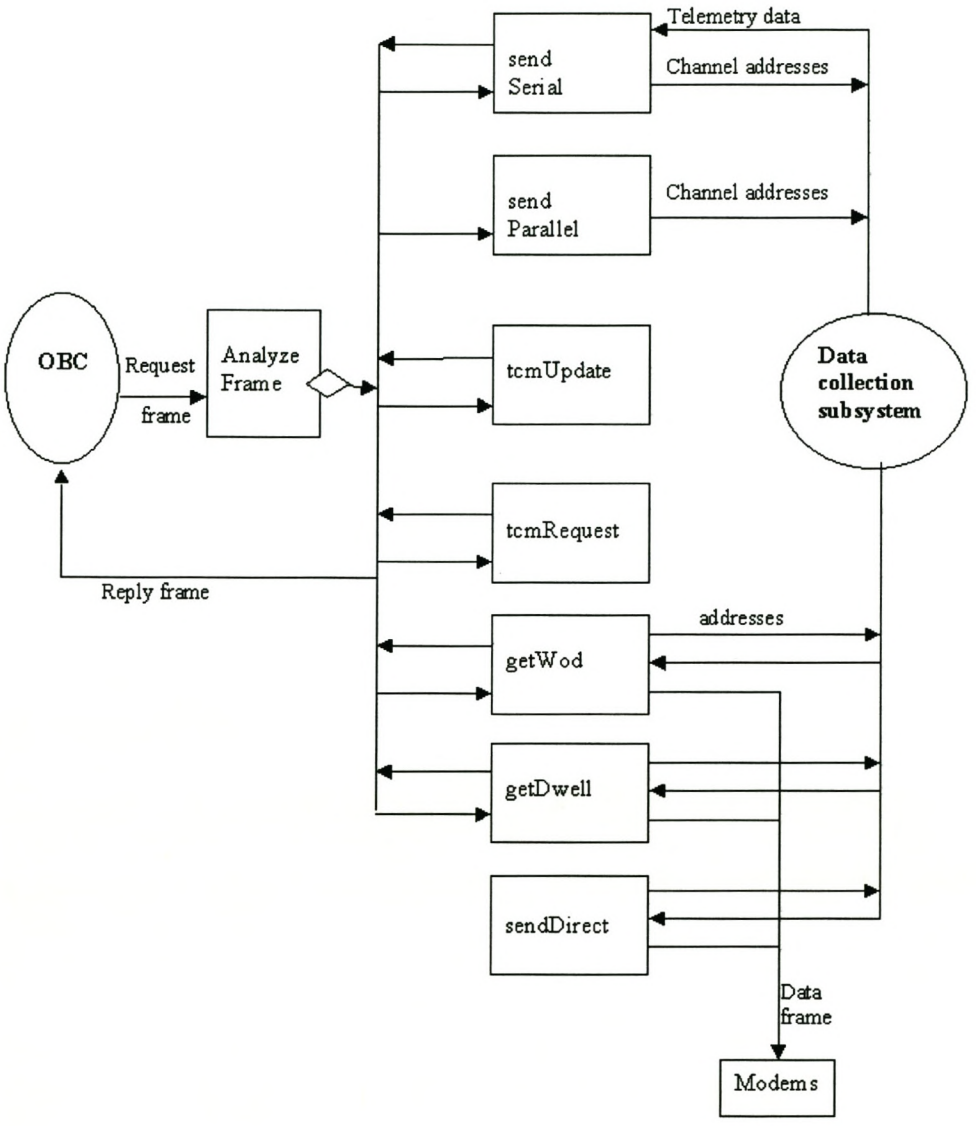

Figure B. 1*: Software functions and Data-flow diagram*

SRS 3.2.2 Process description

Not all the telemetry functions are represented in Figure B.1. See Appendix E for a description of the telemetry functions on the current telemetry system. An overview of all the functions (old and new) is given in Table 5.1. The following process description is based on those functions that reflect the additions and changes to the telemetry software, and only these functions are implemented in the software.

Process: Analyze received frame (processByte).
Input : Request frame from OBC.
Action : Frame validation by checking for the correct destination, source, length, command and LRC.
Output: Signal to call the function that is requested by the command parameter.

Process: Stop execution of long task (idlemode).
Input : Command to stop a specified task.
Action: Stop a specified task by setting or resetting the necessary flags.
Output: A specified task stopped.

Process: Send data using the modems (sendDirect).
Input : Command to send direct data, no. of channels to sample, channel addresses.
Action : Write out addresses to the target subsystem and read the data when it is ready. Calculate CRC as the bytes are being sent one at a time, and send the final CRC value at the end of every frame.
Output : A series of telemetry frames of the structure of Figure 9, send through a modem.

Process: Collect whole orbit data (getWod).
Input :Command to collect WOD, no. of channels to sample, channel addresses.

Action :Write out addresses to the target subsystem and read the data when it is ready. Store the collected data together with the frame counter for every frame, in external memory.

Output: Telemetry frames stored in external RAM.

Process: Send data using the serial port (wodRequest).

Input :Command to send WOD data from external memory to the .

Action : Read WOD data from external RAM, and send via the 1200 baud modems. Calculate CRC as the bytes are being sent one at a time, and send the final CRC value at the end of every frame.

Output : A series of telemetry frames of the structure of figure 9, send through modems.

SRS.3.2.3 Process Flow Requirements

The execution of a task depends on the priority of that task. The decision to use a priority scheme for task execution is outlined in section 5.4.1. The tasks are allocated priorities based on their urgency and importance. Tasks on the same priority level in the queue are executed in the order they were placed into the queue. Interrupts are not scheduler processes since they are called into action by external asynchronous events and totally bypass the scheduler. Figure B.2 shows a grouping of the telemetry tasks in priorities as they were characterized in Table 5.1.

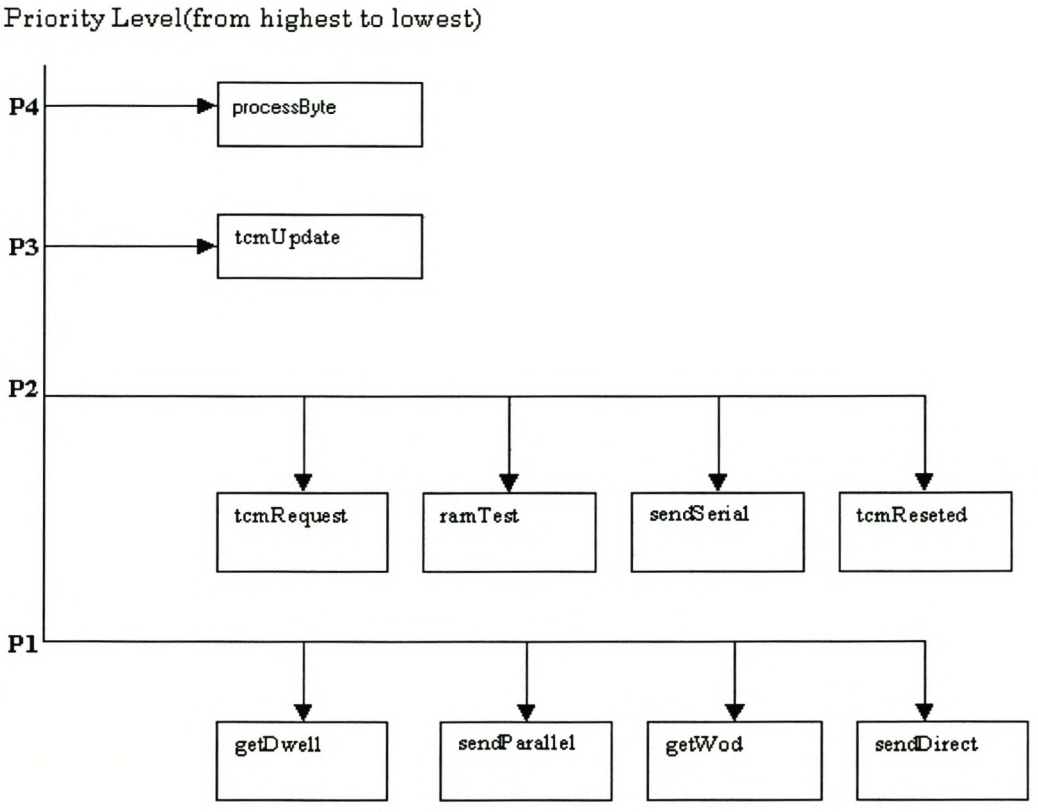

Figure B. 2*: Process-flow diagram*

The priority levels were decided upon in section 5.4.1. Refer to Table 5.1 for the allocation of priorities for all the tasks. The function processByte is indicated as the highest priority task because it is executed inside the ISR that bypasses every other function.

SRS.3.2.4 Data Elements Requirements

- The SSB to OBC interface

The bus holds serial data 9600baud, with one stop bit, one start bit and no parity. It is a half-duplex asynchronous serial bus and makes use of a master slave protocol where the OBC is always a master. See the details of the SSB protocol in appendix B. A SLIP character, <END>, is used at both ends of a data frame. The frame format for this protocol is illustrated in Figure B.3.

The OBC communicates its commands via the SSB. The data field of the frame, D0 to Dn contains the command to the 80C31. The response frame from the micro-controller to the OBC's uses the same frame format where the data field holds the requested information

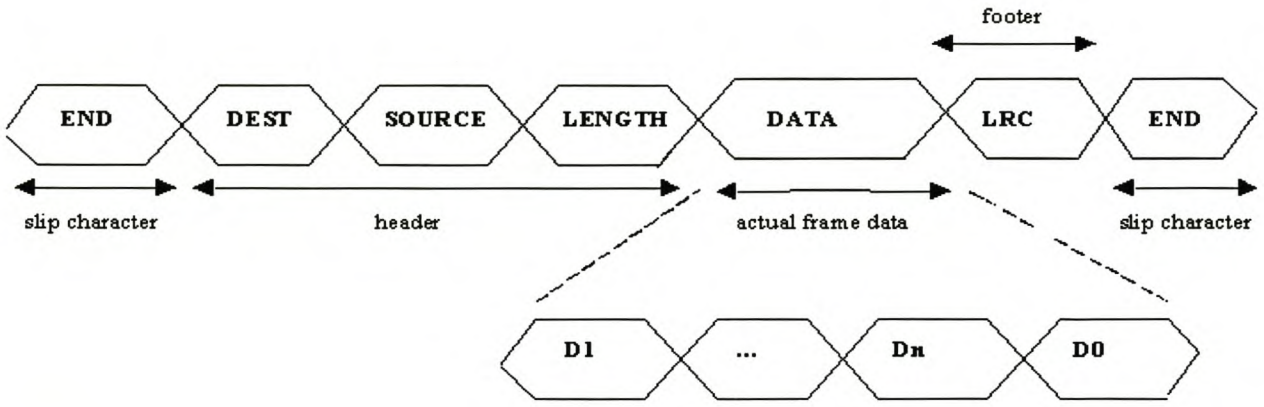

Figure B. 3: *The SSB frame format*

- Direct link data format

The communication protocol for this link is discussed in chapter 4. The contents of the frame include a synchronization word, a frame counter, a frame ID, data and a CRC. The frame structure is shown in Figure 4.3.

SRS.3.3 Performance Requirements

The requirements of the system in terms of performance are stated as follows:

- The controller software must have the ability to execute every requested task whenever it is required.

- Communication between the OBC and the 80C31 controller requires that the micro-controller have the ability to implement the SSB protocol.

- For tasks which are scheduled after a given time interval a timer is required to keep track of the scheduling period.

- The controller must be able to collect telemetry data from the data acquisition subsystems.

- The scheduler must be able to schedule a request even if there is a recursive task in progress.

APPENDIX C:

C. SOFTWARE DETAILED DESIGN

The design is divided into the scheduler design and the telemetry tasks design which described as follows:

C.1 The scheduler

1.1 Processing of requests

OBC requests are received serially from the SSB. The function *processByte ( )* validates the request inside the interrupt service routine. The message validation is implemented as a state machine as illustrated in figure C.1. This ISR implementation strategy is outlined in [N.S, 1999, p.56].

As prescribed by the SSB protocol frame format (Figure B.3), the validation process looks for the correct source, destination, message length, LRC, and ETX in the form of an <END> character. The state machine is initialized to the **rcvDestination** state. A valid destination byte (0x01) changes the state to **rcvSource** state. The valid source address is 0x1F, which changes the state to **rcvLength**. The length byte is any integer value greater 3, which is then stored in a variable *length*. The request information is contained in the bytes D0 to Dn of the data field of the frame and is retrieved in the ***rcvCommand*** state. The received request is stored in a public variable *request.* The state machine stays in the rcvCommand state until all the request bytes are received by comparing the number of received bytes against the message length that is received within the request frame. The request information that represents the channel addresses for to be sampled is stored in a fixed size buffer, *FrameBuf* of length 256. The LRC for the request frame is calculated while the bytes are coming in and is compared to the received LRC in **rcvLRC** state. A second <END> character marks the end of the request

frame. When it is received, in **rcvETX** state, the function *addToReady ( )* is called to immediately add the received request to a *ready list* and the state machine returns to **rcvDestination** to wait for further requests.

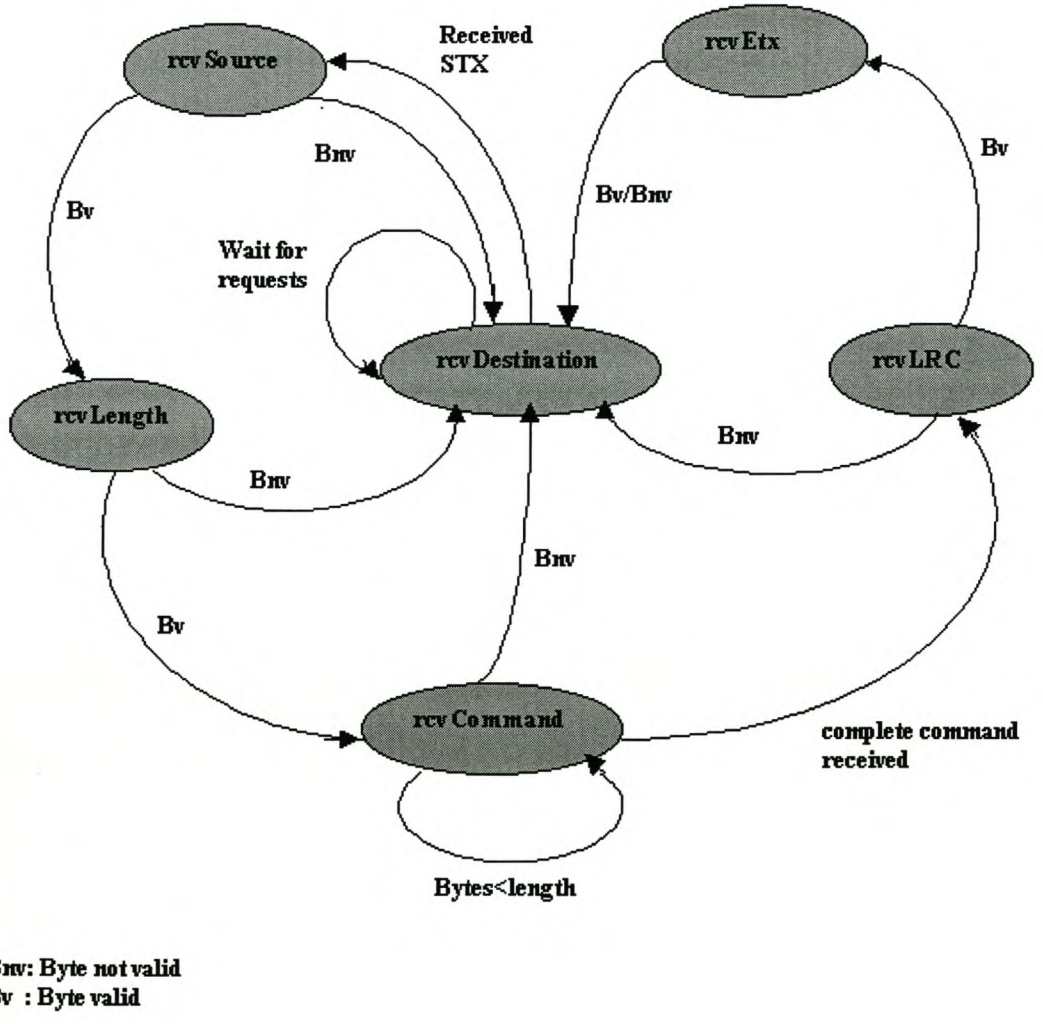

Figure C. 1: State machine for message validation

If a received byte in any state is not the byte expected in that state, the state machine returns to the **rcvDestination** state to wait for new requests.

1.2 Adding tasks to the ready list

The ready list is organized as an array of structures, where each element consists of a task number and its priority. The following code segment illustrates the ready list's structure together with the priorities-array and the allocated task priorities.

```
struct task {
 unsigned char task_tag;
 unsigned char priority;
};

struct task readyList [MAX_TASKS];
unsigned char idata priorities[ ] = {0,2,2,1,1,2,1,1,3,2,2,1,1,1,2,2,1};
```

A valid request is any integer value between 0 and 16, where the value represents a task tag which is in turn mapped to its corresponding priority value from a priority list, to determine the task's priority. For example if the received request value is 8, the priority for task number 8 is the element at index 8 of priorities [ ], which is 3. A valid request is immediately added to a ready list in order of increasing priority, using the function *addToReady ( ).* A task is added to ready list only if that task is not already in the ready list. Every-time a new task is added to the ready list, a flag, *disp* is set to signal to the dispatcher function to schedule the task.

1.3 Task scheduling and dispatching

The need for a task dispatcher and task scheduler was identified in section 5.4.2. To meet this requirement two functions dispatcher( ) and scheduler(uchar t) are defined and they work as follows:

If called, the *dispatcher( )* calls the next task to run provided there is no running task. The task tag of the zeroth element in the list is stored in a varable t, and then deleted from the list, and the list is sorted. Then the task corresponding to the value stored in t is executed. The flow diagam for the dispatcher function is illustrated in Figure C.2.

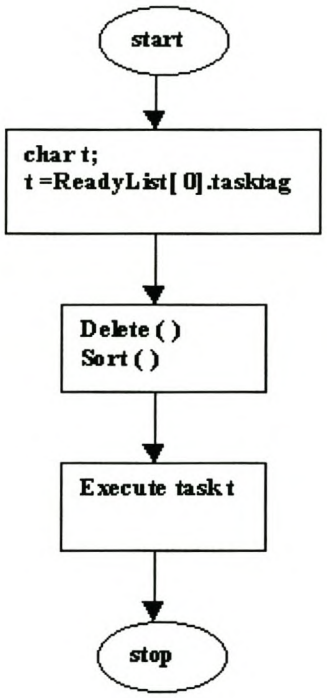

Figure C. 2: *The dispatcher function*

The flow diagram for the scheduler function is illustrated in figure C.3. Four flags, *direct*, *wod, parallel* and *wodreq* are defined. These flags indicate that a reccurent lowest priority task, i.e direct transmission, WOD collection, parallel data collection and WOD transmission respectively, is executing. If any of the mentioned flags are set, the *scheduler ( )* monitors the expiry of a time period so that the task is rescheduled by adding it to the ready list.

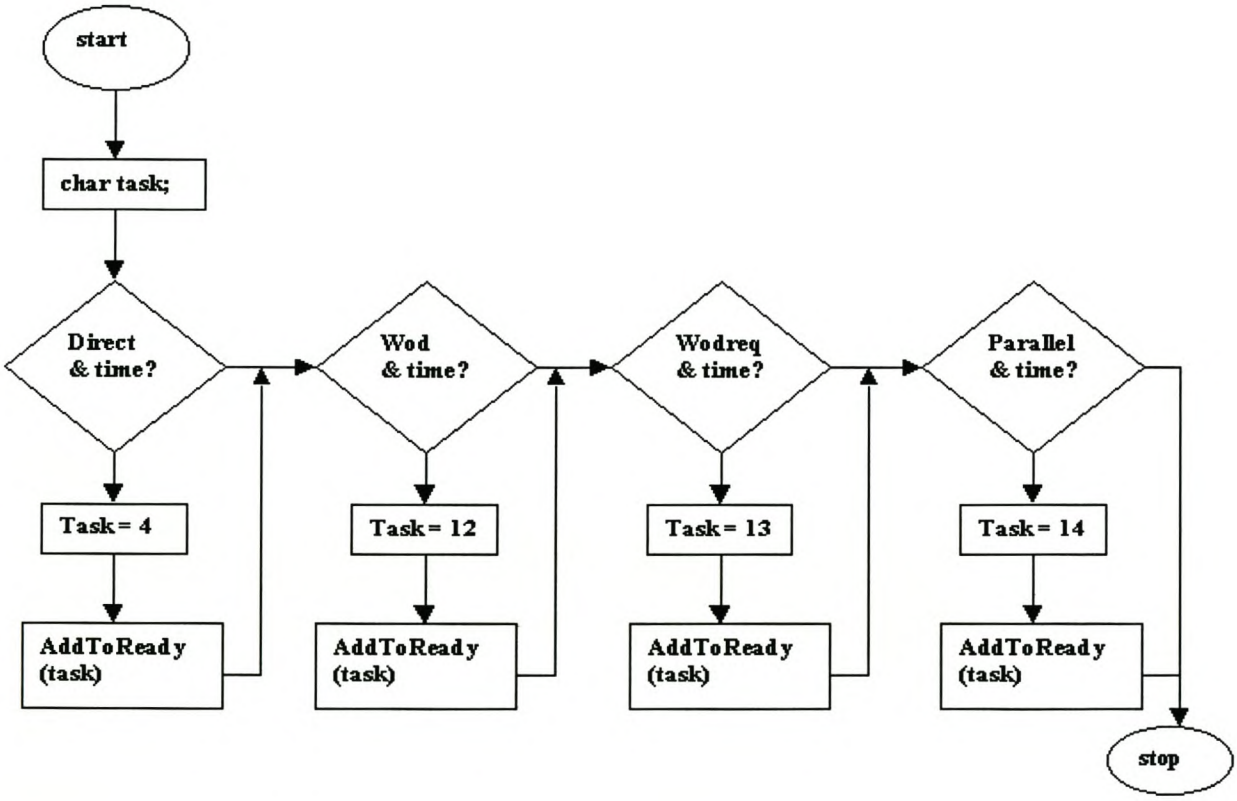

Figure C. 3: *The scheduler function*

1.4 The ISR and the MAIN routine

The telemetry scheduler consists of an interrupt service routine and a main routine. All the main routine does, is to initialize the controller and then sits in an endless loop that calls the function *idle ( ).* The ISR and *idle ( )* are shown in figure C.4:

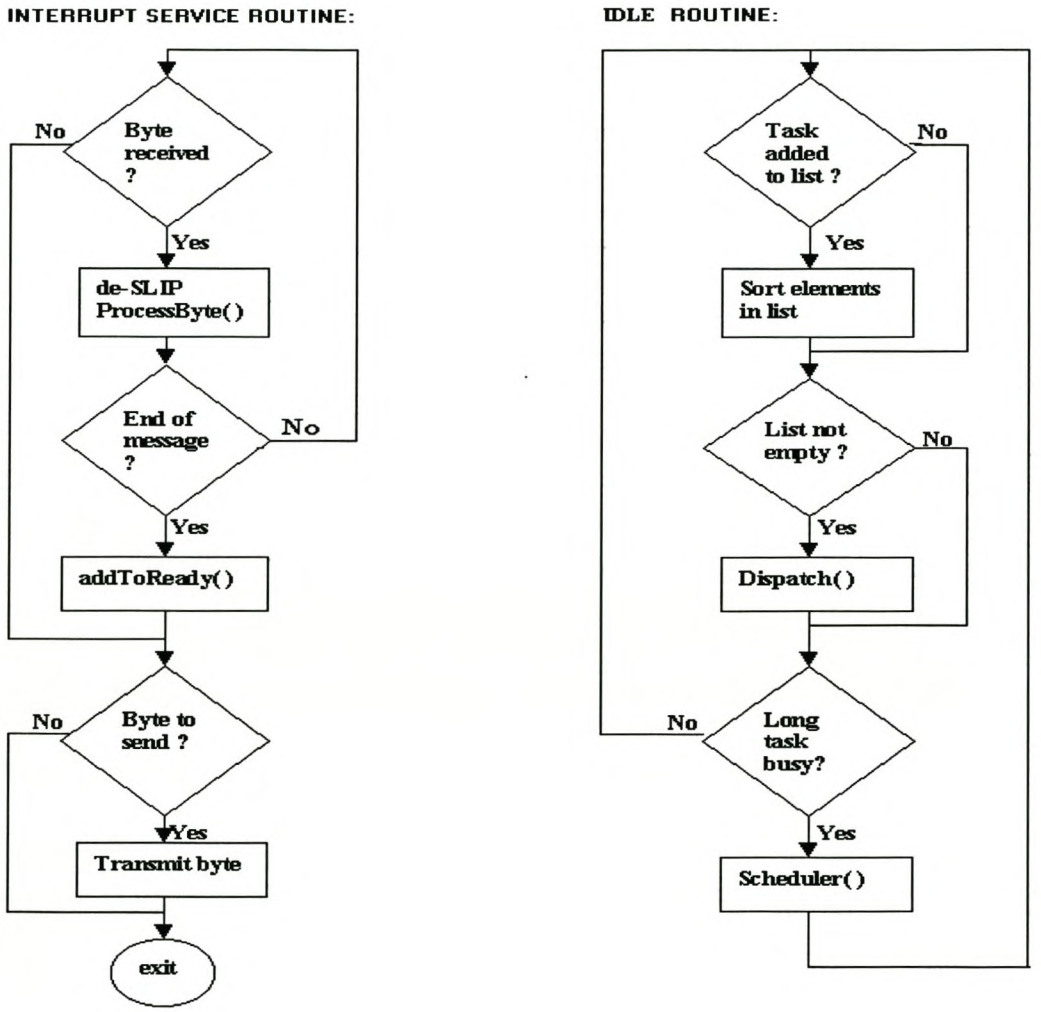

Figure C. 4: *Process flow diagram for the 80C31 scheduler*

Idle ( ) waits for a new task in the ready list and calls *sort ( )* to arrange the list elements. If there are ready tasks in the list, *dispatch ( )* is called. If any of the recurrent tasks are busy, *scheduler (int t)* is called to reschedule such tasks to resume their execution after a specified time interval. The lowest priority tasks will continue to be scheduled until a command suggests otherwise. The complete procedure of receiving requests, processing and adding the request to the ready list, is completed within the ISR.

1.5 Management of the ready list

The elements of the ready list are arranged in order of decreasing priority so that the element at index 0 is the highest priority task or if the priorities of the tasks in the list are the same, then the earliest task scheduled is the element at index 0. This arrangement is achieved by adding the tasks to the list in increasing index order and then sorting the list elements using a sorting algorithm.

Because of the list arrangement, the next task to run at any point is the first element in the list. Every time a task is executed, its task value is replaced by '0' in the list. The list is resorted so that the 0 appears as the last element of the ready list. To verify the status of the ready list, the function *isEmpty ( )* shown below is used.

```
int isEmpty (void)
{
  if(List[0].task_tag==0)return 1;
  else return 0;
}
```

If the element at index 0 is null, then there are no ready tasks in the ready list. The function *isEmpty ( )* returns a 1 if the ready list is empty and a null otherwise.

C.2 The telemetry functions

At the beginning of each function a global flag *busy* is set and it is cleared when the function completes. This flag serves as an indication to the dispatcher that the processor is occupied so that task dispatching is done only when the processor is free.

All the functions that send data on the SSB use a global flag *sending* when transmitting data. This flag is used by the ISR to input incoming data so that it does not collide with outgoing data because the SSB is half-duplex.

2.1 Sending direct data to the modems

The function sends destination, source and length. Then the function stays in a loop and read channel addresses from external storage and output them to the data acquisition system of choice. After outputting each address an external interrupt is enabled. On receiving an interrupt from the A/D converters the converted data is read. The important part of this function is the drivers that allow communication with the data acquisition systems as well as the SSB. Three functions, *sendByte(char c), getChannel* and *external_int0* communicate directly with the hardware and hide away all the driver details.

2.2 Collecting WOD

This function output channel addresses to the data collection system and reads the data on receiving an end of conversion signal. The data is stored in external memory in buffer *WodBuf [32000].* A pointer is incremented every time a byte is stored to monitor the state of the buffer. When the buffer is full the collection of WOD frames is stopped. WOD frames are collected with a given time period between the frames. It is the duty of the

scheduling task to ensure this spacing of the collected frames. A 1-second spacing is used for demonstration. To keep track of the time of the WOD data, each frame is stored together with its counter number to provide the time information.

2.3 Sending WOD to the modems

WOD frames are read from the buffer *WodBuf* and send to the modems one frame at a time. The frames are send with a given spacing between them. A one second period has been used in this software. The same pointer that is used in the collection of the WOD frames is used to determine if there are still frames in the *WodBuf.* The function *txByte(char c)* is used to directly drive the hardware that allows communication with the 1200-baud modems through which the data is send.
At the beginning of each frame the Barker sync word is sent. The function *getCrc(char c)* is used to calculate the CRC of the bytes as they are being sent and the accumulated CRC value is send at the end of every frame.

Finally the design for the telemetry scheduler as well as some of the telemetry functions that it will manage has been proposed.

APPENDIX D:

D. TELEMETRY SYSTEM SOURCE CODE

```
/*****************************************************************************************/
/* Target device: Intel 80C31 */
/* This program receives requests via the SUNSAT serial bus and validates the
   received bytes in the function processByte(c). A valid request is added to
   a ready list and then dispatched when the main routine calls the dispatcher
   Those tasks which are recursive are rescheduled by the scheduler after a
   given time interval. The microcontroller responds to commands by sending
   the required data to the correct destination, i.e either the OBC or modems. */
/*****************************************************************************************/
#pragma SMALL              // Select small memory model
#include "reg_8031.h"      // special function registers 89S8252
#include "int_8031.h"      // Interrupt definition file

#define FOREVER for (;;)   // Endless loop
#define FALSE 0
#define TRUE  1
#define MAX_TASKS 10       // Maximum no. of tasks schedulable
#define STX 0XC0           // Start of Frame = END
#define Slip1 192          // END
#define Slip2 219          // ESC
#define Slip3 220          // Transpose END
#define Slip4 221          // Transpose ESC
#define syncword1 0x1f     // First byte of sync word
#define syncword2 0x35     // Second byte of sync word
#define wodID 0x01         // WOD frames identity
#define directID 0x02      // Direct frames identity
#define uchar unsigned char
```

```
/*****************************************************************************************/
                        /* Global declaration of variables */
/*****************************************************************************************/
 unsigned char xdata FrameBuf[256];
 unsigned char xdata WodBuf[32000];

unsigned int idata crc_16_table[16] = {
  0x0000, 0xCC01, 0xD801, 0x1400, 0xF001, 0x3C00, 0x2800, 0xE401,
  0xA001, 0x6C00, 0x7800, 0xB401, 0x5000, 0x9C01, 0x8801, 0x4400 };

 unsigned char idata priorities[]={0,2,2,1,1,2,1,1,3,2,2,1,1,1,2,2,1};
 enum States{rcvDest,rcvSource,rcvLength,rcvCommand,rcvLrc,rcvETX};
 enum States state;
 state=rcvDest;

 struct task {
  unsigned char task_tag;
  unsigned char priority;
 };

 struct task List[MAX_TASKS];
 int idata wodin,pnt2;
 unsigned short idata crc;
 unsigned char idata t_pos,task,xor,dest,source,length,LRC,x,fin,value;
 unsigned char idata A2D_value,a2d,fcounter1,fcounter2,fcounter3,fcounter4;
 unsigned char idata ticks,min,second,tensecond,request,address;
 bit startwod,exor,escape,disp,txok,sec,msec;
 bit wodreq,busy,direct,wod,add,inlist;

/*****************************************************************************************/
                        /* This function sets up the serial port */
/*****************************************************************************************/
 void serial_init(void)
 {
      TMOD = 0x21;            // Timer1 in 8bit auto reload.
      TH1  = 0xFF;            // 9600 baud at 3.6MHz.
      TCON = 0x40;            // start timer1.
      PCON = 0x00;            // double baudrate bit not set.
```

```
    SCON = 0x58;            // Mode1 8-bit UART,Enable serial reception.
    ES   = TRUE;            // Enable serial interrupt
    EA   = TRUE;            // Enable all interrupts.
    EX0  = FALSE;           // Disable external interrupt0.

}

/*****************************************************************************************/
                        /* This function sets up the timer */
/*****************************************************************************************/
 void timer0_init(void)
{
    TH0 = 0xFE;             // 1's complement of 1ms/[(1/Fosc)*6]
    TL0 = 0xCC;             // =[TH0,TL0]. Fosc=3.686MHz=>[0xFE,0xCC]
    ET0 = TRUE;             // enable timer0.
    TR0 = TRUE;             // start timer0.
}
/*****************************************************************************************/
                        /* This function sets up the A2D convertor.*/
/*****************************************************************************************/
 void ADC_initialize (void)
{
  IT0 = TRUE;               // enable single conversion mode
  EX0 = FALSE;              // Enable A2D converter interrupt
}
/*****************************************************************************************/
                    /* This function sets up the micro-controller */
/*****************************************************************************************/
 void main_init(void)
{
  ADC_initialize();
  serial_init();
  timer0_init();
}
/*****************************************************************************************/
                        /* Timer Interupt Function */
/*****************************************************************************************/
void Timer_Int (void) interrupt TIMER0 {
```

```
  TR0=FALSE;                // Stop Timer0.
  TH0=0xFE;                 // 1's complement of 1ms/[(1/Fosc)*6]
  TL0=0xCC;                 // =[TH0,TL0]. Fosc=3.67MHz=>[0xFE,0xCC]
  TR0=TRUE;                 // Restart Timer0.

  ticks++;                  // Increment ticks counter
  if(ticks==100){           // 100 * 1ms = 100ms.
    ticks=0;
    msec=TRUE;
    tensecond++;            // increment tensecond counter
  }
  if(tensecond==10){        // 100 * 10 = 1second
    tensecond=0;
    sec=TRUE;
    second++;               // increment second counter
  }
    if(second==6){          // 10 * 6 = 1min
      second=0;
      min++;
    }
      if(min==100){
        min=0;              // reset min counter
      }
}
/*****************************************************************************************/
/* Calculate crc. This function receives a byte that is being send and calculates
   a running crc which is then passed to the calling function. The CRC-16
   polynomial is used .This source code fragment was obtained from [SG,1999] */
/*****************************************************************************************/
unsigned short int getCrc (unsigned char c) {
  int r;

    /* compute checksum of lower four bits of *p */
    r=crc_16_table[crc&0xF];
    crc=(crc>>4)&0x0FFF;
    crc=crc^r^crc_16_table[c&0xF];
```

```
   /* now compute checksum of upper four bits of *p */
   r=crc_16_table[crc&0xF];
   crc=(crc>>4)&0x0FFF;
   crc=crc^r^crc_16_table[(c>>4) & 0xF];

   return crc;
 }
/*****************************************************************************************/
                          /* Send a single byte via the serial bus */
/*****************************************************************************************/
 void sendByte(unsigned char c)
{
   value = P3;              // enable Int_B
   value&=0x0F7;            // enable Int_C
   value|=0x35;             // prevent contention
   P3   = value;
   TXD = TRUE;              // enable transmission
   REN=FALSE;               // disable reception
   SBUF=c;                  // transmit byte on SBUF
   while(!txok){            // wait for successful transmission
   }
   txok= FALSE;
   P3  |=0x38;              // open BUS P1
   REN =TRUE;               // enable serial reception
}
/*****************************************************************************************/
        /* Send a single byte via the serial port to the 1200baud modems*/
/*****************************************************************************************/
 void txByte(unsigned char c)
{
   TXD = TRUE;              // enable transmission
   getCrc(c);               // calculate crc
   SBUF=c;                  // transmit byte on SBUF
   while(!txok){            // wait for successful transmission
   }
   txok= FALSE;
}
/*****************************************************************************************/
```

```
                    /* Send a character with slip bytes */
/*****************************************************************************************/
 void sendSlip (uchar c){

    if(!exor)LRC=LRC^c;        // calculate LRC
    exor=FALSE;
    if(c==Slip1){              // character = 192?
      sendByte(Slip2);         // send 219
      sendByte(Slip3);         // send 220
    }
    if(c==Slip2){              // character = 219?
      sendByte(Slip2);         // send 219
      sendByte(Slip4);         // send 221
    }
    if((c!=Slip1)&&(c!=Slip2)){}
      sendByte(c);
}
/*****************************************************************************************/
                    /* Output new channel address */
/*****************************************************************************************/
 void getChannel(void){
    P3|=0x038;                 // select 80C31 to write on P1 bus
    P1= address;               // output address on P1 bus
    P3&=0x0D7;                 // address latch enable low
    P3|=0x028;                 // address latch enable high
    P1= 0x0FF;                 // configure P1 as input port
}
/*****************************************************************************************/
/* Calculate next frame_counter value. It is a simulation of the hadware generated counter*/
/*****************************************************************************************/
 void getFcounter(void)
{
     fcounter4=fcounter4+1;
     if(fcounter4==10){
          fcounter4=0;
          fcounter3=fcounter3+1;
     }
         if(fcounter3==10){
```

```
            fcounter3=0;
            fcounter2=fcounter2+1;
          }
          if(fcounter2==10){
             fcounter2=0;
              fcounter1=fcounter1+1;
          }
}

/*****************************************************************************************/
 /* Collect a single frame consisting of data from all channels and send via
    the SSB. */
/*****************************************************************************************/
 void sendSerial(void)
{
    busy=TRUE;
    sendByte(Slip1);
    sendByte(source);
    sendByte(dest);
    sendSlip(length);
    getFcounter();
    sendSlip(fcounter1);
    sendSlip(fcounter2);
    sendSlip(fcounter3);
    sendSlip(fcounter4);

    EX0=TRUE;
    while(!a2d){ }
    a2d=FALSE;
    EX0=FALSE;
    fin=0;
    address=0;
     while(fin<252){
       getChannel();          // output address
       EX0=TRUE;              // enable external interrupt0
       while(!a2d){           // wait for data ready
       }
```

```
      a2d=FALSE;
      EX0=FALSE;              // clear external interrupt0
      address=address+1;      // increment address counter
      sendByte(A2D_value);    // send converted byte
      fin++;
     }
   exor=TRUE;
   sendSlip(LRC);
   sendByte(Slip1);
   LRC=0;
   busy=FALSE;
}
/*****************************************************************************************/
/* Collect a single frame consisting of data from all channels and send via
  the direct link.The frame contais a 2-byte sync word, a 4-byte frame
  counter, direct frame ID, 252 data bytes as well as a CRC header */
/*****************************************************************************************/
 void sendDirect(void)
{
   busy=TRUE;
   txByte(syncword1);
   txByte(syncword2);
   txByte(directID);
   getFcounter();
   txByte(fcounter1);
   txByte(fcounter2);
   txByte(fcounter3);
   txByte(fcounter4);

   EX0=TRUE;
   while(!a2d){ }
   a2d=FALSE;
   EX0=FALSE;

    fin=0;
    address=0;
    while(fin<252){
      getChannel();
```

```
      EX0=TRUE;
      while(!a2d){
      }
      a2d=FALSE;
      EX0=FALSE;
      address++;
      txByte(A2D_value);
      fin++;
    }
     txByte(crc);
     crc=0;
     busy=FALSE;
}
/*****************************************************************************************/
/*Collect whole orbit data from all telemetry channels and store data in
 external ram in the buffer WodBuf[]. Every frame is stored with its frame
 counter value. If the buffer is full data collection is stopped. */
/*****************************************************************************************/
void getWOD(void)
{
   unsigned char in,x;
   x=0;
   busy=TRUE;
   if(startwod){
    startwod=FALSE;
   }
    txByte(request);
    getFcounter();
    WodBuf[wodin++]=fcounter1;
    WodBuf[wodin++]=fcounter2;
    WodBuf[wodin++]=fcounter3;
    WodBuf[wodin++]=fcounter4;

    in=0;
    address=0;
    while((in<252)&&(wodin<32000)){
     getChannel();
     EX0=TRUE;
```

```
      while(!a2d){ }
      a2d=FALSE;
      EX0=FALSE;
      address=address+1;
      WodBuf[wodin++]=A2D_value;
      in++;
     }
    if(wodin>=32000){wod=FALSE;}
    busy=FALSE;
}

/*****************************************************************************************/
          /*Collect WOD data from a external ram and send to the modems*/
/*****************************************************************************************/
 void wodRequest()
{
  int pnt;
  pnt=0;
  busy=TRUE;
  txByte(syncword1);
  txByte(syncword2);
  txByte(wodID);

  while((pnt<256)&&(pnt2<wodin)){
   txByte(WodBuf[pnt2]);
   pnt++;
   pnt2++;
  }
  if(pnt2>=wodin){
   wodin=0;
   pnt2=0;
   wodreq=FALSE;
  }
  crc=0;
  busy=FALSE;
}
/*****************************************************************************************/
```

```
                                /* Stop a running long task */
/*****************************************************************************************/

 void stopMode(void)
 {
   busy=TRUE;
   if(direct)direct=FALSE;
   if(wodreq)wodreq=FALSE;
   if(wod)wod=FALSE;
   busy=FALSE;
 }
/*****************************************************************************************/
/* The rest of the telemetry functions are can be added here. Only a few are
  shown with empty bodies */
/*****************************************************************************************/
 void tcmUpdate(void){}

 void ramTest(void){}

 void sendParallel(void){}

 void tcmRequest(void){}
/*************************************************************************/
                          /* Sort the ready list in order */
/*************************************************************************/
 int sort()
 {
   int count=t_pos;
   int a;int b;

   struct task temp;

   for(a=1;a<count;++a)
     for(b=count-1;b>=a;--b){
       if(List[b-1].priority<List[b].priority){
         temp=List[b-1];
              List[b-1]=List[b];
              List[b]=temp;
```

```
      }
    }
}
/*****************************************************************************************/
                /* Add a new task to the ready_list . */
/*****************************************************************************************/
 void addToReady(int req)
{
   unsigned char t;
   for(t=0;t<10;t++)
     if(List[t].task_tag==req)inlist=TRUE;
     if(!inlist){
       List[t_pos].task_tag=req;
       List[t_pos].priority=priorities[req];
       t_pos++;
       if(t_pos==10)t_pos=0;
     }
       inlist=FALSE;
       disp=TRUE;
}
/*****************************************************************************************/
                        /* Delete executed tasks from the list*/
/*****************************************************************************************/
 int delete ()
{
   List[0].task_tag=0;
   List[0].priority=0;
   sort();
   t_pos=t_pos-1;
   return;
}
/*****************************************************************************************/
                                /*Get list status*/
/*****************************************************************************************/
 int isEmpty (void)
{
   if(List[0].task_tag==0)return 1;
   else return 0;
```

```
}
/*****************************************************************************************/
                /* Get highest priority task to run */
/*****************************************************************************************/

int dispatch()
{
  uchar t;
  t=List[0].task_tag;
  delete();
  sort();

  if(t==1){sendParallel();}
  else if(t==2){tcmRequest();}
  else if(t==3){tcmUpdate();}
  else if(t==4){getWOD();}
  else if(t==5){sendSerial();}
  else if(t==6){ramTest();}
  else if(t==7){stopMode();}
  else if(t==8){wodRequest();}
  else if(t==9){sendDirect();}
}
/*****************************************************************************************/
                    /* Schedule a recurrent task when time expires */
/*****************************************************************************************/
void scheduler(uchar task)
{
   if((wodreq)&&(sec)){
     task=8;
     request=task;
     addToReady(task);
   }
   if((direct)&&(sec)){
     task=9;
     request=task;
     addToReady(task);
   }
   if((wod)&&(sec)){
```

```
      task=4;
      request=task;
      addToReady(task);
    }
    sec=FALSE;
}



/*****************************************************************************************/
                        /* Process Received Messages */
/*****************************************************************************************/
 void processByte(uchar ch){

  switch(state){

  case rcvDest:
    dest=ch;
    if(dest==1){state=rcvSource;}
    else {state=rcvDest;}
    break;

  case rcvSource:
    source=ch;
    if(source==0x1F){state=rcvLength;}
    else {state=rcvDest;}
    break;

  case rcvLength:
    length=ch;
    xor=xor^length;
    if(length>2){state=rcvCommand;x=0;}
    else {xor=0;state=rcvDest;}

    break;

  case rcvCommand:
```

```
    xor=xor^ch;
    if(length>=2){length=length-1;FrameBuf[x]=ch;x++;}
    else {
      request=FrameBuf[0];
      if((request>=1)&&(request<16))state=rcvLrc;
      else {xor=0;state=rcvDest;}
    }
    break;

  case rcvLrc:
    if(ch==xor){
      xor=0;
      state=rcvETX;
    }
    else {xor=0;state=rcvDest;}
    break;

  case rcvETX:
    if(ch==192){
      state=rcvDest;
      LRC=xor=0;
      if(request==4){
        wod=TRUE;
        wodin=0;
        startwod=TRUE;
      }
      if(request==8){
        wodreq=TRUE;
      }
      if(request==9){
        direct=TRUE;
      }
      addToReady(request);
    }
    else {LRC=xor=0;state=rcvDest;}
    break;
  }
}
```

```
/*****************************************************************************************/
/* The function is called by the main routine to call the dispatcher and
 scheduler  whenever it is required */
/*****************************************************************************************/
 void idle(void)
{
    uchar ch;
    if(disp){
      disp=FALSE;
      sort();
    }
    if((isEmpty()==0)&&(!busy)){
       dispatch();
    }
    else if((wod)||(direct)||(wodreq)){
      if(sec){ch=0;scheduler(ch);}
    }
}
/*****************************************************************************************/
                              /* External Interrupt 0 Function */
/*****************************************************************************************/
 void External_Int0 (void) interrupt EXTI0{
   P3&=0x0C7;              // TLMDATA low
   A2D_value=P1;           // read raw data from port
   P3|=0x038;              // TLMDATA high
   a2d=TRUE;
}
/*****************************************************************************************/
                              /* Serial Interrupt Function */
/*****************************************************************************************/
 void Serial_Int (void) interrupt SINT{
   unsigned char rec;
   if(RI){
     rec=SBUF;
     if(rec==Slip2)escape=TRUE;
     else if(escape){
       escape=FALSE;
       if(rec==Slip3)rec=Slip1;
```

```
      if(rec==Slip4)rec=Slip2;
     }
      if(!escape){processByte(rec);}
      RI=FALSE;
   }
    else if (TI){
      TI=FALSE;
      txok=TRUE;
    }
}
/*****************************************************************************************/
                                    /* Main program */
/*****************************************************************************************/
void main (void){
 main_init();

 FOREVER{
   idle();
 }
}
/*****************************************************************************************/
                                    /* END */
/*****************************************************************************************/
```

APPENDIX E:

E. ADDITIONAL DETAILS ON THE SUNSAT TELEMETRY SYSTEM OPERATION

E.1 Introduction

This appendix provides additional information on the SUNSAT telemetry system. The SSB protocol and the current requirements of the telemetry system as well as description of the different functions implemented by the system are described. This information is obtained from the SUNSAT documentation on telemetry, "SUNSAT telemetry system", version 2, 2001.

E.2 The SUNSAT Serial Bus protocol

The SSB transmits serial data at 9600 Baud with one start bit, one stop bit and no parity bits using a SLIP protocol, in other words an <END> character is added to the start and end of each frame. Extra SLIP characters may be added to the rest of the frame by the sender when necessary. These extra SLIP characters then are removed by the receiver.

The SSB frame format is illustrated in Figure B.3. A frame sent to the TMS tray will be interpreted as follows: The **first byte** in the frame content is either the number of telemetry channels requested or the telecommand subsystem number of which the status must be updated. The **second byte** is a command byte. It must have a value between 1 and 10 to be valid (the meaning of this value will be explained later). In the case of a telemetry request the maximum number of channels that can be requested is 250: there can only be 256 bytes in one SSB frame. Six bytes are used for the header

(3), footer (1), number of channels (1) and telecommand (1). XORing all the bytes in the frame body, in other words calculates the footer:

Checksum = D0 XOR D1 XOR D2 ----- XOR DN.

The checksum is calculated by the sender (before any SLIP characters have been added) and are transmitted as the footer of the frame. The receiver (the 8031 processor on the TMS tray) calculates the checksum in the same manner as the transmitter and then compares it with the received checksum.

SLIP characters are added to a frame by the sender when the reserved bytes 192 (<END>) or 219 (<ESC>) are present in the frame body. The particular reserved byte is then replaced by:

- <192> is replaced with the sequence <219 ><220>
- <219> is replaced with the sequence <219>< 221>

The receiver always checks for a <219> (<ESC>) byte and when observed with either a <220> or <221> byte immediately following, replaces the sequence with the correct byte.

E.3 The 80C31 software

Depending on the value of the command byte, the 8031 will enter a specific mode. As stated before, the TMS system is dual redundant (System B and System C are identical). When the 8031 is collecting telemetry data, it can use either System's output data. A decoder determines whether the 8031 will use the interrupt (signalling the end of an A/D conversion) from System B or C. The meaning of the command byte (and modes) are as follows:

1. Send one telemetry frame (obtained from System B) on SSB.
2. Send one telemetry frame (obtained from System C) on SSB.
3. Continuously supply addresses to System B, 9600 bps (OBC reads data with parallel port interface.

4. Continuously supply addresses to System C, 9600 bps (OBC reads data with parallel port interface.
5. Stop supplying addresses (only valid if 8031 is busy with 3. or 4.).
6. Continuously supply addresses to System B, 1200 bps (OBC reads data with parallel port interface.
7. Continuously supply addresses to System C, 1200 bps (OBC reads data with parallel port interface.
8. Update the status of a tele-command latch.
9. Supply the status of all tele-command latches (subsystems 1 and 2).
10. Indicate to OBC if 8031 has been reset after the initial power-on reset.

Tele-command switches select which subsystem's data (B or C) will be applied to port 1 of the 8031. For instance, when an OBC commands the 8031 (via the SSB) to supply addresses to System B, it must ensure that the tele-command switches that control the relevant buffers on the TMS board, apply the correct system's data on the 8031 port. The same goes for System C.

An example of the command decoding process and reply for mode 1 and mode 2 is as follows:

Mode 1

An OBC requests up to 250 telemetry channels: D0 is the number of channels, D1 is equal to 1 to indicate this mode and D2 to Dn contains the requested channel numbers. The 8031 in answer sends the header of the next frame to the OBC and starts to collect the data for the requested channels (by putting out the address, waiting for the end of conversion (EOC) interrupt and reading the sampled data). In Mode 1, the 8031 uses System B's EOC signal. As the sampled data is being read-back by the 8031, it is sent out byte by byte on the SSB until the full requested frame set has been collected. The order of the channels in the reply frame is the same as was requested, preceded by D1, the number of channels. *Only one frame is sent in reply where after the 8031 returns to a neutral, no-mode state.*

Mode 2

This mode is identical to Mode 1, except that the 8031 now sets up System C's EOC signal to generate the interrupt that signals that a valid channel can be read by the 8031. *Only one frame is sent in reply where after the 8031 returns to a neutral, no-mode state.*

The remainder of the modes follows a similar procedure.

The communication interface is a PC, used to emulate the OBC. The PC is connected to the micro-controller serially, and sends commands using the SSB protocol. The PC was also used for easier debugging of the software during the development phase. Test statements are inserted at target places inside the software to monitor the state of variables during program execution.

6.4 Integration

This is the final phase of the software development process and involves porting of the developed telemetry software into the telemetry system hardware. The integration setup is illustrated in Figure 6.2. The setup consists of the SUNSAT telemetry board, a communication interface and a data interface.

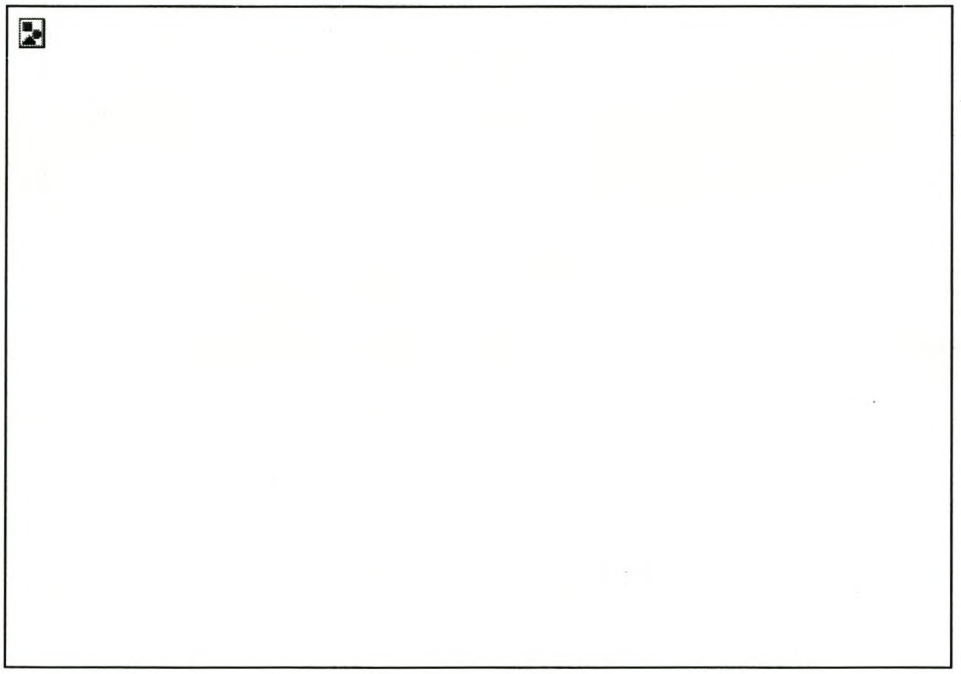

Figure 6.2: *The integration setup*