1

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

PRELIMINARY FLIGHT SOFTWARE SPECIFICATION FOR THE PETITE AMATEUR NAVY SATELLITE (PANSAT)

by

Teresa Owen Ford

March, 1994

Thesis Advisor:	Douglas J. Fouts
Second Reader:	Frederick W. Terman

Approved for public release; distribution unlimited.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704

1. AGENCY USE ONLY	2. REPORT DATE March, 1994	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE PRELIMINARY FLIGHT SOFTWARE SPECIFICATION FOR THE PETITE AMATEUR NAVY SATELLITE (PANSAT)	5. FUNDING NUMBERS
6. AUTHOR(S) Ford, Teresa Owen	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT (maximum 200 words)

PANSAT is a small, spread-spectrum, communications satellite under design at the Naval Postgraduate School. It will support a store and forward bulletin board system for use by the amateur radio community. The flight software is responsible for the autonomous telemetry collection and hardware control operations of the satellite, communications and file transfer protocols allowing access to the bulletin board system, and command interpretation and response to ground control commands. In this thesis, the complete flight software architecture and module interfaces are specified using the Estelle Formal Description Technique. The module bodies dealing with communications and file transfer protocols are specified in detail in Estelle. The current design goal for the remainder of the flight software modules are discussed. Appendices include the preliminary flight software specification itself, a data flow diagram interpretation of the specification, and a summary of the Estelle syntax used.

14. SUBJECT TERMS PANSAT, Software Specification, Estelle, Amateur Radio, Store and Forward	15. NUMBER OF PAGES 211
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

Preliminary Flight Software Specification For the Petite Amateur Navy Satellite (PANSAT)

by

Teresa Owen Ford
Lieutenant, United States Navy
B.S., United States Naval Academy, 1985

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 1994

Author: [illegible]
Teresa Owen Ford

Approved by: Douglas J. Fouts
Douglas J. Fouts, Thesis Advisor

Frederick W. Terman
Frederick W. Terman, Second Reader

Michael A. Morgan
Michael A. Morgan, Chairman
Department of Electrical and Computer Engineering

ABSTRACT

PANSAT is a small, spread-spectrum, communications satellite under design at the Naval Postgraduate School. It will support a store and forward bulletin board system for use by the amateur radio community. The flight software is responsible for the autonomous telemetry collection and hardware control operations of the satellite, communications and file transfer protocols allowing access to the bulletin board system, and command interpretation and response to ground control commands.

In this thesis, the complete flight software architecture and module interfaces are specified using the Estelle Formal Description Technique. The module bodies dealing with communications and file transfer protocols are specified in detail in Estelle. The current design goals for the remainder of the flight software modules are discussed. Appendices include the preliminary flight software specification itself, a data flow diagram interpretation of the specification, and a summary of the Estelle syntax used.

TABLE OF CONTENTS

LIST OF TABLES

SYMBOL TABLE

SYMBOL	WHERE USED	MEANING
bold	thesis and specification	A Pascal or Estelle reserved word.
italics	thesis and specification	A specification constant, including a member of an enumerated type.
ALL_CAPS	thesis and specification	Module name or state name.
Beginning_Caps	thesis and specification	Name of a user defined type.
'single_quotes'	thesis	Name of a variable or record field in the specification, or the contents of a variable or record field.
0xNN	thesis and specification	A hexadecimal number. "N" stands for the digits 0 through F.
uchar	thesis and specification (primitive data type)	"Unsigned character": 8 bits of binary data, a 1 byte unsigned integer, or 1 ascii character.
uint	thesis and specification (primitive data type)	"Unsigned integer": 16 bits of binary data, a 2 byte unsigned integer, or 2 ascii characters.
ulong	thesis and specification (primitive data type)	"Unsigned long integer": 32 bits of binary data, a 4 byte unsigned integer, or 4 ascii characters.
int	thesis and specification (primitive data type)	"Integer": a signed integer.

ACKNOWLEDGEMENT

I would like thank my advisor, Professor Fouts, for allowing me infinite flexibility in my approach to this thesis, and my second reader, Professor Terman, for his diligent proof reading and many helpful comments. Very many thanks go to the knowledgeable Mr. Jim Horning of the Space Systems Academic Group, without whose tireless assistance this project would never have been begun, much less completed.

Also I would like to thank my husband, Bruce, my daughter, Robin, and my son, Douglas, whose loving support enabled me to persevere.

I. INTRODUCTION

A. PANSAT

The Petite Amateur Navy Satellite (PANSAT) is a small, experimental, communications satellite currently being designed and constructed at the Naval Postgraduate School (NPS) and scheduled to be launched from the space shuttle in 1996. The satellite will use half-duplex, spread-spectrum communications and will support a store and forward bulletin board system for use by the amateur "HAM" radio community. PANSAT will operate autonomously in the performance of many of its functions, while also carrying out commands issued by the ground control station located in Monterey, CA at NPS.

B. GOALS OF THE FLIGHT SOFTWARE

The flight software will control the autonomous operation of the satellite, allow amateur radio operators access to the onboard bulletin board or "mailbox" system, and provide the means for the satellite to respond to commands from the ground control station. The major functional areas of the flight software are listed in Table 1.1. There are other software functions which are equally important to the PANSAT project, but outside the scope of the flight software. These include the "bootstrap" software which will control initial configuration of the satellite systems upon launch or reboot, "client" ground station software for use by the amateur radio community, and "commanding" software for the ground control station.

TABLE 1.1 FUNCTIONAL AREAS OF THE FLIGHT SOFTWARE
1. Communications Protocols for Amateur Radio User Access and File Transfer
2. Telemetry Gathering and Storage
3. Control of Satellite Hardware Systems
4. Command Interpretation and Response to NPS Ground Station Control

C. SCOPE OF THIS THESIS

This thesis is the result of the initial attempt at specification of the flight software for PANSAT. The hardware systems are still evolving, which means that hardware control, command interpretation, and telemetry gathering requirements are still being defined. The only requirements which can be completely defined this early and remain essentially unchanged, regardless of the final hardware configuration of the satellite, are the communications protocol and mail handling functions. For this reason, the actual preliminary software specification, which can be found in Appendix A, concentrates on the mailbox system and the transfer protocols for uploading and downloading files. The preliminary architecture for the remaining software modules is sketched out, and interfaces have been defined between all modules.

Some existing third party software, designed specifically for communications satellites, will be used aboard PANSAT to support the applications software being developed at NPS. The commercial software used will be introduced in Chapter II. A PANSAT file header has been developed to assist in proper maintenance of the mail storage system. The elements of this file header will be explained in Chapter III. Chapter IV will discuss the specification language, Estelle, in which Appendix A is written. Chapters V through VIII will explain the functionality of the software modules which have received the most attention in Appendix A, those dealing with file transfer and mailbox control. Chapter IX will introduce the preliminary goals for the remainder of the flight software, which has yet to be specified in detail. Chapter X will contain conclusions and recommendations for further work. Appendix B contains a graphical interpretation of the specification in Appendix A. This interpretation uses data flow diagrams which provide a visual means of identifying the software architecture and module interfaces. Appendix C will provide some details of the syntax of Estelle.

II. THIRD PARTY SOFTWARE

A. SPACE CRAFT OPERATING SYSTEM

The role of DOS (Disc Operating System) on a personal computer is filled by SCOS (Space Craft Operating System) on the computer aboard PANSAT. SCOS is a real time, multi-tasking, operational environment designed specifically for the needs of a small satellite.[Ref. 1] It supports applications written in the "C" programming language, and many of the primitive functions familiar to "C" programmers are available through the Space Craft Operating System. These include file management capabilities, dynamic memory allocation, bit and byte manipulations, and logical and mathematical operations. Within the SCOS operating environment, the various modules which make up the flight software are running as concurrent processes, able to pass data among themselves. They are put to "sleep" when they are not needed, saving CPU time, and are "woken up" again whenever their services are required.

B. BAX

1. AX.25

AX.25 is a variant of the CCITT X.25 link-layer protocol, and is designed to provide reliable data transport between two signaling terminals.[Refs. 2, 3] This asynchronous data transfer protocol is currently in wide use by the amateur radio community for packet communications, and has thus been selected for use in communications with PANSAT. To incorporate AX.25 functionality into the communications software aboard PANSAT, a program called BAX is employed.

BAX is the BekTek corporation's implementation of the AX.25 protocol. It is designed to work with the Space Craft Operating System. Because of the availability of the BAX software, the actual functionality of the AX.25 protocol is transparent to the applications programs developed for the satellite. The software designer and programmer need only consult the BAX manual [Ref. 4] to discover how to access the capabilities required. For the amateur radio operator, the AX.25 protocol is normally implemented by a piece of dedicated hardware known as a TNC (terminal node controller), or by software contained in a PC (personal computer) based system.

2. BAX Application Programs

In order to utilize the capabilities of BAX, a "BAX application" program must be written. In the PANSAT flight software specification, the primary BAX application is known as the DATA_TRANSFER module (see Chapter V). Other modules, such as GROUND_CONTROL, may also access the services of BAX.

BAX has the capability of receiving frames addressed to various applications which are distinguished from each other by having different ssid's (subsystem identification numbers). PANSAT will be addressed by a multi-character Amateur Radio Callsign, "PANSAT", for example. This callsign may be modified by use of ssid's to access different functions aboard the satellite. For instance, the data transfer module will have ssid '1', and the HAMs will send their mail to "PANSAT-1". The command interpreter, GROUND_CONTROL, may have ssid '2', so that commands from the NPS ground station could be sent there directly by BAX without having to be in the same format as that recognized by the bulletin board system.

BAX communicates directly with the hardware drivers which operate the radio equipment on the satellite. As incoming frames are received, BAX handles all of the AX.25 protocol requirements, and notifies the appropriate application module of the receipt and the source of each transmission. In the case of the data transfer module, the frames passed to it by BAX must be assembled into the packets required at the next higher protocol level. This is the level of the PACKET_TRANSFER modules described in Chapters VI and VII. When the data transfer module receives a packet from a packet transfer module for transmission to a ground user, the data transfer module breaks the data into frames which it passes "down" to BAX for transmission in accordance with the AX.25 protocol.

3. BAX Functions

The accessing of most BAX functions by PANSAT application modules is represented in the software specification as messages being passed through an Abstract_Bax_Channel. In fact, in the specification model, all communication between modules is accomplished via "channels", each channel having certain types of messages defined which can be passed through it. These channel and message definitions form the interface specification between software modules (see Chapter III and Appendix A). The BAX functions accessed are listed and briefly explained in Table 2.1.

TABLE 2.1 BAX FUNCTIONS

Function	Purpose
qax_input	BAX informs PANSAT application of user connection request, user disconnect, or incoming data frame.
qax_claim	PANSAT application tells BAX what callsign and ssid it will be using.
qax_data	PANSAT application passes data to BAX for transmission.
qax_busy	PANSAT application informs BAX that it is "busy" and will not be accepting incoming frames.
qax_unbusy	PANSAT application informs BAX that it is no longer "busy" and will once again accept incoming frames.
qax_con_acpt	PANSAT application accepts a user connection request
qax_con_rej	PANSAT application rejects a user connection request
qax_connect	PANSAT application sends a connection request to ground station
qax_ui	PANSAT application sends an unnumbered information frame to a ground station.
qax_ disconnect	PANSAT application disconnects from a ground station.

C. FILE TRANSFER LEVEL 0

File Transfer Level 0 (FTL0) is an asynchronous connected mode file transfer protocol developed by Jeff Ward and Harold E. Price for use with the PACSATs (packet satellites). In a connected mode protocol, there is a virtual link between each user and the satellite, with each transmission having a specific destination. This is in contrast to a broadcast, or unconnected mode protocol, in which communications are intended to be picked up by anyone listening.

An implementation of FTL0 is currently available to amateur radio operators in the form of the program "PG" along with several utility programs that work along with it, such as "PHS" and "PFHADD". Although the specification for FTL0 contains provisions for both uploading and downloading files from a satellite, only the uploading capabilities are implemented by the current version of "PG". For downloading from the satellites which currently employ FTL0, the non-connected mode, "PACSAT Broadcast Protocol" [Ref. 5], is used. This is implemented by the program "PB" and it's utilities.

The specification for FTL0 [Ref. 6] is used as the motivation for the specification of the PACKET_TRANSFER module aboard PaANSAT (Chapters VI and VII). The specification of the packet transfer module in Appendix A is much more detailed than [Ref. 6], in an attempt to show how the protocol will actually be implemented by the software aboard the satellite at the lowest possible level. The PANSAT implementation will employ an FTL0-like connected-mode protocol for both upload and download.

An effort has been made to remain as compatible as possible with any other FTL0 implementation. Some of the specific requirements of PANSAT and certain design decisions have led to some variation from [Ref. 6]. As source code for "PG" was not available, it is unknown at this point whether that software will actually be able to communicate with PANSAT. PANSAT-specific ground station software, capable of communicating with the packet transfer module specified here, will be developed by NPS and made available to the amateur radio community.

III. THE PANSAT FILE HEADER

A. FUNCTION

Each file maintained in the mail box memory of the satellite must begin with a PANSAT file header. The header includes information such as the file number, file name, file length, source callsign, destination callsigns, upload time, and expiration time. The information in the header is necessary for the proper maintenance and administration of the mail box. It can also be used by a client to determine which files onboard the satellite may be of interest. The *select_cmd* makes use of the various fields of the PANSAT file header in its selection criteria (see Chapter VI).

B. STRUCTURE

The PANSAT file header is inspired by, but is not the same as, the PACSAT File Header developed by Jeff Ward and Harold Price [Ref. 7]. It is arranged as a variable length array of unsigned characters (bytes). The fields nearest to the beginning of the header have fixed positions and fixed lengths. The lengths of other fields are specified within the header itself, causing the positions of the later fields to be variable, and dependant on the fields ahead of them.

The byte positions, field names and formats are listed in Table 3.1. Note that positions and field lengths through byte 41 are fixed. Each of the fields "Destination 1" through "Destination 7" is either present, with a fixed length of 6 bytes, or absent completely, based upon the contents of the "Number of Destinations" field. The fields "Title" and "Keywords" have variable length, based upon the contents of the fields "Title Length" and "Keyword Length", respectively. The column Const refers to the constant name given to the associated field in the specification of Appendix A.

TABLE 3.1 PANSAT FILE HEADER FIELDS

Byte(s)	Const	Name	Format
[0..1] fixed	*fl*	Flag	<0xBB> <0x55>
[2..5] fixed	*mn*	Mail Number	ulong
[6..9] fixed	*ml*	File Length	ulong
[10] fixed	*ft*	File Type	uchar
[11] fixed	*ct*	Compression Type	uchar
[12..13] fixed	*bo*	Body Offset	uint
[14] fixed	*dc*	Download Count	uchar
[15..20] fixed	*sc*	Source	array[6] of uchar
[21] fixed	*pr*	Priority	uchar
[22..25] fixed	*ut*	Upload Time	ulong
[26..29] fixed	*et*	Expire Time	ulong
[30..37] fixed	*na*	PANSAT File Name	array[8] of uchar
[38..40] fixed	*ex*	PANSAT File Extension	array[3] of uchar
[41] fixed	*nd*	Number of Destinations	uchar
[42..47] approx.	*ds*	Destination 1	array[6] of uchar
[48..53] approx.		Destination 2	array[6] of uchar
[54..59] approx.		Destination 3	array[6] of uchar
[60..65] approx.		Destination 4	array[6] of uchar
[66..71] approx.		Destination 5	array[6] of uchar

TABLE 3.1 PANSAT FILE HEADER FIELDS

Byte(s)	Const	Name	Format
[72..77] approx.		Destination 6	array[6] of uchar
[78..83] approx.		Destination 7	array[6] of uchar
[84] approx.		Title Length	uchar
[85..114] approx.	*ti*	Title	array[30] of uchar
[115] approx.		Keyword Length	uchar
[116..195] approx.	*kw*	Keywords	array[80] of uchar
[196..197] approx.		Header Checksum	uint
[198..199] approx.		Body Checksum	uint

C. FIELDS TO BE FILLED IN BY SOURCE

A PANSAT file header must be prepended to any file before it is uploaded to the satellite. Certain fields within the header must be completed by the user station where the file originates, while other fields are filled in by the satellite once the file has been completely uploaded. The user must place all zeros in those fields which the satellite will complete. These satellite responsible fields will all be of fixed length. The fields for which the user is responsible are as follows:

1. Flag

The flag indicates that this is the beginning of a file with a PANSAT file header. The flag must always consist of the same two bytes: '0xBB' followed by '0x55'. Example of the flag field: 10111011 01010101.

2. Mail Number

A mail number, or file number, is assigned by PANSAT to each file. This number is not known to the user until it is provided by the satellite in an *upload_go_resp* following an *upload_cmd* from the ground station. (See Chapter VI, section E.). The user software has two options here. The simplest is to leave 4 bytes of 0's in this field, and let the satellite update it after the upload. If the upload is interrupted, however, it will be the responsibility of the user software to "remember" the number associated with the partially uploaded file, and to provide it to the satellite in the next *upload_cmd* which will continue the process. The obvious place to store the number is in the file header. For this reason, it may make more sense to choose the second option, which is to place the proper number in the header before transmission of the file begins. Of course, this will also necessitate adjusting the header checksum before transmitting. (See subsection 12).

3. File Length

The file length is a four byte unsigned integer (ulong). The source software must place in this field the number of bytes contained in the file, including the PANSAT file header.

4. File Type

The file type is a one byte field which indicates the format of the file body. The satellite does not care about the file format, as it treats all files simply as arrays of bytes. The information in this field is for the use of anyone who downloads the file, so that they will know how it must be read. The contents of this field will be interpreted as in Table 3.2. The first eleven of these types are the same as those defined by Price and Ward in [Ref. 7], and some of them might never be used aboard PANSAT. They are included for completeness, and to provide as much parallelism as possible between this specification and FTL0.

TABLE 3.2 FILE TYPES

Field Contents	File Type
00000000	ASCII file
00000001	RLI/MBL message body. Single message.
00000010	RLI/MBL import/export file. Multiple message.
00000011	UoSAT Whole Orbit Data
00000100	Microsat Whole Orbit Data
00000101	UoSAT CPE Data
00000110	MS/PC-DOS .exe file
00000111	MS/PC-DOS .com file
00001000	Keplerian elements NASA 2-line format
00001001	Keplerian elements "AMSAT" format
00001010	Simple ASCII text file, but compressed

TABLE 3.2 FILE TYPES	
Field Contents	**File Type**
10100000	PANSAT short format telemetry file
10100001	PANSAT long format telemetry file
10100010	PANSAT bax telemetry file
11111110	User defined type.

5. Compression Type

If the body of the file is compressed, the source must indicate the type of compression used in this one byte field. Again, the satellite does not care whether or not a file is compressed, or if so, how. This information is for the use of the downloading user only. Note that no matter what file format or compression type is used in the file body, the PANSAT file header will always be uncompressed ASCII text. Compression types are indicated by Table 3.3. "User defined type" in Table 3.2 and "Other" in Table 3.3 indicate that a file format or compression type not listed is being used. The user must know the type, perhaps based on the source or title.

TABLE 3.3 COMPRESSION TYPES	
Field Contents	**Compression Type**
00000000	No compression
00000001	PKARC
00000010	PKZIP
00000011	Other

6. Body Offset

The body offset is a two byte unsigned integer (uint). The source must enter in this field the byte number at which the file body begins; that is, the number of the byte following the last byte in the PANSAT file header. This is where the file format and compression type will take effect, as far as the ground user is concerned. Note that the first byte in the file header is number 0. If there are 200 bytes in the file header, then the body offset will be '200' (0x00C8).

7. Source

The source field identifies the origin of the file, or the ground station from which it was uploaded. The uploading user's HAM callsign, consisting of six ASCII characters, must be entered in this field by the client software.

8. Priority

No particular use for the one byte priority field is currently specified for the satellite software. The user is free to use this field for his own purposes, such as to indicate the relative urgency of messages to addressees who share the same interpretation for this field. Any one byte bit pattern may be entered in the field, as long as the header checksum takes the contents into account.

9. Number of Destinations and Destination 1 through Destination 7

If the message to be uploaded is intended for receipt by between 1 and 7 individual destination stations, then this is the number which is placed in the one byte unsigned integer of the "Number of Destinations" field. The appropriate number of

"Destination" fields are then used to indicate the HAM callsigns of the addressees. Any unused destination fields are left out of the header. If the source wishes to indicate that a message is intended for all users, then the number '0x00' is placed in the "Number of Destinations" field, and no destination fields are used.

To modify the "all users" destination, the uploading station may choose to include a "source path" or a "destination path" to further define the intended audience for the file. If a source path is to be included, then the number '0x08' is placed in the "Number of Destinations" field, and all 7 of the destination fields are included as a single 42-byte path field. Any ASCII string may be placed in this field to indicate a source path or other source identification. Similarly, if a destination path is to be included, the number '0x09' is placed in the "Number of Destinations" field, and the 42-byte path field is used to indicate a destination path or to identify the intended audience.

The satellite will not attempt to interpret destination paths or identifications. It is up to the potential downloaders to use this information, either by reading it after downloading file directories, or by providing strings to compare with the path field in *select_cmds* (see Chapter VI).

10. Title Length and Title

The "Title" field is a variable length array of from 0 to 30 bytes. The "Title Length" field must be entered by the source to indicate the actual length of the title. The title should be an ASCII string which will indicate to potential downloaders the contents of the file body. If there is an original file name, which it is important to keep with the file, it may be entered here. PANSAT does not otherwise retain original file names, assigning its own after upload. The title information is for the use of potential downloaders only, and the satellite does not attempt to interpret this field.

11. Keyword Length and Keywords

Like the "Title" field, the "Keywords" field is a variable length array of ASCII characters. The "Keyword Length" field must be used to specify the actual length, of between 0 and 80 bytes. Keywords should be separated from each other by one or more spaces. The satellite does interpret keyword information, but will attempt to find keywords of interest within this field if so directed by a *select_cmd*.

12. Header Checksum and Body Checksum

The header and body checksums are used to verify the integrity of a file after uploading or downloading is complete. The body checksum must be calculated first, since it is included in the header and is thus a factor in the header checksum. All bytes in the body of the file are added together as unsigned 8-bit integers. The least significant two bytes of the resulting sum are placed in the body checksum field. The header checksum is the result of adding all bytes in the header together, except for the header checksum itself, and taking the least significant two bytes of the sum. The source must take care to update the checksums if any part of the file or header is changed before the actual upload begins. When the file number to be used has been identified by the satellite, for instance, if the source then replaces the zeroes in the "File Number" field with the proper number, those four bytes must also be added to the header checksum.

If the source is not able or does not desire to calculate these checksums, either or both of them may be left out. In this case, the fields must be filled with all zeroes. The satellite will not fill in or update "all zero" checksum fields. When the satellite performs file integrity checks, any all zero checksum will be ignored and that check will be skipped. Because of this, corrupted files may remain aboard the satellite undetected. It is up to the file's source to determine whether checksums are required. Currently, there is no way to distinquish between "no checksum" and a checksum which is actually '0'. Consequently, any checksum which is calculated to be exactly zero will be ignored. This can be remedied by adding a character to the Keywords or Title field, adjusting the appropriate length field and recalculating the checksum.

D. FIELDS TO BE FILLED IN BY SATELLITE

As previously stated, the uploading source of a file may choose to leave the "File Number" field of the PANSAT file header filled with zeroes. If the satellite finds that this has been done, it will fill in this field and update the header checksum appropriately. When the satellite "updates" a checksum, it does so simply by adding to it any bytes with which it has replaced zeroes. When nonzero field contents must be changed, the existing bytes are subtracted from the checksum, and the new bytes added to it. This happens, for instance, when the "Download Count" is updated (see below). The least significant two bytes of the sum are placed into the checksum field. The checksum is not completely recalculated, as this would invalidate the purpose of checking the integrity of the bytes uploaded.

There are several additional fields within the PANSAT file header which must be filled in by the satellite. The satellite software updates the header checksum whenever it places information in any of these fields. The satellite will never change a body checksum. The satellite responsible fields are described in the following subsections.

1. Download Count

In this one byte field, the satellite software keeps track of how many times a particular file has been successfully downloaded. For a file addressed to "all", the download count is incremented each time a download is completed. For a file addressed to between one and seven individual callsigns, the count is incremented only when a download is completed to one of the intended addressees. The information in this field is used to determine whether a file has been previously downloaded when the default selection list is being formed (see Chapter VI, section K). The satellite software looks at the header of each file to see if the current client is one of the addressees. If there are five addressees listed, for instance, and the client is one of them, but the download count is already at '5', it is assumed that the client has already downloaded this file.

2. Upload Time and Expire Time

The "Upload Time" field is filled in after a complete file has been successfully uploaded. When the final bytes of a file have been received, and the file has passed the integrity checks (such as checksums), the satellite "stamps" it with its current onboard time. This time is in the form of a four byte unsigned integer which is a count of the number of seconds since January 1, 1970 (the UTC, or Universal Time

Constant). Then the current amount of time allotted to each file to stay aboard the satellite is added to the upload time to form the expiration time. This number is placed in the Expire Time field. The amount of time allowed for each file may change based upon satellite loading. When the expiration time for a file is exceeded by the clock onboard the satellite, that file is discarded.

3. PANSAT File Name and PANSAT File Extension

As each file is uploaded to the satellite, the satellite software assigns a DOS file name and extension to it. This is the file name which will be used by the onboard file management system to access the file. It is also used to easily associate each file with it's source without having to read any header fields. The 8-byte file name assigned consists of the 6 character source callsign preceded by two ASCII spaces. The file extension consists of 3 ASCII numerals (0 through 9), which indicate the sequence of files uploaded from this particular source. For instance, the first file uploaded by callsign ABCDEF would be named " ABCDEF.001", the second would be " ABCDEF.002", etc. Extensions are repeated after number "999". It is unlikely that any file would remain with an extension that is up for re-use. But if that happens, the next unused extension will be assigned instead.

Certain file names are used by the satellite to indicate particular kinds of files generated aboard the satellite, rather than uploaded by users. These include "BULLETIN.xxx" and "USRTELEM.xxx". Files with the name "BULLETIN" contain information of general interest posted by the satellite or ground control operators and addressed to all users. Files with the name "USRTELEM" contain satellite telemetry which may be of interest to users. USRTELEM files will be in the format "PANSAT short telemetry file". This format has not been completely specified as yet, and will be published at a later date.

IV. THE SPECIFICATION LANGUAGE - ESTELLE

A. FORMAL DESCRIPTION TECHNIQUE

A formal description technique (FDT) is a method of precisely defining the behavior of a system. It is generally advantageous to employ an FDT in the design and specification of software because descriptions produced in this way tend to be more complete, consistent, precise, concise, and unambiguous than descriptions produced in a natural language, such as English. For the specification of the PANSAT flight software, the language Estelle has been chosen. Estelle is a formal description technique which is based on an extended state transition model and uses much of the familiar syntax of the programming language Pascal.[Ref. 8]

As stated in Chapter II, the operating system chosen for the computer aboard PANSAT supports software written in the "C" programming language. For this as well as other reasons, such as development and debugging tools currently available to the Space Systems Academic Group at NPS, the implementation languages for the flight software will be "C", "C++", and assembly code as required. In spite of this, there are many reasons for developing the software specification in a description language like Estelle, prior to implementing it in a compilable language such as "C". Some of these reasons are addressed in the following sections.

B. CLARITY

One of the most important aspects of a software specification is clarity. The purpose of the specification is to clearly communicate the intended behavior of the program to those who must actually write the software (both the original version and later revisions) as well as to those who must use it. The behavior described by the specification must be verifiable to be the correct behavior by the systems designers who define the requirements of the system. A programming language like "C" is certainly very precise, but is often lacking in the required clarity, at least as far as humans other than the original programmer are concerned.

A particular "C" statement is written in a particular way and will cause a particular event to occur. What is not obvious is whether the particular event that occurs is exactly the event intended. When some software requirement is translated directly from an English description into a programming language implementation, there are several dangers. First of all, it is difficult to guarantee that the English description is sufficiently unambiguous that it will be understood and translated in exactly the same way by everyone. Second, if the translation *is* off somewhat and the software written implements a slightly different requirement than that intended, it can be difficult to catch the error by examining the code. Third, since the code *is* more precise than the original English description, it may be tempting to use *it* as the description of required behavior as the program is debugged and modified. Some programming languages, "C" in particular, are sufficiently terse that it can be difficult to extract a complete understanding of the intended behavior directly from the code without intense examination. Comments are used to alleviate this problem - and we are back to the ambiguities of the English language. Of course, even if a precise description of behavior is extracted from the code and comments, it may no longer be the intended one.

The Pascal syntax used in Estelle, though more precise and unambiguous than English, is more obvious and easily readable than "C". Simple, well-understood, and extremely precise programming language constructs are used. These include **while** statements, **if-then-else** constructs, and **for** loops, as well as function and procedure calls [Ref. 8]. The specific Pascal syntax used in Appendix A is summarized in Appendix C, Table C.1. Pascal was developed as an educational language and is designed specifically to be clear and easily understood; the syntax is very straight forward. The intricate and often inscrutable statement construction of a high-powered language such as "C" is avoided.

Using Estelle, an English description of required behavior can be translated into a precise series of program-like statements. These statements are sufficiently readable that the resulting behavior can be easily analyzed and compared with the intended requirements. An ambiguous requirement statement is made crystal clear, once it has been set down in the proper series of precise Estelle statements. Once the formal specification is in place, there should be only one way to translate it, the correct way. Any software implementation must then be checked against the required behavior imparted by the Estelle description. When the program does not act in a useful way, it can be easily determined whether the original requirements statement was at fault, or whether the program code is flawed. When the software must be modified throughout the life cycle of the host system, the originally intended behavior of the existing code will be more easily understood from the specification than from the code itself.

Of course, the specification must be maintained up-to-date along with the code. If the system requirements change, this must be reflected in the specification. The specification should always be the most accurate description of the currently intended behavior of the software system.

C. STATE MACHINE MODEL

Many software systems, including communications protocols, can be modeled as state machines. A major function of the PANSAT flight software is to implement communications and file transfer protocols between the ground users and the satellite. State machines provide a convenient way of modeling the software and describing its required behavior. Estelle extends the syntax of Pascal to include constructs specifically designed to clearly convey a state machine architecture. The behavior of each module is defined by its reactions to each legal stimulus it may receive while in each specific state. Even where several different states are not required for the proper functioning of a module, the state machine architecture still provides a convenient way to show the module's reactions to different inputs, and provides a means of identifying what inputs are anticipated and legal and what inputs are illegal or unexpected.

While individual statements primarily use common Pascal syntax, the hierarchy of the program modules and the module interfaces are defined by the Estelle state machine model. There are Estelle-specific reserved words which are used to establish the state machine architecture and to define other aspects of the specification which are beyond the scope of the Pascal syntax. These reserved words, the specification segments with which they are associated, and their functions are listed in Appendix C, Table C.2.

D. MODULE COMMUNICATIONS

Communications between various program modules are very clearly defined in Estelle. What types of information are passed between precisely which program modules is thoroughly spelled out. The set of **channel** definitions, which controls the flow of information between modules, is also the module interface definition.

In the software specification, several **channels** are defined. Each channel definition includes a list of the message types which can be "passed" through that channel. Each end of the channel is named and the message types are direction-specific. For instance, module 'A', attached to the 'User' end of a particular channel, may request information from module 'B', at the 'Provider' end, using one of several different 'request' messages. Module 'B' will reply using one of a completely different set of 'response' messages.

The name of a message type and the channel it is passed through may in itself provide all the information that is needed. In other situations, specific parameters must be passed. The parameters to be passed with each message are listed in parenthesis next to the message name in the channel definition. Estelle is a strongly typed language, and this requirement extends to the parameters passed between modules. The type of each parameter is indicated in the message definition.

Each program module has a module header definition and a module body definition. The module header definition includes a list of all the "interaction points" available to the module. These interaction points are channels, and the end of the channel to which the module is attached is indicated for each. The interaction points are the only means by which information can be passed from one module to another. The nature of each information exchange is thus precisely defined. In the **modvar** section at the end of the software specification, channels are attached explicitly between the various modules. The **channel** definitions, module header definitions, and the **modvar** section, taken together, completely define the architecture of the software system.

E. DETAIL AND ABSTRACTION

One final advantage of using a formal description technique such as Estelle, is that various levels of abstraction can be used *to clarify the specification*. Abstraction can be used to ignore details irrelevant to the context at any point, so that the local complexity of the description can be decreased and the overall understanding increased. Abstraction can also be used to continue with a description even though some essential details are not yet known. Commonly used functions, such as those assumed to be readily available from the operating system, can be defined as "primitives", the actual details of their internal implementations unimportant. Hardware specific details which are not known when the specification is being developed can be defined abstractly, with the specifics to be filled in later.

In contrast, any level of detail desired can be included. Thus, if minute details of the specific hardware implementation to be used are known, they can be indicated in the specification to avoid mistakes. Detailed algorithms which demonstrate a method for obtaining the specific results desired can be drawn out. In the specification of Appendix A, the communications protocols and mailbox control are described in somewhat minute detail, at a level where specific hardware requirements are unimportant. The portions of the specification dependant upon hardware configurations are merely indicated in a high-level architecture, with all details to be worked out as more information becomes available.

V. DATA TRANSFER MODULE

A. FUNCTION

The DATA_TRANSFER module provides the interface between the high level file transfer protocol used by PANSAT and the BAX link-level AX.25 protocol software. It is the primary "BAX application program." The PACKET_TRANSFER module, described in Chapters VI and VII, relies on the data transfer module to reassemble the AX.25 level frames passed from BAX into the complete packets uplinked from the ground station. The data transfer module also receives packets from the packet transfer module, breaks them down into frames, and passes them on to BAX to be transmitted to the intended user.

B. THE BAX CONTROL BLOCK

Communication with the BAX program is accomplished via the BAX functions listed in Chapter II. These are represented in the Estelle specification of Appendix A by the message types within the Abstract_Bax_Channel. Many of these messages have a parameter of the type Control_Block. The control block is a data structure defined in [Ref. 4] which carries much of the actual information passed between BAX and the application program. QAX_CLEAN_CB is a BAX function which provides a control block structure with all fields initialized to zero. This is the only BAX function referenced in the specification by a procedure call rather than by a message type.

The definition of the Control_Block type appears somewhat differently in Appendix A than in [Ref. 4]. It has been altered to match the syntax and avoid the reserved words of the remainder of the specification and includes only those fields which are actually used by the data transfer module. The control block fields used are listed in Table 5.1 along with the information each conveys.

TABLE 5.1 BAX CONTROL BLOCK FIELDS

App. A Name	[Ref. 4] Name	Type	Information
link	channel	uint	Indicates which of the 30 possible BAX links a frame has come in on, or which it should be sent out over. In effect, designates the ground user at the other end.
kind	type	enumerated Frame_Type	Indicates the type of information carried by the control block. If *qat_data*, then the data from a data frame has been placed in an Fdata buffer, included as another message parameter. If *qat_state*, then the state of the link has changed, and the new state is indicated by the '1_state' field. If *qat_ui*, then an unnumbered information frame has been received.

TABLE 5.1 BAX CONTROL BLOCK FIELDS

App. A Name	**[Ref. 4] Name**	**Type**	**Information**
l_state	state	enumerated Link_State	Indicates the new link state in a *qat_state* kind of Control_Block. Only two of these states concern the data transfer module: *qas_connect_pend* indicates that a user has requested to be connected with the satellite. *qas_disconnected* indicates that a user link has been terminated.. The reason for termination can be determined from the 'why' field.
why	cause	enumerated Cause	Indicates the reason for a link state change. Causes include *qac_local* (action of the satellite), *qac_remote* (action of the ground station), *qac_remotefrmr* (AX.25 protocol error) and *qac_timeout* (maximum number of frame retries exceeded).
my_call	struct AX25_ADDR my_call	Callsign_Type = array[6] of uchar	Indicates the application program's call sign, which is always the satellite's call sign.
my_ssid		uchar	Indicates the application's subsystem identification number (to distinguish the data transfer module from the ground control module, for instance).

TABLE 5.1 BAX CONTROL BLOCK FIELDS

App. A Name	[Ref. 4] Name	Type	Information
his_call	struct AX25_ADDR his_call	Callsign_Type	Indicates the ground station's call sign.
his_ssid		uchar	Indicates the ground station's subsystem identification number
t1	t1	uchar	The number of seconds to use for the link-level frame timeout-timer. If a frame acknowledgement is not received within t1 seconds of transmission, the frame must be retransmitted.
maxframe	maxframe	uchar	The maximum number of frames "in flight" at one time - the link-level sliding window size. Must be 1 - 7.
retry	retry	uchar	The maximum number of times to retry a frame transmission before terminating the link.
paclen	paclen	uint	The maximum size of the data field on an outgoing frame. Must be < = 256 bytes.

C. STATES OF THE DATA TRANSFER MODULE

The data transfer module has only two states, NORMAL and BUSY. The module is initialized in the NORMAL state, and is expected to remain in that state for the majority of the time. A transition to the BUSY state occurs only as the result of a message from the either the GROUND_CONTROL module or the AUTO_CONTROL module.

If the performance of the satellite, as judged by the onboard decision-making software or by the ground control station, deteriorates to the point where it seems beneficial to allow fewer users to access the satellite for a period of time, a lockout message can be sent to the data transfer module. The type of lockout, ('l_kind') may be *new*-user or *all*-user. When a *new*-user lockout message is received, the data transfer module remains in the NORMAL state, but rejects all new user connection requests. The satellite will continue communications with all users already logged on when the message is received. When the data transfer module receives an *all*-user lockout message, the state will change to BUSY and incoming communications from everyone except the NPS ground control station will be rejected. The data transfer module will send a 'busy' message to every BAX link, and BAX will respond to any frame (except those from NPS) with a "receive-not-ready" frame.

The control software may also find it necessary to turn the transmitter off for an extended period of time, such as during a battery recharge. When this occurs, and the transmitter will not be ready at a moment's notice, a 'transmitter.off' message will be sent to the data transfer module. This message will not change the state of the module, which will still be able to receive any incoming frames, but it will change the state variable 'transmit_ok' to false. When this occurs, the data transfer module will not attempt to transmit any frames, and any logged-in users will most likely disconnect due to frame time-outs.

VI. PACKET TRANSFER MODULE - PACKET TYPES

A. FILE TRANSFER LEVEL 0

The packet transfer module specified in Appendix A has its origins in the File Transfer Level 0 (FTL0) Pacsat Protocol developed by Jeff Ward and Harold E. Price [Ref. 7]. The basic data structures and state transitions are functionally equivalent to FTL0, with some modifications. In order to make use of the satellite software specified in Appendix A, the corresponding ground station software must be developed which will produce packets in the proper format to be interpreted by PANSAT. Therefore, the bit-level structure of the ground station packets is described below, as well as the proper ground interpretation of the packets originating on the satellite.

B. PACKET FORMAT

Each packet to be transmitted consists of an information field of 0 to 2047 bytes, preceded by a two byte header. The header identifies the type of packet and indicates the number of bytes in the information field. The packet structure is defined as follows in the software specification:

```
Packet_Type    = record
      length_lsb:      uchar;
      hl:              uchar;
      info:            Pdata;
end;
```

This structure indicates that the header portion of the packet consists of the two unsigned characters (octets) 'length_lsb' and 'hl'. The information field is given the type 'Pdata', which is defined in the specification as an array of 0 to 2047 unsigned characters. Since this is a variable length array, its length must be indicated in the header.

The octet 'length_lsb' contains the least significant 8 bits of the data length. The octet 'hl' contains the 3 most significant bits of the data length, as well as an indication of the type of packet. The bits of 'hl' are labeled '76543210'. Bits 7-5 are the 3 most significant bits of the data length, and must be prepended to the 'length_lsb' to give the full length of the information field. Bits 4-0 of 'hl' provide a number from 0 to 31. This number is decoded into packet type as indicated in Table 6.1.

TABLE 6.1 PACKET TYPES

Packet Number	Specification Constant	Packet Name
0	*data*	Data
1	*data_end*	Data End
2	*login_resp*	Login Response
3	*upload_cmd*	Upload Command
4	*ul_go_resp*	Upload Go Response
5	*ul_error_resp*	Upload Error Response
6	*ul_ack_resp*	Upload Acknowledged Response
7	*ul_nak_resp*	Upload Not Acknowledged Response
8	*download_cmd*	Download Command
9	*dl_error_resp*	Download Error Response
10	*dl_aborted_resp*	Download Aborted Response
11	*dl_completed_resp*	Download Completed Response

TABLE 6.1 PACKET TYPES		
Packet Number	**Specification Constant**	**Packet Name**
12	*dl_ack_cmd*	Download Acknowledged Command
13	*dl_nak_cmd*	Download Not Acknowledged Command
14	*dir_short_cmd*	Directory Short Command
15	*dir_long_cmd*	Directory Long Command
16	*select_cmd*	Select Command
17	*select_resp*	Select Response
18 - 29		reserved
30	*del_cmd*	Delete Command
31	*del_resp*	Delete Response

The ground software and satellite software are peer entities at this level, rather than master and slave. However, since the ground must initiate all data exchanges, with the satellite acting as a server responding to requests made from the ground, the identifier 'cmd' is used to indicate packets originating from the ground, while 'resp' indicates packets sent from the satellite. The *data* and *data_end* packets can originate from either the ground station or the satellite. Explanations of each of the packet types and the contents of their information fields are given in the following sections.

C. THE DATA AND DATA END PACKETS

Any file to be transmitted, either from the ground or from the satellite, will be broken up into an appropriate number of *data* packets, depending on its length. The information field of each *data* packet will be the bytes from the file to be transmitted. Bits 4-0 of the 'hl' field of the packet header will be '00000', identifying the packet as containing file data in its information field. Bits 7-5 of 'hl' and the octet 'length_lsb' will together indicate the number of bytes of file data being transmitted in this packet. The transmission of the end of the file will be indicated by sending a *data_end* packet immediately after the transmission of the last *data* packet. The *data_end* packet has no 'info' field.

Bit-level examples of the various packet types will be given in Tables 6.2A through 6.2S. In these examples, 0's and 1's will be shown where particular bit patterns must be used. Where arbitrary bit patterns may be present, other symbols will be used. The intended meanings of these symbols will be made clear in the "interpretation" section of each table.

TABLE 6.2A EXAMPLE OF A *data* PACKET

length_lsb	hl	info
LLLLLLLL	HHH 00000	dddddddd dddddddd ...
	Interpretation	
low order bits of the data length	HHH = high order bits of the data length. '00000' = *data* packet.	file data in bytes

TABLE 6.2B EXAMPLE OF A *data_end* PACKET	
length_lsb	**hl**
00000000	000 00001
Interpretation	
no info field	'00001' = *data_end* packet.

D. THE LOGIN RESPONSE PACKET

Neither the FTL0 protocol of Ward and Price, nor the packet transfer protocol specified here, has an explicit Login Command packet. A login request from the user is made implicitly whenever a data link is established between the ground station and the satellite on the lower, AX.25, data transfer level. When the AX.25 protocol software, BAX, recognizes a "connection request" frame from a new user, it informs the satellite data transfer module. The data transfer module tells BAX whether to accept the connection or not. If the connection is accepted, BAX sends the "accept connection" frame to the ground user, and the data transfer module informs the packet transfer module that a data link has been established. At this point, the satellite packet transfer protocol calls for the transmission of a Login Response packet.

The purpose of the *login_resp* packet is to inform the user of the time onboard the satellite when the data link is established. The *login_resp* packet also has a one byte login flag. This flag indicates whether the user currently has an active selection list (explained below) and whether the satellite requires Pacsat file headers. The Pacsat file header was developed by Jeff Ward and Harold Price for use with their Pacsat Protocol

Suite, which includes FTLO [Ref. 8]. A PANSA'.' file header has been developed which does not match the Pacsat file header of Ward and Price, and so the login flag in the *login_resp* packet will always indicate that Pacsat file headers are not required. PANSAT file headers will always be required for files uploaded to PANSAT. T h e information field of the *login_resp* packet includes a 4-byte unsigned integer indicating the login time (the number of seconds since January 1, 1970), followed by a 1-byte login flag. Thus, the information length indicated by the header must be 5. Bits 7-4 of the login flag will be '0000'. Bit 3, the 's' bit, will be '1' if the client already has an active selection list, and will be '0' if not. Bit 2 will be '0', indicating that Pacsat file headers are not used. Bits 1 and 0 indicate the protocol version number. They will be '00' in the case of the protocol specified in Appendix A.

TABLE 6.2C EXAMPLE OF A *login_resp* PACKET

length_lsb	hl	info
00000101	000 00010	tttttttt tttttttt tttttttt tttttttt 0000s000
Interpretation		
5 bytes in info field	*login_resp* packet	4 byte login time login flag: 's' = '1' indicates active selection list

E. THE UPLOAD COMMAND, UPLOAD GO RESPONSE, AND UPLOAD ERROR RESPONSE PACKETS

Before uploading a file to the satellite, the client software must first determine whether there is room in the mail box and if the satellite will accept the upload. To make this determination, the ground station transmits the *upload_cmd*. The satellite will respond with the *ul_go_resp* if the upload will be allowed and with the *ul_error_resp* if not.

The information field of the *upload_cmd* contains a 4-byte file number followed by a 4-byte file length. If this is the first request to upload a particular file, the file number must be '0x00000000'. The file length must be the actual length of the file which is intended for upload, including the PANSAT file header, which must be prepended to each file. When the satellite receives the *upload_cmd*, it will determine if there is room for a file of the indicated length. If there is, the *ul_go_resp* will include a file number to be assigned to the file in the mail box aboard the satellite. When the client receives this file number, it may be placed in the PANSAT file header before upload. If the client does not place the proper file number in the PANSAT file header before upload, then that field must contain all 0's and the satellite will make the correction once the file has been successfully uploaded.

If the upload request is for the continuation of a previously interrupted upload, the *upload_cmd* must contain the actual file number previously assigned by the satellite. The file length must still indicate the full length of the file, regardless of how much of the file was previously uploaded. If the satellite can accept this continued upload, the *ul_go_resp* will include the offset at which the client should begin the transmission of the file. To determine this offset, the satellite simply inspects its partial copy of the file to see how many bytes it has previously received. The complete information field of the *ul_go_resp* includes the 4-byte file number either newly or previously assigned to the file, followed by the 4-byte file offset. If no part of the file has been previously uploaded, then the indicated offset will be '0'.

If there is no room for the file, if the *upload_cmd* includes a non-zero file number which does not correspond to any file onboard the satellite, or if the satellite determines that upload of the indicated file has already been completed, then an *ul_error_resp* is transmitted rather than an *ul_go_resp*. The *ul_error_resp* has a 1-byte information field which simply indicates one of several possible error conditions. The possible errors associated with an upload command and their corresponding bit patterns are indicated in Table 6.3A.

TABLE 6.3A ERROR CODES

Bit Pattern	**Error Code**
00000001	*er_ill_formed_cmd*
00000010	*er_bad_continue*
00000100	*er_no_such_file_number*
00001100	*er_file_complete*
00001101	*er_no_room*

The meanings of most of these error codes are obvious from their names. The code *er_bad_continue* is issued when the file number in the *upload_cmd* is non-zero but the file length in the *upload_cmd* does not agree with the file length stored in the PANSAT header of the partially uploaded file.

TABLE 6.2D EXAMPLES OF *upload_cmd* PACKETS

length_lsb	hl	info
00001000	000 00011	00000000 00000000 00000000 00000000 LLLLLLLL LLLLLLLL LLLLLLLL LLLLLLLL
00001000	000 00011	nnnnnnnn nnnnnnnn nnnnnnnn nnnnnnnn LLLLLLLL LLLLLLLL LLLLLLLL LLLLLLLL
Interpretations		
8 bytes in info field	*upload_ cmd* packet	New upload - unknown file number 4 byte file length
8 bytes in info field	*upload_ cmd* packet	4 byte file number 4 byte file length

TABLE 6.2E EXAMPLE OF AN *ul_go_resp* PACKET

length_lsb	hl	info
00001000	000 00100	nnnnnnnn nnnnnnnn nnnnnnnn nnnnnnnn oooooooo oooooooo oooooooo oooooooo
Interpretation		
8 bytes in info field	*ul_go_resp* packet	4 byte file number 4 byte file offset at which to begin upload

TABLE 6.2F EXAMPLE OF AN *ul_error_resp* PACKET

length_lsb	hl	info
00000001	000 00101	eeeeeeee
Interpretation		
1 byte in info field	*ul_error_resp*	1 byte error code

F. THE UPLOAD ACKNOWLEDGED RESPONSE AND UPLOAD NOT ACKNOWLEDGED RESPONSE PACKETS

When the client software on the ground receives an *ul_go_resp* from the satellite, it will commence to upload the file in a series of data packets, starting with the byte of the file indicated by the offset in the *ul_go_resp*. Once the *data* packet containing the last byte of the file has been transmitted, the *data_end* packet must be sent. After receiving the *data_end* packet, the satellite will check the integrity of the file, as will be explained in Chapter VIII. If the file passes all checks and is successfully stored aboard the satellite, an *ul_ack_resp* will be transmitted to the user. If the file is found to be defective, an *ul_nak_resp* is transmitted instead and the file is discarded. The satellite will remember the file number, however, so that the user can later attempt another upload of the same file.

The *ul_ack_resp* has no information field. The *ul_nak_resp* has a 1-byte information field which consists of one of the error codes of Table 6.3B.

TABLE 6.3B ERROR CODES

Bit Pattern	Error Code
00001101	*er_no_room*
00001110	*er_bad_header*
00001111	*er_header_check*
00010000	*er_body_check*

The *er_bad_header* code is sent when the PANSAT file header is missing, incomplete, or incorrect. The code for *er_header_check* is sent when the checksum on the header fails and *er_body_check* is sent when the checksum on the file body fails.

An *ul_nak_resp* may also be sent by the satellite before a *data_end* packet is received if the satellite needs to terminate the upload for any reason. If the ground station receives an *ul_nak_resp*, it should immediately stop sending *data* packets, and transmit a *data_end* packet if it has not already done so.

TABLE 6.2G EXAMPLE OF AN *ul_ack_resp* PACKET

length_lsb	hl
00000000	000 00110
Interpretation	
no info field	*ul_ack_resp*

TABLE 6.2H EXAMPLE OF AN *ul_nak_resp* PACKET

length_lsb	**hl**	**info**
00000001	000 00111	eeeeeeee
	Interpretation	
1 byte in info field	*ul_nak_resp*	1 byte error code

G. THE DOWNLOAD COMMAND PACKET

Prior to using *download_cmd* packets to request files to be downloaded from PANSAT, the client must establish an active selection list onboard the satellite. This is achieved by using the *select_cmd* which is explained below. Once the client has used the selection list to begin downloading a file or to obtain file directories (see Directory Commands below), the selection list need not remain active to continue downloading, as long as the client knows the file number for each file requested.

Each *download_cmd* packet is used to request a single file. The information field of this packet contains the 4-byte file number for the file requested followed by the 4-byte file offset from which transmission of the file should begin. In FTL0, a file number of '0x00000000' is used to indicate the next file in the active selection list, proceeding from newer files toward older files, while '0xFFFFFFFF' requests the next file in the list proceeding from older files toward newer files. In the current packet transfer specification for PANSAT, both '0x00000000' and '0xFFFFFFFF' will result in requesting the next file in the active selection list proceeding from older files toward newer files. I' he actual file number of the file requested is known, then this number is placed in the *download_cmd* packet. The file offset should be '0' if this is a new download request, and should indicate the byte number from which to proceed if this is a download continuation.

TABLE 6.21 EXAMPLES OF *download_cmd* PACKETS

length_lsb	hl	info
00001000	000 01000	00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00001000	000 01000	11111111 11111111 11111111 11111111 00000000 00000000 00000000 00000000
00001000	000 01000	nnnnnnnn nnnnnnnn nnnnnnnn nnnnnnnn oooooooo oooooooo oooooooo oooooooo
Interpretations		
8 bytes in info field	*download_ cmd*	requesting next file in select list begin download at beginning of file
8 bytes in info field	*download_ cmd*	requesting next file in select list begin download at beginning of file
8 bytes in info field	*download_ cmd*	file number of requested file file offset at which to begin download

When requesting the "next" file in the selection list, the offset should always be '0', since this should only be used to request a new file. Once a download has been interrupted and subsequently continued, the file number should already be known, and this information as well as the offset should be used in the *download_cmd*. If the file offset indicated by the ground station is equal to or greater than the length of the file stored on the satellite, no error is generated. Instead, the satellite transmits a *data_end* packet immediately, with no preceding *data* packets.

H. THE DOWNLOAD ERROR RESPONSE PACKET

When the satellite receives a properly formatted *download_cmd* which it is able to respond to, it immediately begins downloading the file in a series of *data* packets. Once the last byte of the file has been transmitted, the satellite sends a *data_end* packet. If, however, the satellite cannot service the *download_cmd* for any reason, it will transmit a *dl_error_resp* packet.

The *dl_error_resp* information field consists of a 1-byte error code. The possible errors are shown in Table 6.3C.

TABLE 6.3C ERROR CODES

Bit Pattern	Error Code
00000100	*er_no_such_file_number*
00000101	*er_selection_empty*

The code *er_no_such_file_number* is used if a specific file has been requested, the file number of which is not found in the mail box. The code *er_selection_empty* is used when the "next" file is requested, but the user currently has no active selection list. The specification for FTL0 also includes error codes dealing with file forwarding capabilities which are not implemented on PANSAT. These codes are included in the specification of Appendix A for the sake of completion, to ensure they will not be used for any PANSAT specific definitions. They will not be included in any *dl_error_resp* packets from PANSAT, however. These FTL0 error codes, unused by PANSAT, include *er_already_locked* and *er_no_such_destination*.

TABLE 6.2J EXAMPLE OF A *dl_error_resp* PACKET

length_lsb	hl	info
00000001	000 01001	eeeeeeee
Interpretation		
1 byte info field	*dl_error_resp*	1 byte error code

I. THE DOWNLOAD ACKNOWLEDGED COMMAND, DOWNLOAD COMPLETED RESPONSE, DOWNLOAD NOT ACKNOWLEDGED COMMAND, AND DOWNLOAD ABORTED RESPONSE PACKETS

When the client software receives a *data_end* packet from the satellite, it knows the downloaded file is complete. It performs any desired integrity checks on the file (such as checking header and body check sums) to determine whether the download was completed successfully. If the file has been received satisfactorily, the ground station must transmit a *dl_ack_cmd*. The satellite responds to a *dl_ack_cmd* with a *dl_completed_resp* to end the download process. If the ground software does not find the downloaded file to be satisfactory, it transmits a *dl_nak_cmd*, to which the satellite responds with a *dl_aborted_resp*.

In the FTL0 specification, the information field of the *dl_ack_cmd* consists of a one byte 'register_destination'. This information is used by a Pacsat to complete some of the file forwarding operations which are not implemented by the current PANSAT software specification. Therefore, the information field is not necessary in a *dl_ack_cmd* sent to PANSAT, and it will be ignored if it is included. This single byte of information may be adapted for use by PANSAT at a later date. The *dl_completed_resp* transmitted by the satellite has no information field. The *dl_nak_cmd* and the *dl_aborted_resp* likewise have no information fields.

TABLE 6.2K EXAMPLE OF A *dl_ack_cmd* PACKET

length_lsb	hl	info
00000001	000 01100	xxxxxxxx
Interpretation		
1-byte info field	*dl_ack_cmd*	register_destination

TABLE 6.2L EXAMPLE OF A *dl_completed_resp* PACKET

length_lsb	hl
00000000	000 01011
Interpretation	
no info field	*dl_completed_resp*

TABLE 6.2M EXAMPLE OF A *dl_nak_cmd* PACKET

length_lsb	hl
00000000	000 01101
Interpretation	
no info field	*dl_nak_cmd*

TABLE 6.2N EXAMPLE OF A *dl_aborted_resp* PACKET

length_lsb	hl
00000000	000 01010
Interpretation	
no info field	*dl_aborted_resp*

J. THE DIRECTORY COMMAND PACKETS

FTL0 specifies two directory commands, the *dir_short_cmd* and the *dir_long_cmd*. For PANSAT, there is only one directory command, and either of these two packet types will invoke it. The results of each command will be exactly the same. The information field of a *dir_cmd* is a 4-byte file number. This number indicates the file for which a directory entry is requested. A directory entry consists of the PANSAT file header from the file of interest. From this file header, the ground station software can determine any necessary information about the file. The user can decide from this information whether to request the file for download.

In FTL0, a file number of '0x00000000' is used to request the directory entries for the next 10 files in the active selection list, proceeding from newer files toward older files, while '0xFFFFFFFF' requests the next 10 file directories proceeding from older files toward newer files. In the current packet transfer specification for PANSAT, both '0x00000000' and '0xFFFFFFFF' will result in requesting entries for the next 10 files in the active selection list proceeding from older files toward newer files. If the client has no currently active selection list, then a directory entry can only be requested for a file for which the file number is already known.

When the satellite receives a correctly formatted *dir_cmd* which it can respond to, it sends the requested information down in a *data* packet. Since PANSAT file headers are at most 200 bytes long, 10 of them will fit in a single packet. Thus, after one data packet is transmitted, a *data_end* packet will immediately be sent. If the satellite is unable to respond to the *dir_cmd*, it will send a *dl_error_resp* indicating the reason. The error character contained in this packet will be either *er_selection_empty* or *er_no_such_file_number*.

TABLE 6.20 EXAMPLES OF *dir_cmd* PACKETS

length_lsb	**hl**	**info**
00000100	000 01110	00000000 00000000 00000000 00000000
00000100	000 01111	11111111 11111111 11111111 11111111
00000100	000 01111	nnnnnnnn nnnnnnnn nnnnnnnn nnnnnnnn
Interpretations		
4 bytes in info field	*dir_short_cmd*	requesting directory entries for next 10 files in select list
4 bytes in info field	*dir_long_cmd*	requesting directory entries for next 10 files in select list
4 bytes in info field	*dir_long_cmd*	file number of file for which directory entry is requested

K. THE SELECT COMMAND AND SELECT RESPONSE PACKETS

The *select_cmd* is the means by which the user designates files to be placed in an active selection list onboard the satellite. Once this list has been established, it can be used to request file directories or files for download. The *select_cmd* specified by FTL0 assumes use of Pacsat file headers in its structure. A different *select_cmd* structure is specified here, which is based upon the PANSAT file header. This structure is related to, but not exactly the same as, the *select_cmd* specified by Price and Ward [Ref 1.].

Once a *select_cmd* is recognized, if it is not in the PANSAT structure specified here, then a default selection list will be compiled. This list will be comprised of all mail addressed to the requesting user which has not been previously downloaded, all bulletins and user-accessible telemetry and messages addressed to "all". The satellite will send a *select_resp* packet to the user. The information field of this packet consists of a two byte integer which indicates the number of files in the selection list.

If the PANSAT select structure is used, the selection list will be assembled according to the selection criteria contained in the *select_cmd*. If the satellite can interpret the *select_cmd* and compile the corresponding selection list it will transmit the *select_resp* packet. In this case, the two byte integer in the information field indicates the number of files matching the selection criteria. An active list consisting of those file numbers will be maintained aboard the satellite. If the *select_cmd* appears to be in the PANSAT format but cannot be successfully parsed by the satellite software, then a *dl_error_resp* is transmitted with the error code: 00001000 *er_poorly_formed_sel.*

The *select_cmd* has a variable length information field. The information contained in the field consists of a PANSAT specific flag, the number of selection criteria present, and the criteria themselves. The selection criteria are combined with each other using the operators **and** and **or**, forming a restricted type of postfix logical equation. In this postfix equation, each logical operator is preceded by its two operands. The first two selection criteria are combined logically, according to the first operator, to form the single operand **true** or **false**. This operand is then followed by another selection and another operator. The second operator combines the two operands preceding it to form a single operand. The process is continued until the last logical operator present, which will be the last component of the equation, has been used to combine its two operands. The result will be a single value of **true** or **false**. Each file for which the selection equation yields a value of **true** will have its file number added to the active selection list aboard the satellite.

The first byte of the *select_cmd* information field should be '0xFF', a flag indicating that the *select_cmd* is in the PANSAT format. Upon recognizing this flag, the satellite will attempt to translate the *select_cmd* into the appropriate logical equation. If this flag is not present, the default selection list described above will be compiled and the remainder of the select structure will be discarded.

The second byte in the information field is an unsigned integer indicating the number of selection criteria contained in the remainder of the structure. The selection criteria can be defined as follows:

```
selection = record
    relop:          uchar;
    header_item:    uchar;
    item_len:       uchar;
    compare_item:   array[item_len] of uchar;
end;
```

Bit '7' of the one byte 'relop' (relational operator) must always = '0'. Bits '654' have the interpretations shown in Table 6.4, and bits '3210' are translated as indicated in Table 6.5. The 'header_item' identifies which item in the PANSAT file header to compare the 'compare_item' with. The one byte 'header_item' is decoded as Table 6.6 indicates.

TABLE 6.4 BITS '654' OF THE RELATIONAL OPERATOR

Bits 654	Relation
000	equal to
001	greater than
010	less than
011	not equal to
100	greater than or equal to
101	less than or equal to

TABLE 6.5 BITS '3210' OF THE RELATIONAL OPERATOR

Bits 3210	Interpretation of 'compare_item'	Notes
0000	Multi-byte unsigned integer	'item_len' must equal 1, 2 or 4.
0011	Array of characters, convert to lower case before comparison	Valid only with Bits '654' = '000' or '011'

TABLE 6.6 HEADER ITEMS		
'header_item' Bit Pattern	**Const**	**Name of Header Field**
00000000	*fl*	Flag
00000010	*mn*	Mail Number (File Number)
00000110	*ml*	Mail Length
00001010	*ft*	File Type
00001011	*ct*	Compression Type
00001100	*bo*	Body Offset
00001110	*dc*	Download Count
00001111	*sc*	Source Call Sign
00010101	*pr*	Priority
00010110	*ut*	Upload Time
00011010	*et*	Expire Time
00011110	*na*	PANSAT File Name
00100110	*ex*	PANSAT File Extension
00101001	*nd*	Number of Destinations
00101010	*ds*	Destination Call Signs or Path
01010100	*ti*	Title
01110100	*kw*	Keywords

The short integers formed by the 'header_item' bit patterns correspond to the normal byte offsets within the PANSAT file header of the beginning of each of the listed header fields. This is useful in other areas of the software specification.

The one byte integer 'item_len' gives the byte length of the last item in the 'selection', the 'compare_item.' The compare item is interpreted, as indicated by the 'relop', as either an unsigned one, two or four byte integer, or an array of characters. The relational operation specified by the 'relop' is performed between the designated header item and the compare item. If the relation 'header_item' 'relop' 'compare_item' is satisfied, then this selection equation is evaluated as true.

If the user has only one criteria for selection, the Select_Structure can end after just one 'selection'. The user may, however, specify multiple selection criteria. For this purpose, the bit patterns for logical operators are defined as in Table 6.7.

TABLE 6.7 LOGICAL OPERATORS

'logop' Bit Pattern	Logical Operation
10000000	and
11100000	or

A completed Select_Structure may appear as follows:

0xFF num_selections selection selection logop selection logop selection logop,etc.

When 'equal to' string comparisons are made between 'compare_items' and certain header fields, the comparison is defined as successful if the 'compare_item' string is found anywhere within the header item string. In these same fields, a 'not equal to' comparison is successful if the 'compare_item' string is not found anywhere within the header item string. Header fields for which this applies are the "Title" and "Keywords" fields. This can also apply to the destination fields under certain circumstances, which will be elaborated on below.

To indicate that a file is addressed to "all", the source places the number '0x00' in the "Number of Destinations" field, and the following 7 "Destination" fields are left out of the header completely. A one-byte integer comparison between the "Number of Destinations" field and the number '0x00' can determine that a file is addressed to all users. When there are between 1 and 7 individual destination call signs, this number is placed in the "Number of Destinations" field, and the appropriate number of "Destination" fields are included. To find files which are addressed to a specific user, a string comparison can be made between a 6-byte call sign as the 'compare_item' and the header item "Destination Call Signs or Path". This header item refers to all "Destination" fields present. The satellite will compare the designated call sign to each destination listed, and the comparison will be successful if a match is found with any of them.

It may be that the user wishes to designate an audience for the file which is broader than 7 individual call signs but narrower than "all" users in the world. In order to achieve this, the user places the number '0x08' or the number '0x09' in the "Number of Destinations" field, and all 7 "Destination" fields are then included as a single 42-byte array. In this combined field can be placed information to further define the audience for which the file is intended. If the "Number of Destinations" is '0x08', then the file is addressed to "all", and the source has included path, location or other information about *himself* in the following "Path" field. If the "Number of Destinations" is '0x09', then the source has included further information about the intended audience in the following "Path" field. Users may use this information in *select_cmds*, in which case any 'compare_item' will be searched for anywhere within the "Path" field. The header item to use within the 'selection' will again be "Destination Call Signs or Path", but this time the satellite will look for any matching string, not simply matching 6-byte call signs. Source or destination path information can probably better be used by ground software as the result of *dir_cmds*, in which case the software can present the user with any information which may help the user in choosing individual files to download.

TABLE 6.2P EXAMPLES OF *select_cmd* PACKETS

length_lsb	hl	info
00001001	000 10000	11111111 00000001 0 001 0000 00010110 00000100 tttttttt tttttttt tttttttt tttttttt
00011001	000 10000	11111111 00000011 0 000 0011 01110100 00000100 01000001 01010010 01001101 01011001 0 000 0011 01110100 00000100 01001110 01000001 01010110 01011001 11100000 0 001 0000 00010110 00000100 tttttttt tttttttt tttttttt tttttttt 10000000

Interpretations

TABLE 6.2P EXAMPLES OF *select_cmd* PACKETS

9 bytes in info field	*select_cmd* packet	PANSAT *select_cmd* flag, 1 'selection' 'relop' - greater than a multibyte unsigned integer 'header_item' - Upload Time 4 byte 'compare_item' - last login time (This command requests all files which were uploaded after the user's last login time.)
25 bytes in info field	*select_cmd* packet	PANSAT *select_cmd* flag, 3 'selections' 'relop' - equal to an array of characters 'header_item' - Keyword 4 characters in the 'compare_item' 'compare_item' - "ARMY" 'relop' - equal to an array of characters 'header_item' - Keyword 4 characters in the 'compare_item' 'compare_item' - "NAVY" 'logop' - or 'relop' - greater than a multibyte unsigned integer 'header_item' - Upload Time 4 byte 'compare_item' - last login time 'logop' - and (This command requests all files which were uploaded after the user's last login time and which also have either "ARMY" or "NAVY" as a keyword.)

TABLE 6.2Q EXAMPLE OF A *select_resp* PACKET

length_lsb	hl	info
00000010	000 10001	nnnnnnnn nnnnnnnn
Interpretation		
2 byte info field	*select_resp*	Number of files placed in select list

L. THE DELETE COMMAND AND DELETE RESPONSE PACKETS

The specification for FTL0 does not allow for a user-requested file deletion. It assumes that files will simply remain aboard the satellite until their expiration dates are exceeded, or until the satellite itself or the ground controllers cause the files to be deleted. This *del_cmd*, therefore, has no equivalent in FTL0.

The information field of the *del_cmd* contains only the 4-byte file number of the file to be deleted. The satellite responds to a *del_cmd* with a *del_resp* packet. The information field of this packet merely contains one of the one-byte error codes of Table 6.3D.

TABLE 6.3D ERROR CODES

Bit Pattern	Error Code
00000000	*no_error*
00000100	*er_no_such_file_number*
10010000	*er_permission_denied*

If the satellite indicates *no_error* in the *del_resp*, then the file has been successfully deleted. A user may only delete files uploaded by him or addressed to him as the sole destination. The satellite ensures these criteria are met by inspecting the appropriate fields in the file's header before any deletion is carried out. An attempt to delete any other file will result in *er_permission_denied*.

TABLE 6.2R EXAMPLE OF A *del_cmd* PACKET

length_lsb	hl	info
00000100	000 11110	nnnnnnnn nnnnnnnn nnnnnnnn nnnnnnnn
Interpretation		
4-byte info field	*del_cmd*	file number of file to be deleted

TABLE 6.2S EXAMPLE OF A *del_resp* PACKET

length_lsb	hl	info
00000001	000 11111	eeeeeeee
Interpretation		
1-byte info field	*del_resp*	1-byte error code

VII. PACKET TRANSFER MODULE - STATE TRANSITIONS

A. TRIGGERS

The operation of the packet transfer module is based upon state transitions triggered by the receipt of packets from the user and messages from other flight software modules. When the user on the ground sends a *cmd* packet to the satellite, the response will depend, in part, on what state the packet transfer module is in.

The architecture of the packet transfer state machine can easily be seen in the Estelle specification of Appendix A. The **trans** section of the module definition clearly shows all possible transitions from one state to another, along with what packet or message triggers each transition, and what action is taken as a result. This textual description can be translated into a more visual format by means of state transition graphs, such as those included with the data flow diagrams of Appendix B.

B. INSTANTIATIONS

It is intended that multiple users should have access to the mail box onboard the satellite "simultaneously". This is achievable because all user transmissions to the satellite are packetized. BAX, the AX.25 data transfer level software, can administer up to 30 user links at once. As each AX.25 frame is received, BAX determines which user it is from, and deals with it according to the AX.25 protocol and the state of the link with that particular user.

In order for the packet transfer level to also be administered for many "simultaneous" users, there must be a copy of the packet transfer module associated with each virtual link. The fact that multiple copies of the packet transfer module are initialized can be seen in the **modvar** section at the end of the Estelle software specification. This says that one packet transfer module is created for each link. The definition of "Link_Type", near the beginning of the specification, indicates that there are between 0 and 30 links. When a packet transfer module receives messages from, or sends messages to, another software module, the specific instantiation involved is indicated by the initialization parameter 'link'. The packet transfer modules, as well as the channels associated with them, are referenced as array elements; the 'link' each is associated with acts as the array index. For instance, in the module header definition of the MAILBOX_CONTROL_TYPE, it is stated that this module has an array of 30 (*maxlinks*) Mailbox_Access_Channels. In the **modvar** section, each of these channels is connected to a different copy of the packet transfer module.

C. TRANSITIONS

The state transitions for a particular instantiation of the packet transfer module are affected only by packets from the user associated with the link being administered by that module. The data transfer module, as explained in Chapter V, must assemble complete packets from the frame data sent to it by BAX. Each packet is sent to the appropriate packet transfer module, depending upon which BAX link it was received on. Likewise, as the packet transfer modules send *resp* packets to the data transfer module for transmission via BAX, the data transfer module must break each packet up into frames and send them via the appropriate BAX link to reach the intended recipient.

While a packet transfer module is in any particular state, only certain packets from the user will have meaning. An unexpected packet will cause any actions in progress (such as uploading or downloading a file) to be aborted. FTL0 defines unexpected or incorrect packets as sufficient cause to terminate the link with a user. The specification in Appendix A, however, only calls for the packet transfer module to return to a waiting state after abandoning any action in progress. At this point, the module is ready to accept any valid command from the user. The user will be informed of the problem via an appropriate *error_resp* packet. After receipt of any error message, the user should assume that the satellite is waiting for the ground station to initiate a new action.

D. STATES

FTL0 was designed primarily for use with satellites with full duplex capabilities. For this reason, it maintains two separate state machines, one for the uplink process and the second for the downlink process. PANSAT is a half-duplex communications satellite. The two state machines of FTL0 have been combined into a single machine in the specification of the PANSAT packet transfer module. The states are listed in Table 7.1. Explanations are included in the following subsections.

TABLE 7.1 PACKET TRANSFER STATES

State Identifier	Explanation of State
UL/DL_UNINIT	Upload/Download Uninitiated
UL/DL_CMD_WAIT	Waiting for an Upload or Download Command
WAIT_MAILBOX	Waiting for a Message from the Mailbox Control Module
UL_DATA_RX	Ready to Uplink Data
UL_ABORT	Upload Aborted
DL_FILE_DATA	Downloading a File

1. UL/DL_UNINIT

UL/DL_UNINIT is the state into which the packet transfer module is first initialized, before a user link has been established with it. In this state, the module does nothing but wait to be assigned a user. Upon receipt of the 'connection' message from the data transfer module, the packet transfer module asks the mailbox control module whether or not the new user has an active selection list, and moves into the WAIT_MAILBOX state to await a reply. When the reply message is received, the module will enter the UL/DL_CMD_WAIT state. The packet transfer module returns to the UL/DL_UNINIT state when it is sent a 'disconnect' message by the data transfer module, regardless of what state it is in when this message is received.

TABLE 7.2 STATE TRANSITIONS FROM UL/DL_UNINIT		
Received Message	**Action**	**Next State**
connection	active_sl_req message (Asks the mailbox control module if there is an active selection list for this user.)	WAIT_MAILBOX

2. UL/DL_CMD_WAIT

In the UL/DL_CMD_WAIT state, the packet transfer module is waiting for a packet from the user which will initiate either an upload process or a download process. Packets which can be legally received while in this state are listed in Table 7.3, along with the resultant actions and transitions. Any other, unexpected, packets will result in an *ul_error_resp* packet with the error code *er_ill_formed_cmd*, and the module will remain in the state UL/DL_CMD_WAIT.

TABLE 7.3 STATE TRANSITIONS FROM UL/DL_CMD_WAIT		
Received Packet or Message	**Action**	**Next State**
upload_cmd	mail_num_req message (Request a new file number or a current file offset from the mailbox control module.)	WAIT_MAILBOX
del_cmd	mail_del_req message (Request that the mailbox control module delete a file.)	WAIT_MAILBOX
select_cmd	mselect_req message (Request the mailbox control module form a selection list.)	WAIT_MAILBOX

TABLE 7.3 STATE TRANSITIONS FROM UL/DL_CMD_WAIT

Received Packet or Message	Action	Next State
dir_short_cmd *dir_long_cmd*	dir_req message (Request directory information from the mailbox control module.)	AIT_MAILBOX
download_cmd	mail_req message (Request file data from mailbox control module.)	DL_FILE_DATA
dl_nak_cmd	none	UL/DL_CMD_WAIT
disconnect	none	UL/DL_UNINIT
other packets	*ul_error_resp* packet	UL/DL_CMD_WAIT

3. WAIT_MAILBOX

As can be seen in Table 7.3, most packets received while in the UL/DL_CMD_WAIT state result in a transition to the WAIT_MAILBOX state, with no immediate response packet to the user. This is because the packet transfer module requires information from the mailbox control module before it can make a proper reply to the user. The mailbox control module analyzes each information request message and replies with an appropriate response message. The response of the mailbox control module will determine which state the packet transfer module will enter when it leaves the WAIT_MAILBOX state, as well as what packet it sends to the user. The WAIT_MAILBOX state may also be entered from the UL/DL_UNINIT state as shown above, or the UL_DATA_RX state, as will be explained below. Table 7.4 summarizes the mailbox access channel messages just prior to a transition to the WAIT_MAILBOX state, the possible reply messages from the mailbox control module, and the resulting further actions and state transitions of the packet transfer module. No user command packets are expected while in the WAIT_MAILBOX state, as the user should still be waiting for a reply from the last packet sent to the satellite.

TABLE 7.4 STATE TRANSITIONS FROM WAIT_MAILBOX

Message	Reply Message	Action	Next State
active_sl_req	active_sl_resp	*login_resp* packet	UL/DL_CMD_WAIT
mail_num_req	mail_num_resp, no errors	*ul_go_resp* packet	UL_DATA_RX
	mail_num_resp, error	*ul_error_resp* packet	UL/DL_CMD_WAIT
mail_recv	mail_recv_resp, no errors	Change current upload offset	UL_DATA_RX
	mail_recv_resp, error	*ul_nak_resp* packet	UL_ABORT
mail_close_req	mail_close_resp, no errors	*ul_ack_resp* packet	UL/DL_CMD_WAIT
	mail_close_resp, error	*ul_nak_resp* packet	UL_ABORT
mail_del_req	mail_del_resp	*del_resp* packet	UL/DL_CMD_WAIT
mselect_req	mselect_resp, no errors	*select_resp* packet	UL/DL_CMD_WAIT
	mselect_resp, error	*dl_error_resp* packet	UL/DL_CMD_WAIT

TABLE 7.4 STATE TRANSITIONS FROM WAIT_MAILBOX

Message	Reply Message	Action	Next State
dir_req	directory, no errors	*data* packet, *data_end* packet	UL/DL_CMD_WAIT
	directory, error	*dl_error_resp* packet	UL/DL_CMD_WAIT
disconnect			UL/DL_UNINIT

4. UL_DATA_RX

The packet transfer module enters the UL_DATA_RX state after the user has sent an *upload_cmd* and the mailbox control module has replied to the resulting inquiry with the appropriate file number or offset. That is, this state is first entered from the WAIT_MAILBOX state. When the module is in the UL_DATA_RX state, it is ready to receive *data* packets from the user. As each packet it received, the packet transfer module passes the file data on to the mailbox control module for storage, entering the WAIT_MAILBOX state each time to await acknowledgement. When the *data_end* packet is received, the packet transfer module requests that the mailbox control module close the file and conduct integrity checks on it. The result of these checks will determine whether the packet transfer module returns directly to the UL/DL_CMD_WAIT state, or goes into the UL_ABORT state, as indicated in Table 7.4. The state transitions out of UL_DATA_RX are summarized in Table 7.5. The only legal user packets which can be received while in this state are *data* and *data_end*. If the packet transfer module receives an unexpected packet while in this state, it will send a 'mail_close_req' to the mailbox control module, an *ul_error_resp* packet to the user, and then return to the UL/DL_CMD_WAIT state. If the user becomes disconnected while the module is in the UL_DATA_RX state, it will send a 'mail_close_req' message to the mailbox control module and then return to the UL/DL_UNINIT state.

TABLE 7.5 STATE TRANSITIONS FROM UL_DATA_RX

Received Packet or Message	**Message Sent**	**Next State**
data	mail_recv	WAIT_MAILBOX
data_end	mail_close_req	WAIT_MAILBOX
other packets	mail_close_req, *ul_error_resp* Packet	UL/DL_CMD_WAIT
disconnect	mail_close_req	UL/DL_UNINIT

5. UL_ABORT

The UL_ABORT state is entered whenever a problem is found with an ongoing upload prior to receipt of the *data_end* packet. While the packet transfer module is in the UL_ABORT state, all *data* packets are discarded. It will remain in this state until a *data_end* packet is received, an unexpected packet is received, or the user is disconnected. The state transitions out of UL_ABORT are summarized by Table 7.6.

TABLE 7.6 STATE TRANSITIONS FROM UL_ABORT

Received Packet or Message	**Packet Sent**	**Next State**
data	none	UL_ABORT
data_end	none	UL/DL_CMD_WAIT
other packets	*ul_error_resp*	UL/DL_CMD_WAIT
disconnect		UL/DL_UNINIT

6. DL_FILE_DATA

The state DL_FILE_DATA is entered from the UL/DL_CMD_WAIT state whenever a properly formatted *download_cmd* is received from the user. If a badly formatted *download_cmd* is received, the user will be sent a *dl_error_resp* and the packet transfer module will remain in the UL/DL_CMD_WAIT state.

Just prior to entering the DL_FILE_DATA state, the packet transfer module sends a 'mail_req' message to the mailbox control module. While in the DL_FILE_DATA state, the packet transfer module simply waits for 'mail_resp' messages from the mailbox module containing file data to be transmitted to the user. As each piece of the file arrives, it is sent on to the user in a *data* packet. When the mailbox control module indicates that the last byte of the file has been provided, a *data_end* packet is sent to the user. The packet transfer module remains in the DL_FILE_DATA state until either a *dl_ack_cmd* or a *dl_nak_cmd* is received from the ground. Then it returns to the UL/DL_CMD_WAIT state. The satellite takes no particular action upon receipt of a *dl_nak_cmd*. It will be the responsibility of the ground station to request a new download of the same file at a future time if the user so desires. If the file number requested for download does not exist, a *dl_error_resp* packet will be transmitted and the module will return immediately to the UL/DL_CMD_WAIT state. The state transitions from DL_FILE_DATA are shown in Table 7.7.

TABLE 7.7 STATE TRANSITIONS FROM DL_FILE_DATA

Received Message or Packet	**Packet Sent**	**Next State**
mail_resp, error	*dl_error_resp*	UL/DL_CMD_WAIT
mail_resp, *no error*	*data*	DL_FILE_DATA
mail_resp, end of file	*data_end*	DL_FILE_DATA
dl_ack_cmd	*dl_completed_resp* dl_ack message	UL/DL_CMD_WAIT
dl_nak_cmd	*dl_aborted_resp*	UL/DL_CMD_WAIT
other Packets	*dl_error_resp*	UL/DL_CMD_WAIT
disconnect		UL/DL_UNINIT

VIII. MAILBOX CONTROL MODULE

A. FUNCTION

The primary role of the MAILBOX_CONTROL module is to keep track of the mail files which have been uploaded to PANSAT from users on the ground. It also keeps track of user-accessible telemetry files which have been prepared by the TELEMETRY_GATHER module for downloading to interested users, and "bulletins" which have been posted by the ground control station for the information of all PANSAT clients. The users' active selection lists are also maintained by the mailbox control module.

The mailbox control module has only one state, WAIT. This state name exemplifies the method employed by the module to carry out its duties. It "waits" until it receives a request for information or a packet of file data from the packet transfer module, or is notified of a file posted by the telemetry module or by the ground control module. Most housekeeping functions within the "mailbox" are triggered by receipt of these messages. The mailbox control module responds to the received message, carries out any necessary activity, and then continues waiting until the next message arrives. A few administrative functions, such as purging all mail, must be directed by special command messages from the ground control or auto control modules.

B. SOURCE RECORDS

The method employed by the mailbox control module to keep track of all uploaded files and all users' active selection lists is a linked list of Source_Records. The Source_Record type is a data structure which contains information which links every stored file with the source from which it was originally uploaded, as well as an active selection list for any source (user) that has requested one. The fields of the Source_Record are listed in Table 8.1, along with the function of each field.

TABLE 8.1 FIELDS OF THE SOURCE RECORD

Field	Type	Function
source_num	uint	Contains a unique integer assigned to each ground user who has uploaded any files currently stored onboard the satellite or has an active selection list. Used as the first 2 bytes in the file numbers assigned to each file uploaded by this user.
call	Callsign_Type	The call sign belonging to the client assigned the above 'source_num'. The call sign will be used as the DOS file name for all files uploaded by this client. Each file will be assigned an extension from "001" to "999".
selected	Select_List	The Select_List structure includes the fields 'num_sel', a uint indicating the number of files in the client's selection list, and 'sel', a variable length array of the mail numbers of those files. 'num_sel' must be $<=$ *max_mail*, the maximum number of files allowed in one selection list. A 'num_sel' equal to '0' indicates "no active selection list".

TABLE 8.1 FIELDS OF THE SOURCE RECORD

Field	Type	Function
next_mail	uint	The index into the 'sel' array which marks the "next" mail file in the selection list not yet downloaded by the client. When 'next_mail' becomes >= 'num_sel', the selection list is "empty" if another request to download the "next" file arrives.
next_dir	uint	The index into the 'sel' array which marks the "next" mail file in the selection list for which a directory entry has not yet been downloaded by the client. When 'next_dir' becomes >= 'num_sel', the selection list is "empty" if another request to download directories for the "next" 10 files arrives. When both 'next_mail' and 'next_dir' are >= 'num_sel', 'num_sel' reverts to '0' and the client no longer has an active selection list.
next_ext	File_Ext = 000..999	The next file extension to be used on a file uploaded by this client. In binary form, the 'ext' is used as the last 2 bytes in the file number assigned to the file. In ascii form, it forms the 3 character DOS file extension.
num_act	uchar	The number of files uploaded by this client which are still being stored aboard the satellite.
next_num	^Source_Record (pointer to Source_Record)	Pointer to the next Source_Record, numerically by 'source_num'.
next_call	^Source_Record	Pointer to the next Source_Record, alphabetically by 'call'.

C. RESPONSE TO MESSAGES FROM THE PACKET TRANSFER MODULE

By far the greatest number of messages received by the mailbox control module originate from the packet transfer module, via the Mailbox_Access Channel. Chapter VII lists many transitions of the packet transfer module to the WAIT_MAILBOX state. These transitions indicate that the packet transfer module has requested information from the mailbox control module and is awaiting a reply. The packet transfer module also sends messages to the mailbox control module while remaining in the UL_DATA_RX state. The activities of the mailbox control module triggered by each message type from the packet transfer module, and the required reply messages, are addressed in the following subsections.

1. The 'active_sl_req' and 'active_sl_resp' Messages

When the packet transfer module sends an 'active_sl_req' message, it is inquiring whether there is an active selection list for a particular user. The mailbox control module must check the source records to see if the user has an active selection list or not. An 'active_sl_resp' message is returned to the packet module, indicating **true** if the user does have an active list, and **false** otherwise.

2. The 'mail_num_req' and 'mail_num_resp' Messages

A 'mail_num_req' message indicates that a user wants to upload a file. If this is a new file, a file number is required for it. If it is an upload continuation, the current file offset is needed. The mailbox module must ensure that there is enough room in memory to store a file of the length indicated in the message. If the indicated file number is '0x00000000', the mailbox will get the user's source number and next extension from the source records (or assign a new source number if necessary) and form a new file number. If the file number in the 'mail_num_req' message is not '0x00000000', the mailbox module will find the current length of it's partial copy of the file. In the 'mail_num_resp' message, the mailbox module will supply the packet module with the required file number or offset for the upload, or indicate that an error has occurred (such as insufficient space or incorrect file number).

3. The 'mail_recv' and 'mail_recv_resp' Messages

The 'mail_recv' message passes file data which has been received from a user to the mailbox module. The data must be appended to the appropriate file. The mailbox module attempts to find and open the file to which the data belongs and append it. The 'mail_recv_resp' will indicate whether the data has been stored successfully or whether an error has occurred.

4. The 'mail_close_req' and 'mail_close_resp' Messages

The 'mail_close_req' message can indicate one of two situations. Either a *data_end* packet has arrived, indicating that an upload has been completed, or an upload has been interrupted due to user disconnect or an unexpected packet. If an upload has been completed, the packet module will indicate this by setting the 'req_resp' parameter of the message to true, requesting a response. In this case, the mailbox module will check the integrity of the uploaded file and report the results in the 'mail_close_resp' message. If 'req_resp' is set to false, the upload has been interrupted and the mailbox module will simply close the file and wait for the upload to be continued at a later time.

5. The 'mselect_req' and 'mselect_resp' Messages

An 'mselect_req' message forwards to the mailbox module the Select_Structure of a client requesting to form a new active selection list. The mailbox module must parse the Select_Structure and either prepare the default selection list or evaluate the selection equation with respect to each file in the mail box. The file number of each matching (or default) file will be placed in the client's selection list. There is a maximum number of file numbers which can be placed in any selection list. When this number is reached, further selection will be discontinued. The 'mselect_resp' message indicates how many files have been placed in the selection list, or if an error has occurred. Any prior existing list will be discarded.

6. The 'mail_req' and 'mail_resp' Messages

The packet transfer module sends a 'mail_req' message in order to obtain file data for download to a client. A file number and offset will be included in the message. If the "next" file in the selection list is requested, the indicated file number will be

'0x00000000', and the mailbox module must consult the client's source record to determine the actual file number of the next file in the list. The offset for the "next" file will always be zero. When the next data set from that file is requested, the file number and appropriate non-zero offset will be known, and included in the 'mail_req' message. The mailbox module will begin at the appropriate file offset and begin copying bytes into the data buffer. It will copy either the number of bytes which will fit into one packet, or the remaining bytes in the file, whichever is less. Either the data buffer or an error indication will be sent back to the packet module in the 'mail_resp' message. When the end of a file has been sent to the packet module, the mailbox module responds to the next 'mail_req' with an empty data buffer and no error code. This indicates to the packet buffer that it is time to send the *data_end* packet.

7. The 'dl_ack' Message

The packet transfer module will send a 'dl_ack' message to the mailbox control module after receiving a *dl_ack_cmd* packet from the user. Only if a 'dl_ack' message is received will the mailbox module change the 'next_mail' field in the user's source record. The 'next_mail' pointer is only advanced after the file it indicates has been successfully downloaded to the client. The number of the file acknowledged will be included in the 'dl_ack' message along with the client's call sign. The file number must match that indicated by the 'next_mail' field of the client's source record for the field to be updated.

8. The 'dir_req' and 'directory' Messages

The 'dir_req' message requests directory information for either the file number indicated, or the "next" ten files in the client's active selection list. Directory information for a file is simply a copy of the PANSAT file header. If a file number is indicated, the mailbox module places a copy of the appropriate header in the data buffer which is send back with the 'directory' message. If the "next" 10 entries are requested, the mailbox module consults the source record to determine whether there is an active list, and if so, which is the "next" file for which a directory entry has not yet been sent. The headers are copied for the next 10 files on the list, beginning with the one marked by 'next_dir'. If there are less than 10 remaining on the list, they are all sent. There is no download acknowledge associated with directories, and the 'next_dir' counter is automatically advanced when the 'directory' message is sent back to the packet transfer module.

9. The 'mail_del_req' and 'mail_del_resp' Messages

The packet transfer module sends a 'mail_del_req' when a user wishes to delete a file from the satellite's mailbox. The mailbox module must first ensure that the user in question is authorized to delete the indicated file. A user may only delete a file which they have uploaded, or one which is addressed to them as the sole recipient. The mailbox module knows who uploaded the file, since the file name is the same as the source call sign. It can consult the destination fields of the file header to determine whether the requesting user is the sole recipient. If the deletion is authorized, it will be carried out, and a *no_error* indication returned to the packet module in the 'mail_del_resp' message. Otherwise, the *er_permission_denied* code will be returned.

D. RESPONSE TO MESSAGES FROM OTHER MODULES

The mailbox control module may also be tasked to respond to messages from modules other than the packet transfer module. These messages may come via one of the Mailbox_Admin_Channels or via the Telemetry_Storage_Channel. The latter channel is connected to the TELEMETRY_GATHER module, while one copy of the former is connected to the AUTO_CONTROL module and another is connected to the GROUND_CONTROL module. None of these three modules has been completely specified, and the requirements for them are still evolving. Some possible functions for them have been suggested, and those which impact upon the mailbox control module will be discussed in the following subsections.

1. The 'list_mail' and 'mail_list' Messages

The NPS ground control station personnel retain the right to inspect all messages in the mailbox, regardless of the upload sources or the addressees. The ground control station, when it is so desired, can request a list of all files currently maintained in the memory, or a partial list of only those files "from" or "to" a particular call sign. This command is received by the GROUND_CONTROL module, which responds .

requesting the appropriate file list from the mailbox control module using a 'list_mail' message. The mailbox module responds with a 'mail_list' message which indicates the number of files matching the criteria of the 'list_mail' message and provides a list of all of the appropriate file numbers. From this list, the ground control module or the ground control station personnel can then choose files to download.

2. The 'post_bulletin' and 'delete_bulletin' Messages

The ground control module has the same access to the file handling facilities of the Space Craft Operating System as does the mailbox control module. For this reason, it does not need to go through the mailbox module in order to "post" a bulletin, which really consists only of storing a file with the name "BULLETIN.xxx" in the mail storage area. (File lists such as those discussed in the previous subsection are requested from the mailbox module merely to take advantage of its enhanced association capabilities using the source records it maintains.) The mailbox module should, however, maintain a complete set of source records, including one for the ground control station. When a bulletin is posted, the ground control module informs the mailbox module using a 'post_bulletin' message, so that an appropriate file number can be assigned and the source record can be updated. Similarly, when a bulletin is deleted, the mailbox module is informed by a 'delete_bulletin' message.

3. The 'full_mailbox' and 'purge_mail' Messages

Whenever a user requests to upload a file, the mailbox module must first determine whether there is room for it in the memory. If it finds that there is not enough room, it does some "house-cleaning", deleting all files which have passed their expiration dates. This is the only time the mailbox module deletes files on its own, so that many files may actually remain onboard the satellite for longer than the nominal time allowed. After the mailbox module has deleted all files which have expired, it once again checks to see if there is enough room to upload the new file as requested by the user. If there is still not enough room, the mailbox module must deny the request to upload. At the same time, it informs the AUTO_CONTROL module of the problem with a 'full_mailbox' message.

Perhaps in response to a 'full_mailbox' message, or perhaps in obedience to a ground station command, or for some other pressing reason, the ground control or auto control module can direct the mailbox module to "purge" the mail box. The 'purge_mail' message will indicate whether all mail files should be deleted, or all files posted prior to some designated upload time, or all files "from" or "to" a particular call sign. This purge is done via the mailbox module, so that it will have the chance to update all affected source records.

4. The 'store_user_telem' and 'delete_user_telem' Messages

Like the ground control and auto control modules, the TELEMETRY_GATHER module also has complete access to the SCOS file management capabilities. When user-accessible telemetry data is to be posted, it merely saves a file called "USRTELEM.xxx" in the mail storage area. These telemetry files can also be deleted by the telemetry module when they become outdated. In the interest of maintaining a complete set of source records, the mailbox control module is informed of these actions via the 'store_user_telem' and 'delete_user_telem' messages.

IX. REMAINING MODULES

A. TELEMETRY GATHERING MODULE

In the current PANSAT flight software specification, 14 separate software modules have been defined at the module header definition level. Of these, detailed module body definitions have been developed for 4. The DATA_TRANSFER, PACKET_TRANSFER, and MAILBOX_CONTROL modules are described in Chapters V through VIII of this thesis. A preliminary module body definition for the PASSWORD_CONTROL module has been written, but will not be released to the general public. Two modules, PRIMITIVE_AX25 and PRIMITIVE_SW_LOADER, are actually commercial software products, BAX and PHTX. The capabilities of these programs will be accessed by various PANSAT modules, but no body definitions will be written for them, because there is no need to specify existing software, only the interfaces to it. The body definitions of the remaining 8 modules will be highly dependant upon the actual hardware configuration of the satellite, which is still undergoing daily design changes. Central to the operation of these remaining modules will be the operation of the TELEMETRY_GATHER module.

The function of the telemetry gathering module is to collect data on the operation of the satellite from which control decisions can be made, both by the automatic control module (AUTO_CONTROL) and the ground control station personnel at NPS. In order to obtain much of this data, the telemetry gathering module has direct control over the

A/D_DRIVER module which operates the analog-to-digital converters and associated multiplexors in order to obtain relevant sensor data, such as battery voltages or solar array temperatures. Other telemetry information will come from the BAX and SCOS software, which maintain various statistics about the communications and operating environments.

The hardware telemetry points which have been defined thus far are listed in Table 9.1. The best situation is for each point to stay within the expected or "nominal" range. When a reading goes outside the nominal range, there is still no serious system degradation unless it also goes outside the "operating range". At this point, there may be no immediate danger to the system, but a trend may have started which will soon lead to operational difficulties. When a reading goes outside the "red alert" range, immediate correctional actions must be initiated, if they have not been already. System failure could be imminent. Many of the exact values for these ranges have not yet been determined. The proper preventive and/or correctional steps to be taken in each situation are also still under study. The values contained in Table 9.1 are the best estimates available at this time, but are subject to change. Those readings for which no estimated values have yet been determined are marked with "tbd". The "totals" listed are for the sensors controlled by one Digital Control System (DCS) board, on which will be running one copy of the flight software. The current design calls for the entire DCS to be duplicated, and for each board to be attached to its own complete and separate set of sensors.

TABLE 9.1 HARDWARE TELEMETRY POINTS

Point	Nominal		Operating		Red Alert	
	Min.	Max.	Min.	Max.	Min.	Max.
Solar Array Temperatures (17 total)	0° C	50° C	-20° C	120° C	-30° C	140° C
Battery Voltages (2 total)	12 V	13 V	11.5 V	13.5 V	10 V	15 V
Battery Temperatures (4 total)	-1.1° C	10° C	-6.7° C	26.7° C	-15° C	50° C
Battery Discharge Currents (2 total)	tbd	tbd	tbd	tbd	tbd	tbd
Electrical Power System (EPS) Bus Voltage (1 total)	tbd	tbd	tbd	tbd	tbd	tbd
EPS Board Temperature (2 total)	0° C	40° C	-10° C	50° C	tbd	tbd
Transmitter Current (1 total)	tbd	tbd	tbd	tbd	tbd	tbd
Transmitter RF Power (2 total)	tbd	tbd	tbd	tbd	tbd	tbd
Transmitter Temperature (2 total)	0° C	40° C	-10° C	50° C	tbd	tbd
Received Signal Strength (2 total)	tbd	tbd	tbd	tbd	tbd	tbd
Receiver Temperature (2 total)	0° C	40° C	-10° C	50° C	tbd	tbd
Sense Relays for State of Communications Hardware (total tbd)	tbd	tbd	tbd	tbd	tbd	tbd
DCS Board Temperature (2 total)	0° C	40° C	-10° C	50° C	tbd	tbd

The telemetry gathering module maintains a list of sensor points with timing intervals and expected operating ranges for each. This list can be updated by commands from the ground control station, which can cause points to be added or deleted, or can change the timing intervals for obtaining readings from various points. Some timing intervals may be changed dynamically by the automatic control module or the telemetry gathering module itself, based upon trends in the readings or upon reading which are out of the expected ranges.

Table 9.2 lists some "operating environment telemetry points" which can be gathered by SCOS, and passed to the telemetry gathering module for inclusion in the telemetry files. Table 9.3 lists some "communications environment telemetry points" which can be gathered by BAX, and may be of interest to the ground station controllers. BAX has the capability to maintain a file of this data itself and to download it directly to the ground control station. Whether the information will be passed to the telemetry gathering module to be included with the rest of the telemetry, or whether this separate "BAX telemetry" file will be maintained and passed to the ground control station as the result of a separate ground station command, has not yet been determined. Table 9.4 contains a list of other general system data which may be collected by the telemetry gathering module directly from the satellite hardware or from the other software modules. In some cases, such as the data points listed under "LOGIN" and "MAILBOX", existing module specifications will have to be modified in order to require the software to gather the data required by the telemetry module. Such modifications will be postponed until it has been decided which of these data points will be of most interest to the ground station controllers, and what sampling intervals will be required.

TABLE 9.2 SCOS TELEMETRY POINTS

Data Point	Data Type	Description
Scheduler Events	List of numbers	Operating System multi-tasking events (tasks running & scheduled).
Timer Events	List of numbers	Operating System tasks in queue.
SCOS Service Calls	List of numbers	General Operating System information.

TABLE 9.3 BAX TELEMETRY POINTS

BAX Data Point	Data Type	Description
smallct	uint	Count of received frames containing < 32 bits.
nonint	uint	Count of received frames with a bit length not evenly divisible by 8
bigcnt	uint	Count of received frames that exceed maximum size.
abortcnt	uint	Count of received frames that are aborted.
overcnt	uint	Count of receiver overruns.
crc	uint	Count of receiver crc errors.
tx_aborted	uint	Count of transmitted frames aborted or flushed.
tx_under	uint	Count of transmitted frame underruns.
tx_abort_call	uint	Count of calls to qio_abort/flush.
qiocurrx	uint	Current number of frames in receiver queue.
qiomaxrx	uint	Maximum number of frames in receiver queue.

TABLE 9.3 BAX TELEMETRY POINTS		
BAX Data Point	**Data Type**	**Description**
qiocurtx	uint	Current number of frames in transmitter queue.
qiomaxtx	uint	Maximum number of frames in transmitter queue.
poolfail	uint	Number of "pool gets" that failed.
retry_exceeded	uint	Count of times the maximum number of frame retires has been exceeded.
quitottx	ulong	Total number of transmitted frames.
quitotrx	ulong	Total number of frames received with no errors.
tdatain	ulong	Total number of data bytes received.
tdataout	ulong	Total number of data bytes transmitted.
tdigi	ulong	Total number of digipeated frames.
daytime	ulong	Total number of 50msec intervals that have expired since system startup.
start_time	ulong	Startup time in seconds. (UTC)
Following are counts of frames types defined in the AX.25 protocol [Ref. 2].		
<I> in	ulong	Number of "information" frames received.
<RR> in	ulong	Number of "receive ready" frames received.
<RNR> in	ulong	Number of "receive not ready" frames received.
<REJ> in	ulong	Number of "reject" frames received.
<DM> in	ulong	Number of "disconnect mode" frames received.
<SABM> in	ulong	Number of "set asynchronous balanced mode" (connect request) frames received.
<DISC> in	ulong	Number of "disconnect request" frames received.
<UA> in	ulong	Number of "unnumbered acknowledge" frames received.
<FRMR> in	ulong	Number of "frame reject" frames received.

TABLE 9.3 BAX TELEMETRY POINTS		
BAX Data Point	**Data Type**	**Description**
<INV> in	ulong	Number of "invalid" frames received.
<UI> in	ulong	Number of "unnumbered information" frames received.
<I> out	ulong	Number of "information" frames transmitted.
<RR> out	ulong	Number of "receive ready" frames transmitted.
<RNR> out	ulong	Number of "receive not ready" frames transmitted.
<REJ> out	ulong	Number of "reject" frames transmitted.
<DM> out	ulong	Number of "disconnect mode" frames transmitted.
<SABM>out	ulong	Number of "set asynchronous balanced mode" frames transmitted.
<DISC> out	ulong	Number of "disconnect request" frames transmitted.
<UA> out	ulong	Number of "unnumbered acknowledge" frames transmitted.
<FRMR>out	ulong	Number of "frame reject" frames transmitted.
<INV> out	ulong	Number of "invalid" frames transmitted.
<UI> out	ulong	Number of "unnumbered information" frames transmitted.

TABLE 9.4 GENERAL SYSTEM TELEMETRY POINTS		
Data Point	**Data Type**	**Description**
	LOGIN Data	
Logins	uint	Number of user logins.
Logouts	uint	Number of user logouts (requested disconnects).
ALogins	uint	Number of authorized logins.

TABLE 9.4 GENERAL SYSTEM TELEMETRY POINTS

Data Point	Data Type	Description
UALogin	uint	Number of unauthorized login attempts.
UALtime	UTC	Unauthorized login attempt time stamp.
Uuser	array of Callsign_ Type	List of Undesirable Users.
MAILBOX Data		
RMail	uint	Count of received mail.
SMail	uint	Count of sent mail.
StMail	uint	Count of stored mail.
Stor	ulong	Amount of storage used.
Communication System Data		
Receiver	boolean	Receiver A or B selected.
Transmitter	boolean	Transmitter A or B selected.
Mode	boolean	Spread Spectrum turned On or Off.
Atten	uint	Attenuation Level 1 through 8 selected.
Digital Control System Data		
DCS	boolean	DCS A or B selected.
SWver	uint	Software version in use.
date	UTC	Current satellite date and time.
SEUc	uint	EDAC (error detection and correction) SEU (single event upset) count.
SEUt	UTC	Start time for EDAC SEU time.
SEUlt	UTC	Time of latest EDAC SEU.
RAMw	ulong	Address of next RAM cell to be "washed".

As the telemetry gathering module completes each round of readings, it updates a file of the "current telemetry" which is accessible to the automatic control module. The automatic control module makes use of this data in its autonomous control of the satellite hardware systems. The telemetry gathering module also stores telemetry data in a telemetry history file, which will continue to grow and store past data until it is purged by a command from the ground control station, or it reaches a pre-determined maximum size. If the maximum file size is exceeded before the file can be downloaded and then purged by the ground control station, the most recent entries for each data point will be maintained, and older entries deleted, in order to control the size of the file. The ground control station will use this larger telemetry file to analyze trends in satellite performance, and to make control decisions beyond the scope of those made by the automatic control module. The telemetry gathering module will also maintain shorter telemetry files containing data which may be of interest to the amateur radio users who access the satellite mail box system. These files are stored in the mail area with the file name "USRTELEM.xxx".

B. AUTOMATIC CONTROL MODULE

The AUTO_CONTROL module carries out periodic functions, such as battery conditioning, on a time scheduled basis. It also carries out aperiodic functions. As indicated in the previous section, the AUTO_CONTROL module makes use of the data collected by the telemetry gathering module to make decisions about the control of the satellite hardware. It also maintains a "time_tagged" command buffer which lists activities which should take place at a particular time in the future. This command buffer is updated by the GROUND_CONTROL module as a result of ground control station commands. The Digital Control System design includes hardware timers which can be programmed by the automatic control module to interrupt the microprocessor at designated time intervals to initiate periodic events or to produce a set of one-time-only interrupts to initiate events controlled by the command buffer. Software timers may also be used for some of the automatic control module functions.

Most of the control functions carried out by the automatic control module will likely be based on a "table look-up" system. When a timer interrupt occurs, an interrupt vector table will contain the address of the appropriate subroutine needed to carry out the scheduled activity. Another table of subroutine addresses will be indexed based on combinations of telemetry readings which call for some action to be taken. These subroutines and tables will be developed as more is learned about the specific requirements of the hardware as it is designed.

Table 9.5 lists some possible functions of the automatic control module which have been identified thus far. The EPS_DRIVER, COMM_DRIVER and DCS_DRIVER modules contain the software drivers required for direct digital control of the electric power system, communications, and digital control system hardware. The services of these modules will be accessed as necessary by the automatic control module in order to carry out functions listed in Table 9.5.

TABLE 9.5 AUTOMATIC CONTROL MODULE FUNCTIONS

Function	Description
Electric Power Supply Control	Turn hardware components off and on as necessary to conserve power, allow battery conditioning, etc.
Condition Batteries	Periodically discharge and recharge batteries in order to prevent battery "memory".
Systems Test Management	Carry out periodic systems checks in addition to normal telemetry gathering. Save test data for download to ground control station.
Communications Control	Transmitter/Receiver component select.
Transmitter Output Power Control	Set level of transmitter power.
Automatic Subsystem Select	Select alternate subsystem upon time-out waiting for response of a primary subsystem.
Real Time Clock Control	Set and remove times for periodic interrupts.
Send Messages	Send periodic messages to the ground control station via BAX.
Copy Vital Statistics	Transfer vital operating system information to alternate processor.
RAM Wash	Periodic reading/writing of system RAM to enable Error Detection and Correction functions.
Digital Control System Health Check	Periodic signal to EPS to ensure proper operation of active DCS. EPS will disable a malfunctioning DCS board and "boot" the alternate when the proper signal is not received on time.

TABLE 9.5 AUTOMATIC CONTROL MODULE FUNCTIONS

Function	**Description**
User Lockout.	Message to data transfer module locking out all or new users.

C. GROUND CONTROL MODULE

The GROUND_CONTROL module contains the command interpreter and the functionality required to carry out commands transmitted by the ground control station at NPS. Ground control packets will be passed directly to the ground control module by BAX, since they will be addressed specifically to the ssid (subsystem identification number) for this module. All ground station commands will be subject to verification by including a time varying password. The PASSWORD_CONTROL module will keep track of the current password aboard the satellite, and will provide this information as necessary to the ground control module. Similar software will track the current password for the ground control station. There will be facilities for determining the current password aboard the satellite, in case the two systems lose synchronization for any reason. The specification of the password control module contains proprietary information, and will not be published for general release.

Once a command has been received from the ground control station, the password has been verified, and the command has been interpreted, the ground control module either carries out the command directly, or communicates with other software modules as necessary to utilize their capabilities. A ground command may involve updating the time-tagged command list of the automatic control module, or varying the time intervals for periodic events carried out by the automatic control or telemetry gathering modules. It may initiate a one-time-only corrective action, or change a basic system parameter. Some ground station commands simply involve the acquisition of information for use by the ground control station software or personnel.

Some possible functions of the ground control module which have been identified thus far are listed in Table 9.6. The PRIMITIVE_SW_LOADER module, which is actually the commercial program "PHTX", is designed to work directly with BAX to upload software. This module will be utilized by the ground control module when a command is received to upload new software. In this way, the flight software can be updated as necessary to correct errors or increase functionality.

TABLE 9.6 FUNCTIONS OF THE GROUND CONTRC . MODULE

Function	Description
Command Interpretation/Validation.	Process a received ground station command.
Update Time-Tagged Command Buffer.	Schedule events to be carried out at a future time by the automatic control module, or delete events from the command buffer.
Set Control Rates.	Update time intervals or list of periodic functions of the automatic control module.
Set Telemetry Polling Rates.	Update time intervals used by the telemetry gathering module for particular telemetry points. Add or delete telemetry points.

TABLE 9.6 FUNCTIONS OF THE GROUND CONTROL MODULE	
Function	**Description**
Software Upload.	Upload, and store new or updated software modules.
Run Software.	Begin using newly uploaded or alternate software module.
Delete Software.	Delete specified software module.
Copy Software.	Copy verified software to alternate processor.
Boot ROM.	Reboot PANSAT from ROM (read only memory).
Boot OS.	Load a new operating system and transfer con
Read OS Information.	Download the current operating system pointers and parameters.
List Mail.	Download a list of all mail messages and bulletins currently stored.
Dump Mail.	Download system bulletins and mail in bulk.
Post Bulletin.	Post a system bulletin in the mailbox area for all users.
Remove Bulletin.	Remove a system bulletin.
Purge Mail.	Purge all or selected mail from the mailbox storage.
Read Current Telemetry.	Download the current telemetry file maintained by the telemetry gathering module.
Read Stored Telemetry.	Download the telemetry history file maintained by the telemetry gathering module.
Purge Stored Telemetry.	Delete all or portions of the telemetry history file.
Read Data.	Download an arbitrary block of data, specified by address pointer, from the file storage area or system RAM.
Set Real Time Clock.	Set satellite's real time clock to a specified time.
Read Real Time Clock.	Download current time on satellite's real time clock.

TABLE 9.6 FUNCTIONS OF THE GROUND CONTROL MODULE

Function	Description
Subsystem Power Control.	Turn power on/off to a particular subsystem.
Condition Battery.	Discharge/Recharge specified battery.
Trickle Charge Battery.	Trickle charge specified battery.
Charge Battery.	Quick charge of specified battery.
Select Battery.	Select specified redundant battery.
Select Receiver.	Select specified redundant receiver.
Select Transmitter.	Select specified redundant transmitter.
Select Processor.	Select specified redundant digital control system board.
Set Mode.	Select communications mode: spread spectrum or BPSK.
Set Maximum Transmitter Power.	Set maximum allowable amplitude of transmitter power.
Set Attenuation.	Set attenuation level of the active transmitter.
Switch to Super User Mode	Functions requiring super user mode are tbd.
Exit Super User Mode.	
Read Event Log.	Download Event Log maintained by the event logging module.
Purge Event Log.	Delete all or portions of the event log,.
Read Time-Tagged Command Buffer.	Download the time-tagged command buffer.
Purge Time-Tagged Command Buffer.	Delete the entire time-tagged command buffer.
User Lockout.	Message to data transfer module locking out all or new users.

D. EVENT LOGGING MODULE.

The purpose of the EVENT_LOGGING module is to maintain a history of all the significant events which happen and commands which are carried out aboard the satellite. It is hoped that this event log will be helpful in trouble shooting problems aboard the satellite, or merely in studying its operation. The event logging module differs from the telemetry gathering module in one major respect. The telemetry gathering module periodically polls the hardware and other software modules, gathering a predetermined list of specified data. The event logging module waits to receive event messages from other modules, informing it of aperiodic events which are deemed significant in some way.

A list of "significant" events will need to be determined, so that the exact nature of the event messages can be defined in the software specification. Some possible events include the occasion of a full mailbox, a telemetry reading beyond the "operating range" (this will also be listed in the telemetry files, of course, but may stand out more here, or be associated with some other event which will make trouble shooting and correction easier), user connections lost because the transmitter has been shut down for power reasons, etc. An event log entry will also be made each time an automatic command function or a ground control command is carried out. The exact format of the event log entries will be developed as the list of significant events and useful information is further defined.

X. CONCLUSIONS AND RECOMMENDATIONS

A. THE USE OF ESTELLE

The formal description technique, Estelle, has proven to be a valuable tool in creating a software specification. Its methods of defining state machine behavior and its channel and message definitions have provided a unique way of visualizing a system, and seeing how all of the pieces fit together. The various levels of abstraction greatly facilitate the advancement of a project, even when all details are not yet known. When details are known, Estelle provides ample means of specification at the lowest possible levels, and the flexibility to define algorithms both simple and complex.

In order to make Estelle even more useful in this project, a few modifications have been made to it. For instance, since "C" has already been chosen as the implementation language, a few data types have been defined to more closely match familiar structures in "C". Array indices start at 0 in this specification, as they do in "C". Multiple dimension arrays are indexed by multiple sets of brackets, "var[i][j]", rather than by multiple indices within one set of brackets, "var[i, j]". The names of the primitive data types are borrowed from "C": "uchar", "uint "and "ulong". Many of the primitive functions and procedures are functions familiar to "C" programmers. In addition, various font modifications have been used to make elements of the Estelle and Pascal syntax stand out, so that their meanings are more obvious in the context. Bold is used to indicate reserved words, user-defined data types begin with Capital_Letters, constants are written in *italics*, etc.

Many of the more complex capabilities of Estelle are not utilized, since they are somewhat confusing and are not needed to make clear the intended behavior of the software being defined. The greatest drawback of Estelle is the specification of Estelle itself, [Ref. 8]. [Ref. 8] is very difficult to read and sometimes impossible to understand. For those interested in using Estelle in future software specification projects, it is recommended that only the Annexes be read. These contain all the information needed, as well as adequate examples to provide understanding of how this language can actually be used.

B. RECOMMENDATIONS FOR FURTHER WORK

This thesis provides a preliminary specification for the flight software of the Petite Amateur Navy Satellite. As much information as is currently available concerning the high-level operational requirements of the satellite has been included. A software architecture has been provided which defines the individual software modules and their interfaces. Detailed definitions for the bodies of the communications and file transfer protocol modules have been developed.

There is obviously much work remaining to be done. The module body definitions for the telemetry gathering, analog to digital conversion, automatic control, ground control, electronic power system driver, communication driver, digital control system driver, and event logging modules must be developed. The channel types and message interfaces between these remaining modules and between them and the existing modules must be defined in greater detail. Once the complete, detailed specification is available for the entire flight software system, the actual code must be written and tested. The ground software and the bootstrap software must be specified and coded, and the interfaces between these programs and the flight software must be tested. Hardware designs must be completed and tested before any software specifications can actually be finalized. A start has been made, and the beginnings of a road map have been drawn. Much more effort will be required before this project is completed.

(This Page Intentionally Left Blank)

APPENDIX A - ESTELLE SOFTWARE SPECIFICATION

```
specification Flight_Software;

type                                    { Primitive Types                                 }
                                        { Note: the notation '0xhh' is used to refer to   }
                                        {   hexadecimal numbers, with the h's             }
                                        {   representing the hex digits 0..F. The number  }
                                        {   of bytes in each hex number is the number of  }
                                        {   digits divided by 2.                          }
   uchar  = 0x00..0xFF;                 { 8 bits of binary data or 1 byte unsigned integer or }
                                        {   1 ASCII character.                            }
   uint   = 0x0000..0xFFFF;             { 2 byte unsigned integer                         }
   ulong  = 0x00000000..0xFFFFFFFF; { 4 byte unsigned integer                             }
   int    = . .;                        { Positive and Negative integers as defined on    }
                                        {   implementation hardware.                      }

const                                   { 'Global' Constant Declarations                  }
   max_file_length  = any ulong;        { Maximum length of a file onboard the satellite  }
   max_mail         = any uint;         { Maximum pieces of mail in Select list           }
   password_length  = any uchar;           { Number of characters in the password.        }
   max_pdat         = 2047;             { Max number of bytes in the data field of a packet. }
   max_fdat         = 256;              { Maximum length of data field in a BAX frame.    }
   maxlinks         = 30;               { Max channels allowed by BAX.                    }
   pansat_call      = any Callsign_Type;   { Pansat's call sign                           }
   nps_call         = any Callsign_Type;   { NPS' call sign.                              }

   no_error                     = 0x00;  { Error Codes                                    }
   er_ill_formed_cmd            = 0x01;  { Incorrect or unexpected command                }
   er_bad_continue              = 0x02;
   er_server_fsys               = 0x03;
   er_no_such_file_number       = 0x04;
   er_selection_empty           = 0x05;
   er_mandatory_field_missing   = 0x06;
   er_no_pfh                    = 0x07;
   er_poorly_formed_sel         = 0x08;
   er_already_locked            = 0x09;
   er_no_such_destination       = 0x0A;
   er_file_complete             = 0x0C;  { 0x0B was a repeat of 0x05                      }
   er_no_room                   = 0x0D;
```

```
er_bad_header           = 0x0E;
er_header_check         = 0x0F;
er_body_check           = 0x10;
er_permission_denied    = 0x90;  { PANSAT specific; not in FTL0.                  }

type                               { Global Type Declarations                      }
  Tpoint_Record      = . .;        { . . = 'To be determined'                      }
  Telem_Data_Type    = . .;
  EPS_Cmd_Type       = . .;
  EPS_Resp_Type      = . .;
  Comm_Cmd_Type      = . .;
  Comm_Resp_Type     = . .;
  DCS_Cmd_Type       = . .;
  DCS_Resp_Type      = . .;
  Sat_Cmd_Type       = . .;
  Sat_Resp_Type      = . .;
  Event_Report_Type  = . .;
  Callsign_Type    = array[6] of uchar;
  Byte_String   = " array[ ] of uchar ";    { Any even number of hex digits surrounded }
                                   {     by double quotes.  Refers to raw binary   }
                                   {     data matching the pattern of the hex digits}
                                   {     and of the same length.                   }
  Pdat_Len           = 0..max_pdat;  { Number of bytes of info in a packet.        }
  Fdat_Len           = 0..max_fdat;  { Number of bytes of info in a frame.         }
  Lockout_Type       = (all, new);   { Each member of an enumerated type is assumed}
                                   {    to be associated with a distinct uchar.    }
  Frame_Type         = (qat_data, qat_state, qat_ui);
  Link_State         = (qas_connect_pend, qas_connected,
                        qas_connecting, qas_disconnected,
                        qas_disconnecting, qas_framereject);
  Cause              = (qac_local, qac_remote, qac_remotefrmr,
                        qac_timeout);
  Password_Type      = array[password_length] of uchar;
  File_Type          = array[max_file_length] of uchar;
  Pdata              = array[max_pdat] of uchar;  { Info field of a packet         }
  Fdata              = array[max_fdat] of uchar;  { Info field of a frame          }
  Num_Mail           = 0..max_mail;
  Direction          = (left, right);
  Packet_Type        = record
      length_lsb:   uchar;
      hl:           uchar;
      info:         Pdata;
  end;
  Link_Type          = 0 .. maxlinks - 1;
```

```
Name_Type        = array[12] of uchar;  { DOS file name.  Character in 9th
                            {              position must be '.'              }
File_List        = array[ ] of Name_Type;   { Variable len array of file names. }
Bit_Type         = 0..1;
Control_Block    = record       { Includes only BAX fields which are used      }
   link:         uint;          { 'channel' in BAX manual                      }
   kind:         Frame_Type;    { 'type' in BAX manual                         }
   l_state:      Link_State;    { 'state' in BAX manual                        }
   why:          Cause;         { 'cause' in BAX manual                        }
   my_call:      Callsign_Type; { AX25_ADDR or AX25_CALL                       }
   my_ssid:      uchar;         {   in BAX manual                              }
   his_call:     Callsign_Type; { Client's Call sign                           }
   his_ssid:     uchar;         { Client's SSID - may not be needed            }
   t1:           uchar;         { t1 frame ACK/NAK timeout timer value         }
   maxframe:     uchar;         { frame sliding window size                    }
   retry:        uchar;         { maximum number of retries for an out frame   }
   paclen:       uint;          { maximum size of info field in outgoing packet }
end;
```

```
                        { Channel Definitions                              }
channel Abstract_Bax_Channel( Bax_End, Data_Transfer_End);
  by Bax_End:
    qax_input( in_cb: Control_Block; idata: Fdata; datal: Fdat_Len);
  by Data_Transfer_End:
    qax_claim( out_cb: Control_Block; grab: uchar);
    qax_data( link: Link_Type; out_cb: Control_Block; odata: Fdata; datal: Fdat_Len);
    qax_busy( link: Link_Type);
    qax_con_acpt;
    qax_con_rej;
    qax_unbusy( link: Link_Type);
    qax_connect( out_cb: Control_Block);
    qax_ui( link: Link_Type; out_cb: Control_Block; odata: Fdata; datal: Fdat_Len);
    qax_disconnect( link: Link_Type; wait: boolean);

channel Abstract_Packet_Channel( Data_Transfer_End, Packet_Transfer_End);
  by Data_Transfer_End:
    connection( callsign: Callsign_Type);
    disconnect;
    command_packet( command: Packet_Type; datal: Pdat_Len);
  by Packet_Transfer_End:
    response_packet( response: Packet_Type);

channel Mailbox_Access_Channel( Packet_Transfer_End, Mailbox_End);
  by Packet_Transfer_End:
    active_sl_req( client_call: Callsign_Type); { Does client have an active select list?  }
    mail_num_req( client_call: Callsign_Type; mail_number, length: ulong);
    mail_recv( mail_number, offset, length: ulong; mail: Pdata);
    mail_close_req( mail_number, offset: ulong; req_resp: boolean);
    mselect_req( client_call: Callsign_Type; select_struct: Pdata);
    mail_req( client_call: Callsign_Type; mail_number, offset: ulong);
    dl_ack( client_call: Callsign_Type; mail_number: ulong);
    dir_req( client_call: Callsign_Type; mail_number: ulong);
    mail_del_req( client_call: Callsign_Type; mail_number, length: ulong);
  by Mailbox_End:
    active_sl_resp( sl: boolean);     { true if client has an active select list.       }
    mail_num_resp( mail_number, offset: ulong; error_code: uchar);
    mail_recv_resp( error_code: uchar);
    mail_close_resp( error_code: uchar);
    mail_send( mail_number, length: ulong; mail: Pdata);
    mselect_resp( num_sel: Num_Mail; error_code: uchar);
    mail_resp( mail: Pdata; mail_number, length: ulong; no_al: boolean);
    directory( len: Pdat_Len; dir: Pdata; no_al: boolean);
    mail_del_resp( error_code: uchar);
```

```
channel Mailbox_Admin_Channel( Control_End, Mailbox_End);
    by Control_End:
        list_mail( bulletins, messages, from, to: boolean; callsign: Callsign_Type);
        post_bulletin( bulletin: Name_Type);
        delete_bulletin( bulletin_name: Name_Type);
        purge_mail( all, from, to: boolean; callsign: Callsign_Type; post_time: ulong);
    by Mailbox_End:
        mail_list( num_files: uint; mail: File_List);
        full_mailbox;

channel Telemetry_Storage_Channel( Telemetry_End, Mailbox_End);
    by Telemetry_End:
        store_user_telem( telem: Name_Type);
        delete_user_telem( telem_file: Name_Type);

channel Password_Control_Channel( Control_End, Password_End);
    by Control_End:
        password_change_request;
        request_current_password;
    by Password_End:
        password( pswd:Password_Type);

channel Data_Transfer_Control_Channel( Control_End, Data_Transfer_End);
    by Control_End:
        change_params( out_cb: Control_Block);
        lockout( l_kind: Lockout_Type);
        unlock( l_kind: Lockout_Type);
        transmitter( off: boolean);
    by Controlled_End:
        acknowledge;

channel Telemetry_Control_Channel( Control_End, Telemetry_Gather_End);
    by Control_End:
        add_point( point: Tpoint_Record );
        delete_point( point: Tpoint_Record );
        change_timing( point: Tpoint_Record);
        read_current_telem;
        read_stored_telem;
        purge_stored_telem;
    by Telemetry_Gather_End:
        ack_point_change( error: uchar);
        current_telem( telem: Cur_Telem_Type);
        stored_telem( telem: Full_Telem_Type);
```

```
channel A/D_Control_Channel( Command_End, A/D_Converter_End);
    by Command_End:
        warmup(device_num: uchar);
        start_conversion( telem_point: uchar);
        report_data( telem_point: uchar);
    by A/D_Converter_End:
        device_ready( device_num: uchar);
        data_ready( telem_point: uchar);
        telem_data( t_data: Telem_Data_Type);

channel SW_Load_Control_Channel( Control_End, Loader_End);
    by Control_End:
        upload( new_software: Name_Type; sw_address: ulong);
    by Loader_End:
        upload_begin( new_software: Name_Type);
        upload_complete( new_software: Name_Type);

channel EPS_Control_Channel( Control_End, EPS_Driver_End);
    by Control_End:
        eps_cmd( cmd: EPS_Cmd_Type);
    by EPS_Driver_End:
        eps_resp( resp: EPS_Resp_Type);

channel Comm_Control_Channel( Control_End, Comm_Driver_End);
    by Control_End:
        comm_cmd( cmd: Comm_Cmd_Type);
    by Comm_Driver_End:
        comm_resp( resp: Comm_Resp_Type);

channel DCS_Control_Channel( Control_End, DCS_Driver_End);
    by Control_End:
        dcs_cmd( cmd: DCS_Cmd_Type);
    by DSC_Driver_End;
        dcs_resp( resp: DCS_Resp_Type);

channel Satellite_Control_Channel( Ground_Control_End, Auto_Control_End);
    by Ground_Control_End:
        sat_cmd( cmd: Sat_Cmd_Type);
    by Auto_Control_End:
        sat_resp( resp: Sat_Resp_Type);

channel Event_Log_Channel( Event_End, Log_End);
    by Event_End:
        event_report( report: Event_Report_Type);
```

```
                                    { Global function declarations             }
function GET_TIME:    ulong;        { Returns a 32-bit unsigned integer indicating the  }
primitive;                          {   number of seconds since January 1, 1970.  }

function C_BIT_SHIFT( d: Direction; num: uchar; b: uchar):   uchar;
primitive                           { Bit-wise shift of the byte specified by 'b' in the   }
                                    {   direction specified by 'd'.  'num' specifies the   }
                                    {   number of bit positions to shift.  Returns a 1     }
                                    {   byte answer.                                       }

function I_BIT_SHIFT( d: Direction; num: uint; b: uint):   uint;
primitive;                          { Bit-wise shift of the uint specified by 'b' in the   }
                                    {   direction specified by 'd'.  'num' specifies the   }
                                    {   number of bit positions to shift.  Returns a 2     }
                                    {   byte answer.                                       }

function C_BIT_AND( a, b: uchar):   uchar;
primitive;                          { Returns Bit-wise AND of the bytes 'a' and 'b'. }

function I_BIT_AND( a, b: uint):  uint;
primitive;                          { Returns the bit-wise AND of the uints 'a' and 'b'. }

function GET_LSB( number: uint):    uchar; { Receives a 16 bit number and returns the  }
primitive;                          {           least significant 8 bits.              }

function GET_MSB( number: uint):    uchar; { Rceives a 16 bit number and returns the   }
primitive;                          {           most significant 8 bits                }

function INT( short: uchar):    uint;  { Receives an 8 bit unsigned number and extends it to}
primitive;                          {   16 bit unsigned number by prepending 8  0's.   }

procedure QAX_CLEAN_CB( cb: Control_Block);
primitive;                          { Initializes all fields of the control block structure  }
                                    {   to 0. This is a procedure provided by BAX.          }

function FORMAT_EVENT_REPORT( event: uint; time: ulong):   Event_Report_Type;
external;                           { This function prepares an Event_Report to be sent }
                                    {   to the EVENT_LOG module.  This function is      }
                                    {   external since the structure of the             }
                                    {   Event_Report_Type has not yet been determined. }
                                    {   The parameter list may have to be modified when}
                                    {   this function is further defined.               }
```

```
                         { Module Header Definitions                       }

module PRIMITIVE_AX25_TYPE  systemprocess;  { BAX.                         }
  ip
    bax:   array[4] of Abstract_Bax_Channel( Bax_End) individual queue;
  end;

module DATA_TRANSFER_TYPE systemprocess;  {Between BAX and FTLO            }
  ip
    pc:    array[maxlinks] of Abstract_Packet_Channel( Data_Transfer_End)
           individual queue;
    bax:   Abstract_Bax_Channel( Data_Transfer_End) individual queue;
    cc:    array[2] of Data_Transfer_Control_Channel( Data_Transfer_End)
           common queue;
    el:    Event_Log_Channel( Event_End) common queue;
  end;

module PACKET_TRANSFER_TYPE systemprocess( link: Link_Type);  { FTL0       }
  ip
    pc:    Abstract_Packet_Channel( Packet_Transfer_End) individual queue;
    mc:    Mailbox_Access_Channel( Packet_Transfer_End) individual queue;
    el:    Event_Log_Channel( Event_End) common queue;
  end;

module MAILBOX_CONTROL_TYPE     systemprocess;
  ip
    mc:    array[maxlinks] of  Mailbox_Access_Channel( Mailbox_End)
           individual queue;
    cc:    array[2] of Mailbox_Admin_Channel( Mailbox_End) common queue;
    ts:    Telemetry_Stroage_Channel( Mailbox_End) individual queue;
    el:    Event_Log_Channel( Event_End) common queue;
  end;

module PASSWORD_CONTROL_TYPE( first_password: Password_Type;
                              shuffle:Shuffle_Type);   systemprocess;
  ip
    cc:    array[2] of Password_Control_Channel( Password_End) common queue;
    el:    Event_Log_Channel( Event_End) common queue;
  end;
```

```
module AUTO_CONTROL_TYPE systemprocess;     { Automatic Housekeeping Functions }
   ip
      bax:   Abstract_Bax_Channel( Data_Transfer_End) individual queue;
      acd:   Data_Transfer_Control_Channel( Control_End) individual queue;
      act:   Telemetry_Control_Channel( Control_End) individual queue;
      acp:   Password_Control_Channel( Control_End) individual queue;
      acm:   Mailbox_Admin_Channel( Control_End) individual queue;
      ace:   EPS_Control_Channel( Control_End) individual queue;
      acom:  Comm_Control_Channel( Control_End) individual queue;
      acdc:  DCS_Control_Channel( Control_End) individual queue;
      sc:    Satellite_Control_Channel( Auto_Control_End) individual queue;
      el:    Event_Log_Channel( Event_End) common queue;
   end;

module GROUND_CONTROL_TYPE systemprocess;     { Command Functions              }
   ip
      bax:   Abstract_Bax_Channel( Data_Transfer_End) individual queue;
      ccd:   Data_Transfer_Control_Channel( Control_End) individual queue;
      cct:   Telemetry_Control_Channel( Control_End) individual queue;
      ccp:   Password_Control_Channel( Control_End) individual queue;
      ccl:   SW_Load_Control_Channel( Control_End) individual queue;
      ccm:   Mailbox_Admin_Channel( Control_End) individual queue;
      cce:   EPS_Control_Channel( Control_End) individual queue;
      ccom:  Comm_Control_Channel( Control_End) individual queue;
      ccdc:  DCS_Control_Channel( Control_End) individual queue;
      sc:    Satellite_Control_Channel( Ground_Control_End) individual queue;
      el:    Event_Log_Channel( Event_End) common queue;
   end;

module PRIMITIVE_SW_LOADER_TYPE systemprocess;  { PHTX                          }
   ip
      cc:    SW_Load_Control_Channel( Loader_End) individual queue;
      bax:   Abstract_Bax_Channel( Data_Transfer_End) individual queue;
   end;

module TELEMETRY_GATHER_TYPE systemprocess; { Automatic Telemetry Gathering }
   ip
      cc:    array[2] of Telemetry_Control_Channel( Telemetry_Gather_End)
             common queue;
      ad:    A/D_Control_Channel( Command_End) individual queue;
      el:    Event_Log_Channel( Event_End) common queue;
      ts:    Telemetry_Storage_Channel( Telemetry_End) individual queue;
   end;
```

```
module A/D_DRIVER_TYPE  systemprocess;  { Driver for Analog-Digital Conv HW    }
  ip
    ad:    A/D_Control_Channel( A/D_Converter_End) individual queue;
  end;

module EVENT_LOGGER_TYPE systemprocess;
  ip
    el: array[maxlinks + 6] of Event_Log_Channel( Log_End) common queue;
  end;

module EPS_DRIVER_TYPE  systemprocess;
  ip
    cc:    array[2] of EPS_Control_Channel( EPS_Driver_End) common queue;
  end;

module COMM_DRIVER_TYPE  systemprocess;
  ip
    cc:    array[2] of Comm_Control_Channel( Comm_Driver_End) common queue;
  end;

module DCS_DRIVER_TYPE  systemprocess;
  ip
    cc:    array[2] of DCS_Control_Channel( DCS_Driver_End) common queue;
  end;


                              { Module Body Definitions                 }

body PRIMITIVE_AX25_BODY for PRIMITIVE_AX25_TYPE;              external;
body DATA_TRANSFER_BODY for DATA_TRANSFER_TYPE;
                                        { AX.25 handler - uses resources of BAX           }
   const
      mail_ssid     = 0x01;          { SSID of this module                                   }
      maxclients    = any uchar;     { the max number of 'active users at any time           }
      t_timeout     = any uchar;     { number of seconds for frame time-out timer.           }
      max_frames    = 0x07;          { frame sliding-window size.                            }
      max_tries     = any uchar;     { max # of retries for outgoing frame                   }
      packet_length = max_pdat + 2;    { max size of FTL0 level packet                       }

   type
      Client_Num    = 0..maxclients;
      Client        = record
         callsign:          Callsign_Type;
         last_comm_time:    ulong;
         data_in_progress:  boolean;
      end;
      Pac_Data = array[packet_length] of uchar;
      Client_Array = array[maxlinks] of Client;
      Data_Record =   record
         running_length:  uint;
         final_length:    uint;
         data:            Pac_Data;
      end;
      Data_Array = array[maxlinks] of Data_Record;

   var
      data:                 Pac_Data;
      length:               uint;
      in_dat:               Data_Array;  { Array of incoming data on each link           }
      clients:              Client_Array;
      cb:                   Control_Block;
      num_clients:          Client_Num;
      new_user_lockout:     boolean;
      all_user_lockout:     boolean;
      packet:               Packet_Type;
      transmit_ok:          boolean;
      i:                    uchar;   { general purpose loop counter/ index               }
      b:                    Bit_Type;
      l:                    Link_Type;

   state      NORMAL, BUSY;          { States of DATA_TRANSFER_BODY                      }
   stateset   EITHER = [NORMAL,BUSY];
```

```
function CONCAT( a, b: Pac_Data):  Pac_Data;
primitive;                        { Concatenates the array 'b' to the end of array    }
                                  {   'a', and returns the combined array of uchars.  }

function PACKET_LEN( d: Pdata):   uint;
begin
   PACKET_LEN := I_BIT_SHIFT( left, 3, INT( C_BIT_AND( d[1], 0xE0)));
   PACKET_LEN := PACKET_LEN + INT( d[0]) + 2;
end;


procedure FILL_PACKET( data: Pac_Data; var packet: Packet_Type);
primitive;                        { Takes the uchars from array 'data' and places     }
                                  {   them, in order, into the record structure of    }
                                  {   'packet'.                                       }

initialize                        { DATA_TRANSFER_BODY                                }
to NORMAL
begin
   for i := 0 to maxlinks do clients[i].callsign := 'none';
   QAX_CLEAN_CB( cb);
   cb.my_call := pansat_call;
   cb.my_ssid := mail_ssid;
   cb.t1 := t_timeout;
   cb.maxframe := max_frames;
   cb.retry := max_tries;
   cb.paclen := max_fdat;
   num_clients := 0;
   new_user_lockout := false;
   all_user_lockout := false;
   transmit_ok := true;
   output bax.qax_claim( cb);
end;

trans
from EITHER to same
when bax.qax_input
provided in_cp.kind = qat_state and in_cb.l_state = qas_disconnected
begin
   if in_cb.callsign < > nps_call then num_clients := num_clients - 1;
   clients[in_cb.link].callsign := 'none';
   output pc[in_cb.link].disconnect;
end;
```

```
from EITHER to same
when cc[b].transmitter
provided off
begin
    transmit_ok := false;
end;

from EITHER to same
when cc[b].transmittter
provided not off
begin
    transmit_ok := true;
end;

from EITHER to same
when cc[b].change_params
begin
    cb := out_cb;
end;

from EITHER to same
when bax.qax_input
provided in_cp.kind = qat_ui
begin                              { No action required - discard frame              }
end;

from NORMAL to same
when cc[b].lockout
provided l_kind = new
begin
    new_user_lockout := true;
end;

from NORMAL to same
when cc[b].unlock
provided l_kind = new
begin
    new_user_lockout := false;
end;
```

```
from EITHER to same
when pc[l].response_packet
provided transmit_ok
begin
   length := PACKET_LEN( response) + 2;
   i := 0;
   while i < length do begin
      while i < out_cb.paclen and i < length do begin
         data[i] := response[i];
         i := i + 1;
      end;
      output bax.qax_data( 1, out_cb, data, i);
      length := length - i;
      i := 0;
   end;
end;

from NORMAL to same
when bax.qax_input
provided in_cb.kind = qat_state and in_cb.l_state = qas_connect_pend
begin
   if  in_cb.his_call = nps_call then begin
      clients[in_cb.link].callsign := nps_call;
      clients[in_cb.link].last_comm_time := GET_TIME( );
      output bax.qax_con_acpt;
      output pc[in_cb.link].connection( nps_call);
   end;
   else
      if num_clients < maxclients and not new_user_lockout then begin
         num_clients := num_clients + 1;
         clients[in_cb.link].callsign := in_cb.his_call;
         clients[in_cb.link].last_comm_time := GET_TIME( );
         output bax.qax_con_acpt;
         output pc[in_cb.link].connection( in_cb.his_call);
      end;
      else output bax.qax_con_rej;
end;
```

```
from NORMAL to BUSY
when cc[b].lockout
provided l_kind = all
begin
    all_user_lockout := true;
    new_user_lockout := true;
    for i = 1 to maxlinks do
        if clients[i].callsign < > 'none' and clients[i].callsign < > nps_call then
            output qax_busy(i);
end;

from BUSY to NORMAL
when cc[b].unlock
provided l_kind = all
begin
    all_user_lockout := false;
    for i = 1 to max_links do
        if clients[i].callsign < > 'none' and clients[i].callsign < > nps_call then
            output qax_unbusy(i);
end;

from BUSY to same
when bax.qax_input
provided  in_cb.kind = qat_state and
          in_cb.l_state = qas_connect_pend and
          in_cb.his_call = nps_call
begin
    clients[in_cb.link].callsign := nps_call;
    clients[in_cb.link].last_comm_time := GET_TIME( );
    output bax.qax_con_acpt;
    output pc[in_cb.link].connection( nps_call);
end;
```

```
    from EITHER to same
    when bax.qax.input
    provided in_cb.kind = qat_data
    begin
        i := in_cb.link;
        clients[i].last_comm_time := GET_TIME( );
        if clients[i].data_in_progress then begin
            data := in_dat[i].data;
            length := in_dat[i].running_length;
            in_dat[i].data := CONCAT( data[0..length], idata);
            length := length + datal;
            if length < in_dat[i].final_length then
                in_dat[i].running_length := length;
            else begin
                clients[i].data_in_prograss := false;
                FILL_PACKET( in_dat[i].data, packet);
                output pc[i].command_packet( packet, in_dat[i].final_length-2);
            end;
        else begin
            length :=  PACKET_LEN( idata);
            if datal < length then begin
                clients[i].data_in_progress := true;
                in_dat[i].data := idata;
                in_dat[i].running_length := datal;
                in_dat[i].final_length := length;
            else begin
                FILL_PACKET( idata, packet);
                output pc[in_cb.link].command_packet( packet, length - 2);
            end;
        end;
    end;

end; { of Data_Transfer_Body }
```

```
body PACKET_TRANSFER_BODY for PACKET_TRANSFER_TYPE;

   const                             { Constants for PACKET_TRANSFER_BODY    }
      data              = 0x00;      { Packet Types                          }
      data_end          = 0x01;
      login_resp        = 0x02;
      upload_cmd        = 0x03;
      ul_go_resp        = 0x04;
      ul_error_resp     = 0x05;
      ul_ack_resp       = 0x06;
      ul_nak_resp       = 0x07;
      download_cmd      = 0x08;
      dl_error_resp     = 0x09;
      dl_aborted_resp   = 0x0A;
      dl_completed_resp = 0x0B;
      dl_ack_cmd        = 0x0C;
      dl_nak_cmd        = 0x0D;
      dir_short_cmd     = 0x0E;      { There is no difference between the short and long }
      dir_long_cmd      = 0x0F;      {    dir formats- both send complete headers.       }
      select_cmd        = 0x10;
      select_resp       = 0x11;
      del_cmd           = 0x1E;      { Delete a file... not provided for in FTL0.        }
                                     {   For del_cmd, the following packet fields apply: }
                                     {   length_lsb := 0x04; hl := 0x1E;                 }
                                     {   info[0..3] := mail_number: ulong.               }
      del_resp          = 0x1F;      { Not provided for in FTL0.                         }
                                     {   Thefollowing packet fields apply:               }
                                     {   lenght_lsb := 0x01; hl := 0x1F;                 }
                                     {   info[0] := error_code: uchar.                   }
      no_active_list    = 0x00;      { These are the FTL0 login flags, assuming          }
      active_list       = 0x08;      {   PACSET File Headers are Not used                }

   var  p:                  Packet_Type;
        client_callsign:    Callsign_Type;
        selection_active:   uchar;
        err_code:           uchar;
        current_ul_mail:    ulong;   { mail_number of file currently being uploaded.    }
        current_ul_offset:  ulong;   { Number of next byte to be uploaded in current file.}
        current_dl_mail:    ulong;   { mail_number of file currently being downloaded.  }
        current_dl_offset:  ulong;   { Num of next byte to be downloaded in current file.}
        data_length:        Pdat_Len;
        select:             Pdata;   { Raw select instruction.                          }
```

```
                                    { States of PACKET_TRANSFER_BODY          }
state      UL/DL_UNINIT, WAIT_MAILBOX, UL/DL_CMD_WAIT, UL_DATA_RX,
           UL_ABORT, DL_FILE_DATA;

stateset   ANY = [ UL/DL_UNINIT, WAIT_MAILBOX, UL/DL_CMD_WAIT,
                  UL_ABORT, DL_FILE_DATA];

                                    { Function Declarations for               }
                                    {   PACKET_TRANSFER_BODY                  }
function CURRENT_COMMAND( packet: Packet_Type):   uint;
begin
   CURRENT_COMMAND := C_BIT_AND( packet.hl, 0x1f);
end;
                                    { Procedure declarations for              }
                                    {   PACKET_TRANSFER_BODY                  }
procedure FORMAT_LOGIN_RESP( login_flag: uchar; var packet: Packet_Type);
   var     login_time:   ulong;
begin
   packet.length_lsb := 0x05;       { 5 byte information field                }
   packet.hl := login_resp;         { login_resp Packet Type                  }
   packet.info[0..3] := GET_TIME( );
   packet.info[4] := login_flag;
end;

procedure FORMAT_UPLOAD_GO_RESP( file_no, offset: ulong; var packet:
                                                        Packet_Type);
begin
   packet.length_lsb := 0x08;       { 8 byte information field                }
   packet.hl := UPLOAD_GO_RESP;
   packet.info[0..4] := file_no;
   packet.info[5..7] := offset;
end;

procedure FORMAT_NI_RESP( tag: uchar; var packet: Packet_Type);
begin
   packet.length_lsb := 0x00;       { No information                          }
   packet.hl := tag;
end;
```

```
procedure FORMAT_UL_ERROR_RESP( error: uchar; var packet: Packet_Type);
begin
    packet.length_lsb := 0x01;
    packet.hl := ul_error_resp;
    packet.info[0] := error;
end;

procedure FORMAT_UL_NAK_RESP( error: uchar; var packet: Packet_Type);
begin
    packet.length_lsb := 0x01;
    packet.hl := ul_nak_resp;
    packet.info[0] := error;
end;

procedure FORMAT_SELECT_RESP( num: uint; var packet: Packet_Type);
begin
    packet.length_lsb := 0x02;
    packet.hl := select_resp;
    packet.info[0..1] := num;
end;

procedure FORMAT_DL_ERROR_RESP( error: uchar; var packet: Packet_Type);
begin
    packet.length_lsb := 0x01;
    packet.hl := dl_error_resp;
    packet.info[0] := error;
end;

procedure FORMAT_DEL_RESP( error: uchar; var packet: Packet_Type);
begin
    packet.length_lsb := 0x01;
    packet.hl := del_resp;
    packet.info[0] := error;
end;
```

```
procedure FORMAT_DATA( len: Pdat_Len; dat: pdata; var packet: Packet_Type);
   var   high_byte:   uchar;
         msb:         uint;
begin
   packet.length_lsb := GET_LSB( len);
   high_byte := GET_MSB( len);
   msb := I_BIT_SHIFT( left, 5, INT( high_byte));
   packet.hl := GET_LSB(msb);
   packet.info[0..len-1] := dat[0..len-1];
end;

initialize                          { PACKET_TRANSFER_BODY                    }
to UL/DL_UNINIT
begin
end;

trans                               {Transition Part of PACKET_TRANSFER_BODY}
from ANY to UL/DL_UNINIT
when pc.disconnect
begin                               { Link has been terminated by client or satellite.   }
end;                                { No action required.                                }

from UL/DL_UNINIT to WAIT_MAILBOX
when pc.connection
begin
   client_callsign := callsign;
   output mc.active_sl_req( callsign);
end;

from WAIT_MAILBOX to UL/DL_CMD_WAIT
when mc.active_sl_resp
begin
   if sl then selection_active := active_list;
   else selection_active := no_active_list;
   FORMAT_LOGIN_RESP( selection_active, p);
   output pc.response_packet( p);
end;

from UL/DL_CMD_WAIT to same     { Default condition for unexpected packet or    }
when others                     {      format.                                   }
begin
   FORMAT_UL_ERROR_RESP( er_ill_formed_cmd, p);
   output pc.response_packet( p);
end;
```

```
from UL/DL_CMD_WAIT to WAIT_MAILBOX
when pc.command_packet
provided CURRENT_COMMAND( command) = upload_cmd
begin
   output mc.mail_num_req( client_callsign, command[2..5], command[6..9]);
end;

from WAIT_MAILBOX to UL/DL_CMD_WAIT
when mc.mail_num_resp
provided error_code < > no_error
begin
   FORMAT_UL_ERROR_RESP( error_code, p);
   output pc.response_packet( p);
end;

from WAIT_MAILBOX to UL_DATA_RX
when mc.mail_num_resp
provided error_code = no_error
begin
   current_ul_mail := mail_number;
   current_ul_offset := offset;
   FORMAT_UPLOAD_GO_RESP( mail_number, offset, p );
   output pc.response_packet( p);
end;

from UL_DATA_RX to UL/DL_UNINIT
when pc.disconnect                { data link terminated by client or satellite.       }
begin
   output mc.mail_close_req( current_ul_mail, current_ul_offset, false);
end;

from UL_DATA_RX to UL/DL_CMD_WAIT
when others                       { Default condition for unexpected packet or         }
begin                             {   format.                                          }
   output mc.mail_close_req( current_ul_mail, current_ul_offset, false);
   FORMAT_UL_ERROR_RESP( er_ill_formed_cmd, p);
   output pc.response_packet( p);
end;
```

```
from UL_DATA_RX to WAIT_MAILBOX
when fc.command_packet
provided CURRENT_COMMAND( command) = data_end
begin
   output mc.mail_close_req( current_ul_mail,  current_ul_offset, true);
end;

from WAIT_MAILBOX to UL/DL_CMD_WAIT
when mc.mail_close_resp
begin
   if error_code = no_error then FORMAT_NI_RESP( ul_ack_resp, p);
   else FORMAT_UL_NAK_RESP( error_code, p);
   output pc.response_packet( p);
end;

from UL_DATA_RX to WAIT_MAILBOX
when pc.command_packet
provided CURRENT_COMMAND( command) = data
   data_length := datal;
   output mc.mail_recv( current_ul_mail, current_ul_offset, lata_length, command.info);
end;

from WAIT_MAILBOX to UL_DATA_RX
when mc.mail_recv_resp
provided error_code = no_error
begin
   current_ul_offset := current_ul_offset + data_length;
end;

from WAIT_MAILBOX to UL_ABORT
when mc.mail_recv_resp
provided error_code < > no_error
begin
   FORMAT_UL_NAK_RESP( error_code, p);
   output pc.response_packet( p);
end;

from UL_ABORT to UL/DL_CMD_WAIT
when others                        { Default condition for unexpected packet or         }
begin                              {   format.                                          }
   FORMAT_UL_ERROR_RESP( er_ill_formed_cmd, p);
   output pc.response_packet( p);
end;
```

```
from UL_ABORT to UL/DL_CMD_WAIT
when pc.command_packet
provided CURRENT_COMMAND( command) = data_end
begin                              { No action required                    }
end;

from UL_ABORT to same
when pc.command_packet
provided CURRENT_COMMAND(command) = data
begin                              { No action required                    }
end;

from UL/DL_CMD_WAIT to WAIT_MAILBOX
when pc.command_packet
provided CURRENT_COMMAND( command) = del_cmd
begin
   output mc.mail_del_req( client_callsign, command.info[0..3]);
end;

from WAIT_MAILBOX to UL/DL_CMD_WAIT
when mc.mail_del_resp
begin
   FORMAT_DEL_RESP( error_code, p);
   output pc.response_packet( p);
end;

from UL/DL_CMD_WAIT to WAIT_MAILBOX
when pc.command_packet
provided CURRENT_COMMAND(command) = select_cmd
begin
   select := command.info[0..datal-1];
   output mc.mselect_req( client_callsign, select);
end;
```

```
from WAIT_MAILBOX to UL/DL_CMD_WAIT
when mc.mselect_resp
begin
    if error_code = no_error then  begin
        selection_active := active_list;
        FORMAT_SELECT_RESP( num_sel, p);
    end;
    else begin
        selection_active := no_active_list;
        FORMAT_DL_ERROR_RESP( error_code, p);
    end;
    output pc.response_packet( p);
end;

from UL/DL_CMD_WAIT to WAIT_MAILBOX
when pc.command_packet
provided  ( CURRENT_COMMAND(command) = dir_short_cmd
          or CURRENT_COMMAND( command) = dir_long_cmd)
          and (
          (command.info[0..3] < > 0x00000000 and command.info[0..3] < >
                                                              0xFFFFFFFF)
          or selection_active = active_list)
begin
    if command.info[0..3] = 0x00000000 or command.info[0..3] = 0xFFFFFFFF then
        output mc.dir_req( client_callsign, 0x00000000);
    else
        output mc.dir_req( client_callsign, command.info[0..3]);
end;

from UL/DL_CMD_WAIT to same
when pc.command_packet
provided  ( CURRENT_COMMAND(command) = dir_short_cmd
          or CURRENT_COMMAND( command) = dir_long_cmd)
          and not (
          (command.info[0..3] < > 0x00000000 and command.info[0..3] < >
                                                              0xFFFFFFFF)
          or selection_active = active_list )
begin
    FORMAT_DL_ERROR_RESP( er_selection_empty, p);
    output pc.response_packet( p);
end;
```

```
from UL/DL_CMD_WAIT to same
when pc.command
provided CURRENT_COMMAND( command) = dl_nak_cmd
begin                                    { no action required                          }
end;

from WAIT_MAILBOX to UL/DL_CMD_WAIT
when mc.directory
begin                                 { Each File Header must =< 200 bytes            }
   if no_al then selection_active := no_active_list;
   if len < > 0 then begin
      FORMAT_DATA( len, dir, p)  { Assumes 10 file headers/DataPacket               }
      output pc.response_packet( p);
      FORMAT_NI_RESP( data_end, p);
      output pc.response_packet( p);
   end;
   else begin
      FORMAT_DL_ERROR_RESP( dir[0], p)
      output pc.response_packet( p);
   end;
end;

from UL/DL_CMD_WAIT to same
when pc.command
provided  CURRENT_COMMAND( command) = download_cmd and (
          (command.info[0..3] = 0x00000000 or command.info[0..3] = 0xFFFFFFFF)
          and selection_active = no_active_list)
begin
   FORMAT_DL_ERROR_RESP( er_selection_empty, p);
   output pc.response_packet( p);
end;

from UL/DL_CMD_WAIT to DL_FILE_DATA
when pc.command
provided  CURRENT_COMMAND( command) = download_cmd and (
          (command.info[0..3] < > 0x00000000 and command.info[0..3] < >
                                                                  0xFFFFFFFF)
          or selection_active = active_list)
begin
   current_dl_offset := command.info[4..7];
   current_dl_mail := command.info[0..3];
   if current_dl_mail := 0xFFFFFFFF then current_dl_mail := 0x00000000;
   output mc.mail_req( client_callsign, current_dl_mail, current_dl_offset);
end;
```

```
from DL_FILE_DATA to UL/DL_CMD_WAIT
when mc.mail_resp
provided length = 0 and mail[0] < > 0 { Error flag from mailbox                    }
begin
   if no_al then selection_active := no_active_list;
   FORMAT_DL_ERROR_RESP( mail[0], p);
   output pc.response_packet( p);
end;

from DL_FILE_DATA to same
when mc.mail_resp
provided length < > 0 or ( mail[0] = 0  and length = 0)
begin
   if no_al then selection_active := no_active_list;
   if  length = 0 then begin
      FORMAT_NI_RESP( data_end, p);
      output pc.response_packet( p);
   end;
   else begin
      FORMAT_DATA( length, mail, p);
      output pc.response_packet( p);
      current_dl_offset := current_dl_offset + length;
      if current_dl_mail = 0x00000000 then current_dl_mail := mail_number;
      output mc.mail_req( client_callsign, current_dl_mail, current_dl_offset);
   end;
end;

from DL_FILE_DATA to UL/DL_CMD_WAIT
when pc.command_packet
provided CURRENT_COMMAND( command) = dl_ack_cmd
begin
   FORMAT_NI_PACKET( dl_completed_resp, p);
   output pc.response_packet( p);
   output mc.dl_ack( client_callsign, current_dl_mail);
end;

from DL_FILE_DATA to UL/DL_CMD_WAIT
when pc.command_packet
provided CURRENT_COMMAND( command) = DL_NAK_DMD
begin
   FORMAT_NI_PACKET( dl_aborted_resp, p);
   output pc.response_packet( p);
end;
```

```
    from DL_FILE_DATA to UL/DL_CMD_WAIT
    when others                        { Default condition for unexpected packet or    }
    begin                              {   format.                                     }
       FORMAT_DL_ERROR_RESP( err_ill_formed_cmd, p);
       output pc.response_packet( p);
    end;

end;                                   { of PACKET_TRANSFER_BODY                       }
```

```
body MAILBOX_CONTROL_BODY for MAILBOX_CONTROL_TYPE;

  const
    eof               = . .;        { End Of File marker used by operating system.      }
    null              = . .;        { The null pointer.   A pointer  which is null points }
                                    {   to nothing, and marks the end of a linked list.   }
    empty_string      = . .;        { An empty string as defined by operating system.   }
    mailbox_full      = any uint;   { Parameter for FORMAT_EVENT_REPORT                  }
    grab              = 0x01;       { A parameter needed by 'qax_claim'.                 }
    mailbox_ssid      = 0x01;
    mail_flag         = 0xbb55;
    min_file_length   = 0x00000029; { Min of 41 bytes in the initialized mail file. }
    max_ext           = 0x03E7;     { Higest mail name extension = 999 dec.         }
    default_stay_time = any ulong;  { Default mail life, in seconds from upload.    }
    numtypes          = any uchar;  { Number of different file types allowed.       }
    numcomps          = any uchar;      { Number of different file compression      }
                                    {           methods allowed.                        }
                                    { Item numbers for fields in the PANSAT file        }
                                    {   header, for use in 'select' statements.         }
    fl                = 0x00;       { flag                                               }
    mn                = 0x02;       { mail_number                                        }
    ml                = 0x06;       { length                                             }
    ft                = 0x0A;       { file_type                                          }
    ct                = 0x0B;       { compression_type                                   }
    bo                = 0x0C;       { body_offset                                        }
    dc                = 0x0E;       { download_count                                     }
    sc                = 0x0F;       { source                                             }
    pr                = 0x15;       { priority                                           }
    ut                = 0x16;       { upload_time                                        }
    et                = 0x1A;       { expire_time                                        }
    na                = 0x1E;       { mail_name                                          }
    ex                = 0x26;       { mail_extension                                     }
    nd                = 0x29;       { num_destinations                                   }
    ds                = 0x2A;       { destination callsigns or paths                     }
    ti                = 0x54;       { title                                              }
    kw                = 0x74;       { keywords                                           }

    pan_sel           = 0xFF;       {Relational operators in 'select_struct'             }
    equal_int         = 0x00;       { equal to an unsigned 1,2 or 4 byte integer         }
    equal_str         = 0x03;       { equal to a string                                  }
    great_int         = 0x10;       { greater than an unsigned 1, 2 or 4 byte integer    }
    less_int          = 0x20;       { less than an unsigned 1, 2 or 4 byte integer       }
    not_equ_int       = 0x30;       { not equal to an unsigned 1, 2 or 4 byte integer    }
    not_equ_str       = 0x33;       { not equal to a string                              }
    gr_equ_int          = 0x40;  { greater than or equal to an unsigned integer     }
    le_equ_int          = 0x50;  { less than or equal to an unsigned integer        }

    l_and               = 0x80;  { logical 'and'                                    }
    l_or                = 0xE0;  { logical 'or'                                     }

type                             { types for MAILBOX_CONTROL_BODY.                  }
    File_Ext              = 0..max_ext;
    Ext_Type              = array[3] of uchar;
    Mail_Array            = array[max_mail] of ulong;  { Array of mail_numbers      }
    Select_List           = record
        num_sel:  Num_Mail;
        sel:      Mail_Array;
    end;
    Source_Record         = record
        source_num:       uint;
        call:             Callsign_Type;
        selected:         Select_List;
        next_mail:        Num_Mail;
        next_dir:         Num_Mail;
        next_ext:         File_Ext;
        num_act:          uchar;
        next_num:         ^Source_Record; { Pointer to a source record.             }
        next_call:        ^Source_Record;
    end;
    Letter_Array          = array[26] of ^Source_Record;
    File_Desig_Array      = array[numtypes] of uchar;
    Compression_Array     = array[numcomps] of uchar;
    S_Name                = array[8] of uchar;
    File_Order            = ( date, name);

var
    done:              boolean;
    filetype:          File_Desig_Array;
    comptype:          Compression_Array;
    next_source_num:   uint;
    nsel:              Num_Mail;
    file_name:         Name_Type;
    rfile, tfile:      ^File_Type;
    first_let:         Letter_Array;
    mail_head:         ^Source_Record;
    temp1, temp2:      ^Source_Record;
    mail_num:          ulong;
    ext:               Ext_Type;
    time:               ulong;
    link:               Link_Type;
    file_length:        ulong;
    file_offset:        ulong;
    body_offset:        uint;
    file_error:         uchar;
    selectl:            Select_List;
    num_dest:           uchar;
    cs:                 Callsign_Type;
    i:                  uint;      { loop counter}
    j, k:               uchar;
    dir_dat:            Pdata;
    report:             Event_Report_Type;
    order:              File_Order;
    sname:              S_Name;
    d_file:             File_Type;

state  WAIT;

function STRING_COMPARE( str1, str2: ^Byte_Array):    boolean;
primitive;                        { Compares 'str1' to 'str2' and returns true if they  }
                                  {   are the same, otherwise returns false.            }

function STRING_FIND( str1, str2: ^Byte_Array):  boolean;
primitive;                        { Looks to see if 'str1' is contained anywhere within }
                                  {   'str2'.  Returns true if it is, and false if its not. }

function GET_LENGTH( file_name: Name_Type):   ulong;
primitive;                        { Returns the length of the stored file 'file_name'   }

function GET_LSI( number: ulong):   uint;
primitive;                        { Takes a 4 byte number and returns the least         }
                                  {   significant 2 bytes.                              }

function GET_MSI( number: ulong):  uint;
primitive;                        { Takes a 4 byte number and returns the most          }
                                  {   significant 2 bytes.                              }

function MEM_SPACE( ):  ulong;
primitive;                            { Returns the number of bytes of available        }
                                      {   space in mail box memory.                     }
```

```
function SIZE_OF( type_indicator):  uint;
primitive;                        { Takes as an argument any type and returns the    }
                                  {   number of bytes needed to store a variable of  }
                                  {   that type.                                     }

function ALLOCATE( size: uint ):    pointer_type;
primitive;                        { Allocates a  block of dynamic memory.  The       }
                                  {   number of bytes in the block is indicated by   }
                                  {   'size'.  The function returns a pointer to the }
                                  {   newly allocated block.                         }

function EXISTS( file_name: Name_Type):   boolean;
primitive;                        { Takes a complete DOS file name as an argument    }
                                  {   and returns                                    }
                                  {   true if an active file by that name currently exists}
                                  {   in the mass storage memory, otherwise returns  }
                                  {   false.                                         }

function GET_FIRST_FILE( order: File_Order; fn: S_Name):  Name_Type;
primitive;                        { Returns the name of the first active (not deleted) }
                                  {   mail file in the mass storage memory.  "First" is }
                                  {   defined as the oldest file ( the one with the  }
                                  {   earliest creation date) if 'order' is date.  If 'order'}
                                  {   is name then 'fn' is a DOS file name minus the }
                                  {   extension, and the file name returned is the one }
                                  {   with the "first" alpha-numeric extension       }
                                  {   associated with the 'fn' given.  If no file matches}
                                  {   the critria given, then empty_string is returned. }

function GET_NEXT_FILE( order: File_Order; prev_file: Name_Type ):  Name_Type;
primitive;                        { Starting at 'prev_file', searches the mail area of }
                                  {   mass storage memory for the next active file.  If }
                                  {   'order' is date, the next file is the one with the }
                                  {   next later creation date.  If 'order' is name then }
                                  {   the next file is the one with the same leading 8 }
                                  {   characters and the next higher alpha-numeric   }
                                  {   extension.  This function returns the complete file}
                                  {   name, if found and returns empty_string if no file}
                                  {   matching the criteria exists.                  }
```

```
function OPEN_FILE( file_name: Name_Type): ^File_Type;
primitive;                    { Opens the file designated by 'file_name' for      }
                              {   reading or writing. Returns a pointer to the    }
                              {   beginning of the file. If the file does not already}
                              {   exist, it will be created, and will be empty except}
                              {   for an eof mark.                                }

function READ_FILE( qty, size: uint; var file_ptr: ^File_Type):   Byte_Array;
primitive;                    { Reads blocks of bytes from memory, starting at the}
                              {   location indicated by 'file_ptr'. The number of }
                              {   blocks is determined by 'qty' and the number of }
                              {   bytes in each block is determined by 'size'. The }
                              {   bytes are placed in the (pre_allocated) variable or}
                              {   buffer space designated on the left side of an   }
                              {   assignment statement of which this function call }
                              {   is the right side. After the read, 'file_ptr' will }
                              {   point to the byte following the last byte read.  }

procedure WRITE_FILE( v: Byte_Array; num_bytes: uint; var file_ptr: ^File_Type);
primitive;                    { Writes the number of bytes indicated by           }
                              {   'num_bytes' to memory starting at the location  }
                              {   indicated by 'file_ptr'. The bytes are copied    }
                              {   beginning  from the first byte of 'v'. 'v' can be a}
                              {   variable, buffer name or file pointer. After the }
                              {   write, 'file_ptr' will point to the byte following }
                              {   the last byte written. If there is already data in }
                              {   the file at the position indicated by 'file_ptr', that}
                              {   data will be overwritten. If writing to the end of}
                              {   a file, the eof marker will be moved to indicate }
                              {   the new end of the file.                         }

procedure FILE_SEEK( num_bytes: int; var file_ptr: ^File_Type);
primitive;                    { Moves the file pointer 'file_ptr' the number of    }
                              {   bytes designated by 'num_bytes', without reading}

procedure FILE_SEEK_SET( num_bytes: int; var file_ptr: ^File_Type);
primitive;                    { Same as 'FILE_SEEK', except that 'file_ptr' is firs}
                              {   moved to the beginning of the file, and then     }
                              {   advanced the number of bytes indicated by        }
                              {   'num_bytes'.                                     }

procedure DELETE_FILE( file_name: Name_Type);
primitive;                    { Deletes the file designated by 'file_name'         }
```

```
procedure CLOSE_FILE( file_name: Name_Type);
primitive;                    { Closes the file designated by 'file_name'        }

procedure FREE(var node_ptr: pointer_type);
primitive;                    { Deallocates a dynamic memory node, and makes  }
                              {   the pointer null.                            }
```

```
procedure DECREMENT_MSG( tcall: Callsign_Type; var m_head: ^Source_Record;
                                                letter: Letter_Array);
   var    temp1, temp2, head, del_node:  ^Source_Record;
          index:                         uchar;
begin
   del_node := null;
   index := tcall[0] - 0x41;
   head := letter[index];
   if head < > null then begin
      if head -> call = tcall then begin
         head -> num_act := head -> num_act - 1;
         if head -> num_act <= 0 and head->selected.num_sel <= 0 then begin
            del_node := head;
            letter[index] := head -> next_call;
         end;
      end;
      else begin
         temp2 := head;
         temp1 := head -> next_call;
         while temp1 < > null do begin
            if temp1 -> call = tcall then begin
               temp1 -> num_act := temp1 -> num_act - 1;
               if temp1 -> num_act <= 0 and temp1->selected.num_sel <= 0
               then begin
                  del_node := temp1;
                  temp2 -> next_call := temp1 -> next_call;
               end;
            end;
            else begin
               temp2 := temp1;
               temp1 := temp1 -> next_call;
            end;
         end;
      end;
      if del_node < > null then begin
         head := m_head;
         if head = del_node then begin
            m_head := head -> next_num;
            FREE( del_node);
         end;
         else begin
            temp2 := head;
            temp1 := head -> next_num;
            while temp1 < > null do begin
```

```
                    if temp1 = del_node then begin
                        temp2 -> next_num := temp1 -> next_num;
                        FREE( del_node);
                    end;
                    else begin
                        temp2 := temp1;
                        temp1 := temp1 -> next_num;
                    end;
                end;
            end;
        end;
    end;
end;

function GET_EXT( num_ext: uint):   Ext_Type;
    var     digit:  uint;
begin
    digit := num_ext/100;
    GET_EXT[0] := digit + 0x0030;
    num_ext := num_ext - (digit * 100);
    digit := num_ext/10;
    GET_EXT[1] := digit + 0x0030;
    num_ext := num_ext - (digit * 10);
    GET_EXT[2] := num_ext + 0x0030;
end;
```

```
procedure INCREMENT_MSG( tcall: Callsign_Type; var next_sn: uint;
    m_head: ^Source_Record; letter: Letter_Array; mail_num: ulong; ext: Ext_Type);

    var     temp1, temp2, new_node:     ^Source_Record;
            index:                      uchar;
begin
    index := tcall[0] - 0x41;
    temp1 := letter[index];
    while temp1 < > null and temp1->call < > tcall do begin
        temp2 := temp1;
        temp1 := temp1->next_call;
    end;
    if temp1 = null then begin
        new_node := ALLOCATE( 1, SIZE_OF(Source_Record));
        new_node->source_num := next_sn;
        GET_NEXT_NUM( m_head, next_sn);
        new_node->call := tcall;
        new_node->selected.num_sel := 0x0000;
        new_node->next_ext := 0x0002;
        new_node->num_act := 0x01;
        new_node->next_num := null;
        new_node->next_call := null;
        temp2->next_call := new_node;
        mail_num := new_node->source_num * 0x00010000 + 0x0001;
        ext := "001";
        temp1 := m_head;
        if temp1->source_num > new_node->source_num then begin
            new_node->next_num := m_head;
            m_head := new_node;
        end;
        else begin
            while temp1->source_num < new_node->source_num and
            temp1->next_num < > null do begin
                temp2 := temp1;
                temp1 := temp1->next_num;
            end;
            if temp1->next_num = null then
                temp1->next_num := new_node;
            else begin
                new_node->next_num := temp1;
                temp2->next_num := new_node;
            end;
        end;
    end;
    else begin
        templ->num_act := templ->num_act + 1;
        mail_num := templ->source_num * 0x00010000 + templ->next_ext;
        ext := GET_EXT( templ->next_ext);
        if templ->next_ext < max_ext then
            templ->next_ext := templ->next_ext +1;
        else templ->next_ext := 0x0001;
    end;
end;

procedure COMPACT_MAIL( var mlist: ^Source_Array; llist: Letter_Array);
begin                                   { This function deletes mail files which are past their}
                                        {   expiration dates.                                   }
        var     this_file:      Name_Type;
                rfile:          ^File_Type;
                now:            ulong;
                expire:         ulong;
                call:           Callsign_Type;

    now := GET_TIME( );
    this_file := GET_FIRST_FILE( date, empty_string);
    while this_file < > empty_string do begin
        rfile := OPEN_FILE( this_file);
        FILE_SEEK( 26, rfile);
        expire := READ_FILE( 1, 4, rfile);
        if expire <= now then begin
            FILE_SEEK( 2, rfile);
            call := READ_FILE( 1, 6, rfile);
            DELETE_FILE( this_file);
            DECREMENT_MSG( call, mlist, llist)
        end;
        else CLOSE_FILE( this_file);
        this_file := GET_NEXT_FILE( date, this_file);
    end;
end;
```

```
procedure GET_NEXT_NUM( mlist: ^Source_Record, var next_sn);
   var                          { Finds the next unused source number.          }
      temp:  ^Source_Record;
      done:  boolean;
begin
   done := false;
   while not done do begin
      if next_sn < 0xFFFF then
         next_sn := next_sn + 1;
      else next_sn := 0x0001;
      temp := mlist;
      while temp < > null and temp->source_num < next_sn do
         temp := temp->next_num;
      if temp->source_num < > next_sn then
         done := true;
   end;
end;

function MAKE_FILE_NAME( source: Callsign_Type; ext: Ext_Type):    Name_Type;
begin
   MAKE_FILE_NAME[0..1] := "  ";
   MAKE_FILE_NAME[2..7] := source;
   MAKE_FILE_NAME[8] := '.';
   MAKE_FILE_NAME[9..11] := ext;
end;
```

```
function GET_NAME( mlist: ^Source_Record; mnum: ulong):  Name_Type;

   var    s, e:        uint;
          node:        ^Source_Record;
          ext:         Ext_Type;
          source:      Callsign_Type;
begin
   s := GET_MSI( mnum);          { source                                   }
   e := GET_LSI( mnum);          { extension                                }
   ext := GET_EXT( e);
   node := mlist;
   while node < > null and node->source_num < > s do
      node := node->next_num;
   if node := null then
      GET_NAME := empty_string;
   else begin
      source := node->call;
      GET_NAME := MAKE_FILE_NAME( source, ext);
   end;
end;

procedure INITIALIZE_MAIL_FILE( source: Callsign_Type; mnum: ulong);
                                ext: Ext_Type; length: ulong);
   var    new_file:  Name_Type;
          f:         ^File_Type;
begin
   new_file := MAKE_FILE_NAME( source, ext);
   f := OPEN_FILE( new_file);
   WRITE_FILE( mail_flag, 2, f);
   WRITE_FILE( mnum, 4, f);
   WRITE_FILE(length, 4, f);
   CLOSE_FILE( new_file);
end;                             { Of Procedure INITIALIZE_MAIL_FILE.       }
```

```
procedure RE_INIT_FILE( file_name: Name_Type; mail_num: ulong);

    var     r_file:     ^File_Type;
            f_length:   ulong;

begin
    r_file := OPEN_FILE( file_name);
    FILE_SEEK( fl, r_file);
    f_length := READ_FILE( 1, SIZE_OF( ulong), r_file);
    DELETE_FILE( file_name);
    INITIALIZE_MAIL_FILE( file_name[2..7], mail_num, file_name[9..11], f_length);
end;

function CRC_CHECKS_OUT( file_name: Name_Type; start: uint; stop: ulong;
                                                        crc: uint): boolean;
                                { This algorithm assumes the crc is a simple check  }
                                {   sum.                                            }
    var     num_bytes, i:   ulong;
            r_file:         ^File_Type;
            sum:            uint;
            next_char:      uchar;

begin
    if crc = 0x00 then CRC_CHECKS_OUT := true;
    else begin
        sum := 0x00;
        r_file := OPEN_FILE( file_name);
        FILE_SEEK( start, r_file);
        for i := 1 to (stop - start) do begin
            next_char := READ_FILE( 1, 1, r_file);
            sum := sum + next_char;
        end;
        CLOSE_FILE( file_name);
        CRC_CHECKS_OUT := ( sum = crc);
    end;
end;

function CHANGE_CASE( str: Byte_Array; len: uchar):     Byte_Array;
    var     i:  uchar;          { Changes any ASCII upper case letters found within}
begin                           {   'str' to lower case.  Returns the modified string. }
    for i := 0 to len-1 do
        if str[i] > 0x40 and str[i] < 0x5B then str[i] := str[i] + 0x20;
    CHANGE_CASE := str;
end;
```

```
function HEADER_CHECK( file_name: Name_Type; mail_num: ulong;
        filetype: File_Desig_Array; comptype: Compression_Array ):    boolean;

    var    good, ok:         boolean;
           r_file:           ^File_Type;
           c, d:             uchar;
           i, body_offset:   uint;
           lo:               ulong;
           call:             Callsign_Type;
           ext:              Ext_Type;
begin
    good := true;
    r_file := OPEN_FILE( file_name);
    i := READ_FILE( 1, 2, r_file);   { Read flag                                    }
    if i < > mail_flag then good := false;
    lo := READ_FILE( 1, 4, r_file);  { Read mail_number                             }
    if lo < > mail_num then good := false;
    lo := READ_FILE( 1, 4, r_file);  { Read length                                  }
    if lo < > GET_LENGTH( file_name) then good := false;
    c := READ_FILE( 1, 1, r_file);   { Read file_type                               }
    ok := false;
    for i := 0 to numtypes - 1 do   { Check file_type against all valid file_types  }
        if c = filetype[i] then ok := true;
    if not ok then good := false;   { HEADER_CHECK, continued.                      }
    c := READ_FILE( 1, 1, r_file);   { Read compression_type                        }
    ok := false;
    for i := 0 to numcomps - 1 do { Check compression_type against all valid        }
        if c = comptype[i] then ok := true;   { compression_types                   }
    if not ok then good := false;
    body_offset := READ_FILE( 1, 2, r_file);   { Read body_offset                   }
    c := READ_FILE( 1, 1, r_file);   { Read download_count                          }
    call := READ_FILE( 1, 6, r_file);     { Read source                             }
    call := CHANGE_CASE( call, 6);
    if call < > CHANGE_CASE(this_file[2..7], 6) then
        good := false;              { Check source vs file_name                     }
    c := READ_FILE( 1, 1, r_file);   { Read priority.  What to do with it is undefined.}
    lo := READ_FILE( 1, 4, r_file);  { Read upload time                             }
    if lo >= GET_TIME( ) then good := false;
    lo := READ_FILE( 1, 4, r_file);  { Read expiration time                         }
    if lo < GET_TIME( ) then good := false;
    i := READ_FILE( 1, 2, r_file);   { Read leading spaces in file name.            }
    if i < > this_file[0..1] then good := false;
    call := READ_FILE( 1, 6, r_file);     { Read file name                          }
    if CHANGE_CASE( call, 6) < > CHANGE_CASE( this_file[2..7], 6) then
```

```
        good := false;              { Check vs known file_name                 }
    ext := READ_FILE( 1, 3, r_file); { Read file extension                     }
    if CHANGE_CASE( ext, 6) < > CHANGE_CASE( file_name[9..11], 6) then
        good := false;              { Check vs known file ext                  }
    c := READ_FILE( 1, 1, r_file);  { Read num_destinations                    }
    if c > 0x09 then good := false;
    if good then begin
        if c > 0x07 then c := 0x07;
        for i := 1 to c do          { Read past all destinations               }
            call := READ_FILE( 1, 6, r_file);
        body_offset := body_offset - min_file_length - c*6;
        c := READ_FILE( 1, 1, r_file);   { Read title_length                   }
        for i := 1 to c do          { Read title                               }
            d := READ_FILE( 1, 1, r_file);
        body_offset := body_offset - 1 - c;
        c := READ_FILE( 1, 1, r_file);   { Read keyword_length                 }
        for i := 1 to c do          { Read keywords                            }
            d := READ_FILE( 1, 1, r_file);
        body_offset := body_offset - 1 - c;
        body_offset := body_offset - 4;   { Account for 2 checksum uints       }
        if body_offset < > 0 then good := false;
    end;
    HEADER_CHECK := good;
end;                                { of HEADER_CHECK( )                       }

function MSG_TO( file_name: Name_Type; call: Callsign_Type):  boolean;

    var   f:              ^File_Type;
          num_dest, i:    uchar;
          dest:           Callsign_Type;
begin
    MSG_TO := false;
    f := OPEN_FILE( file_name);
    FILE_SEEK( nd, f);
    num_dest := READ_FILE( 1, 1, f);
    if num_dest > 0 and num_dest < 0x08 then do
        for i := 1 to num_dest do begin
            dest := READ_FILE( 1, SIZE_OF(Callsign_Type), f);
            if dest := call then MSG_TO := true;
        end;
    CLOSE_FILE( file_name);
end;
```

```
procedure DEFAULT_SELECT( call: Callsign_Type; var m_head: ^Source_Record;
                          cs_array: Letter_Array; next_sn: uint; nsel: Num_Mail);
  var
    file_name:       Name_Type;
    f:               ^File_Type;
    temp1, temp2:    ^Source_Record;
    new_node:        ^Source_Record;
    num_dest, i:     uchar;
    dl_count, j:     uchar;      { download count                              }
    index:           uint;
    num_found:       Num_Mail;
    s_list:          Mail_Array;
    m_num, ul_time:  ulong;      { mail_number, upload_time                    }
begin
  num_found := 0;
  index := 0;
  file_name := GET_FIRST_FILE( date, empty_string);
  while file_name < > empty_string do begin
    f := OPEN_FILE( file_name)
    FILE_SEEK( mn, f);
    m_num := READ_FILE( 1, 4, f);
    FILE_SEEK( dc - ml, f);
    dl_count := READ_FILE( 1, 1, f);
    FILE_SEEK( ut - sc, f);
    ul_time := READ_FILE( 1, 4, f);
    FILE_SEEK( nd - er, f);
    num_dest := READ_FILE( 1, 1, f);
    CLOSE_FILE( file_name);
    if ul_time > 0 then          { Only consider completely uploaded files.    }
      if num_dest = 0x00 or num_dest > 0x07 then begin    { to 'ALL'           }
        num_found := num_found + 1;
        s_list[index..index + 3] := m_num;
        index := index + 4;
      end;
      else if dl_count < num_dest then
        if MSG_TO( file_name, call) then begin
          num_found := num_found + 1;
          s_list[index..index + 3] := m_num;
          index := index + 4;
        end;
    file_name := GET_NEXT_FILE( date, file_name);
  end;
  j := call[0] - 0x41;
  temp1 := cs_array[j];
```

```
        while templ < > null and templ->call < > call do begin
            temp2 := templ;
            templ := templ->next_call;
        end;
        if templ = null then begin
            new_node := ALLOCATE( 1, SIZE_OF(Source_Record));
            new_node := source_num := next_sn;
            GET_NEXT_NUM( m_head, next_sn);
            new_node->call := call;
            new_node->selected.num_sel := num_found;
            new_node->selected.sel := s_list;
            new_node->next_mail := 0x0000;
            new_node->next_dir := 0x0000;
            new_node->next_ext := 0x0001;
            new_node->num_act := 0x00;
            new_node->next_num := null;
            new_node->next_call := null;
            templ := m_head;
            if templ->source_num > new_node->source_num then begin
                new_node->next_num := m_head;
                m_head := new_node;
            end;
            else begin
                while templ->source_num < new_node->source_num and
                templ->next_num < > null do begin
                    temp2 := templ;
                    templ := templ-> next_num;
                end;
                if templ -> next_num = null then
                    templ->next_num := new_node;
                else begin
                    new_node->next_num := templ;
                    temp2->next_num := new_node;
                end;
            end;
        end;
        else begin
            templ->selected.num_sel := num_found;
            templ->selected.sel := s_list;
            templ->next_mail := 0x0000;
            templ->next_dir := 0x0000;
        end;
        nsel := num_found;
    end;
```

```
procedure PANSAT_SELECTION( s_struct: Pdata; call: Callsign_Type;
var m_head: ^Source_Record; cs_array: Letter_Array; next_sn: uint; nsel: Num_Mail;
error: uchar);
    var    file_name:        Name_Type;
           f:                ^File_Type;
           s_list:           Select_List;
           temp1, temp2:     ^Source_Record;
           new_node:         ^Soruce_Record;
           dest_call:        Callsign_Type;
           list_i:           uint;
           m_num:            ulong;
           abort, ok:        boolean;
           num_s, i:         uchar;
           struct_i:         uint;
           header_item:      uchar;
           first, second:    boolean;
           relop, logop, j:  uchar;
           item_len, h_len:  uchar;
           s_int, hs_int:    uchar;
           m_int, hm_int:    uint;
           l_int, hl_int:    ulong;
           compare_item:     Byte_Array;
           h_string:         Byte_Array;
           num_dest, t_len:  uchar;
           ul_time:          ulong;
begin
    abort := false;
    num_s := s_struct[1];
    s_list.num_sel :=  0;
    list_i := 0;
    file_name := GET_FIRST_FILE( date, empty_string);
    while file_name < > empty_string  and not abort do begin
        f := OPEN_FILE( file_name);
        FILE_SEEK( mn, f);
        m_num := READ_FILE( 1, 4, f);
        FILE_SEEK( ut - ml, f);
        ul_time := READ_FILE( 1, 4, f);
        if ul_time > 0 then begin
            ok := false;
            FILE_SEEK( nd - et, f);
            num_dest := READ_FILE( 1, 1, f);
            if num_dest = 0x00 or num_dest > 0x07 then ok := true;
            else
                for j := 1 to num_dest do begin
            dest_call := READ_FILE( 1, 6, f);
            if CHANGE_CASE( dest_call, 6) = CHANGE_CASE( call, 6) then
                ok := true;
        end;
    if ok then begin
        struct_i := 2;
        first := true;
        second := false;
        i := 0
        while i < num_s - 1  and not abort do begin
            relop := s_struct[struct_i];
            struct_i := struct_i + 1;
            h_item := s_struct[struct_i];
            struct_i := struct_i + 1;
            item_len := s_struct[struct_i];
            struct_i := struct_i + 1;
            if relop = equal_str or relop = not_equ_str then begin
                compare_item := s_struct[struct_i..(struct_i + item_len - 1)];
                compare_item := CHANGE_CASE( compare_item, item_len);
                struct_i := struct_i + item_len;
                if h_item < ds then begin
                    FILE_SEEK_SET( h_item, f);
                    h_string := READ_FILE( item_len, 1, f);
                    h_string := CHANGE_CASE( h_string, item_len);
                    second := STRING_ COMPARE( h_string, compare_item);
                    if relop = not_equ_str then second := not second;
                end;
                else begin
                    FILE_SEEK_SET( nd, f);
                    num_dest := READ_FILE( 1, 1, f);
                    if h_item = ds then begin
                        if num_dest > 0 and num_dest < 0x08 then
                            if item_len = 6 then begin
                                for j := 1 to num_dest do begin
                                    h_string := READ_FILE( 1, 6, f);
                                    h_string := CHANGE_CASE( h_string, 6);
                                    if not second then
                                        second := STRING COMPARE( h_string,
                                        compare_item);
                                end;
                            end;
                            else second := false;
                        else if num_dest > 0x07 then begin
                            h_string := READ_FILE( 42, 1, f);
                h_string := CHANGE_CASE( h_string, 42);
                second := STRING_FIND( compare_item, h_string);
            end;
            else second := false;
            if relop = not_equ_str then second := not second;
        end;
        else
            if h_item = ti then begin
                FILE_SEEK( ds + num_dest*6, f);
                t_len := READ_FILE( 1, 1, f);
                h_string := READ_FILE( t_len, 1, f);
                h_string := CHANGE_CASE( h_string, t_len);
                second := STRING_FIND( compare_item, h_string);
                if relop = not_equ_str then second := not second;
            end;
            else
                if h_item = kw then begin
                    FILE_SEEK( ds + num_dest*6, f);
                    t_len := READ_FILE( 1, 1, f);
                    FILE_SEEK( t_len, 1, f);
                    t_len := FILE_READ( 1, 1, f);
                    h_string := READ_FILE( t_len, 1, f);
                    h_string := CHANGE_CASE( h_string, t_len);
                    second := STRING_FIND( compare_item, h_string);
                    if relop = not_equ_str then second := not second;
                end;
                else abort := true;
    end;
end;
else
    if h_item < na then begin
        FILE_SEEK_SET( h_item, f);
        if item_len = 1 then begin
            s_int := s_struct[struct_i];
            struct_i := struct_i + 1;
            hs_int := READ_FILE( 1, 1, f);
            hl_int := 0x00000000 + hs_int;
            l_int := 0x00000000 + s_int;
        end;
        else if item_len = 2 then begin
            m_int := s_struct[struct_i..struct_i + 1];
            struct_i := struct_i + 2;
            hm_int := READ_FILE( 1, 2, f);
            hl_int := 0x00000000 + hm_int;
```

```
                l_int := 0x00000000 + m_int;
            end;
            else if item_len = 4 then begin
                l_int := s_struct[struct_i..struct_i + 3];
                struct_i := struct_i + 4;
                hl_int := READ_FILE( 1, 4, f);
            end;
            else abort := true;
            if not abort then
                if relop = equal_int then
                    second := ( hl_int = l_int);
                else if relop = great_int then
                    second := ( hl_int > l_int);
                else if relop = less_int then
                    second := ( hl_int < l_int);
                else if relop = not_equ_int then
                    second := ( hl_int < > l_int);
                else if relop = gr_equ_int then
                    second := ( hl_int >= l_int);
                else if relop = le_equ_int then
                    second := ( hl_int <= l_int);
                else abort := true;
        end;
        else abort := true;
    if not abort then begin
        i := i + 1;
        logop := s_struct[struct_i];
        if logop = l_and then begin
            struct_i := struct_i + 1;
            first := (first and second);
        end;
        else if logop = l_or then begin
            struct_i := struct_i + 1;
            first := (first or second);
        end;
        else first := (first and second);
    end;
  end;
  if first then begin
    s_list.num_sel := s_list.num_sel + 1;
    s_list.sel[list_i..list_i + 3] := m_num;
    list_i := list_i + 4;
  end;
end;
```

```
        end;
        CLOSE_FILE( file_name);
        file_name := GET_NEXT_FILE( date, file_name);
    end;
    if abort then
        error := er_poorly_formed_sel;
    else begin
        error := no_error;
        j := call[0] - 0x41;
        temp1 := cs_array[j];
        while temp1 < > null and temp1->call < > call do begin
            temp2 := temp1;
            temp1 := temp1->next_call;
        end;
        if temp1 = null then begin
            new_node := ALLOCATE( 1, SIZE_OF(Source_Record));
            new_node := source_num := next_sn;
            GET_NEXT_NUM( m_head, next_sn);
            new_node->call := call;
            new_node->selected.num_sel := s_list.num_sel;
            new_node->selected.sel := s_list.sel;
            new_node->next_mail := 0x0000;
            new_node->next_dir := 0x0000;
            new_node->next_ext := 0x0001;
            new_node->num_act := 0x00;
            new_node->next_num := null;
            new_node->next_call := null;
            temp1 := m_head;
            if temp1->source_num > new_node->source_num then begin
                new_node->next_num := m_head;
                m_head := new_node;
            end;
            else begin
                while temp1->source_num < new_node->source_num and
                temp1->next_num < > null do begin
                    temp2 := temp1;
                    temp1 := temp1-> next_num;
                end;
                if temp1 -> next_num = null then
                    temp1->next_num := new_node;
                else begin
                    new_node->next_num := temp1;
                    temp2->next_num := new_node;
                end;
            end;
        end;
        else begin
            temp1->selected.num_sel := s_list.num_sel;
            temp1->selected.sel := s_list.sel;
            temp1->next_mail := 0x0000;
            temp1->next_dir := 0x0000;
        end;
        nsel := s_list.num_sel;
    end;
end;

procedure COPY_HEADER( file_name: Name_Type; var buffer: Byte_Array;
                                                          index: ulong);
    var   r_file:       ^File_Type;
          body_offset:  uint;
begin
    r_file := OPEN_FILE( file_name);
    FILE_SEEK( bo, r_file);
    body_offset := READ_FILE( 1, 2, r_file);
    FILE_SEEK_SET( 0, r_file);
    buffer[index..index + body_offset - 1] := READ_FILE( body_offset, 1, r_file);
    CLOSE_FILE( file_name);
    index := index + body_offset;
end;
```

```
initialize
to WAIT
begin                                      { PACSAT File Types from H. Price                  }
    filetype[0] := 0x00;                   { ascii                                            }
    filetype[1] := 0x01;                   { RLI/MBL message body. Single message.            }
    filetype[2] := 0x02;                   { RLI/MBL import/export file. Multiple message.    }
    filetype[3] := 0x03;                   { UoSAT Whole Orbit Data.                          }
    filetype[4] := 0x04;                   { Microsat Whole Orbit Data.                       }
    filetype[5] := 0x05;                   { UoSAT CPE Data.                                  }
    filetype[6] := 0x06;                   { MS/PC-DOS .exe file.                             }
    filetype[7] := 0x07;                   { MS/PC-DOS .com file.                             }
    filetype[8] := 0x08;                   { Keplerian elements NASA 2-line format.           }
    filetype[9] := 0x09;                   { Keplerian elements "AMSAT" format.               }
    filetype[10] := 0x0A;                  { Simple ASCII text file, but compressed.          }
                                           { PANSAT File Types.                               }
    filetype[11] := 0xA0;                  { PANSAT short telemetry file.                     }
    filetype[12] := 0xA1;                  { PANSAT long telemetry file.                      }
    filetype[13] := 0xA2;                  { PANSAT bax telemetry file.                       }
    filetype[14] := 0xFE;                  { User defined type. User must know. 0xFF          }
                                           {   'ESCAPE ' not implemented in PANSAT file       }
                                           {   headers.                                       }
    for i := 15 to numtypes - 1 do
        filetype[i] := 0x00;               { Extra space for types defined later.             }
                                           { PACSAT file compression types - H. Price.        }
    comptype[0] := 0x00;                   { No compression                                   }
    comptype[1] := 0x01;                   { Body compressed using PKARC.                     }
    comptype[2] := 0x02;                   { Body compressed using PKZIP.                     }
    comptype[3] := 0xFE;                   { Other, user-known compression type.              }
    for i := 4 to numcomps - 1 do
        comptype[i] := 0x00;               { Extra space for compression types defined later. }
    next_source_num := 0x0001;
    for i := 0 to 25 do
        first_let[i] := null;
    mail_head := null;
end;
```

```
trans
from WAIT to same
when  mc[link].mail_num_req
begin
    if mail_number[link] = 0x00000000 then begin
        if length[link] > MEM_SPACE( ) then
            COMPACT_MAIL( mail_head, first_let);
        if length[link] <= MEM_SPACE( ) then begin
            INCREMENT_MSG( client_call[link], next_source_num, mail_head, first_let,
             mail_num, ext);
            INITIALIZE_MAIL_FILE( client_call, mail_num, ext, length[link]);
            output mc[link].mail_num_resp( mail_num, 0x00000000, no_error);
        end;
        else begin
            output mc[link].mail_num_resp( 0x00000000, 0x00000000, er_no_room);
            time := GET_TIME( );
            report := FORMAT_EVENT_REPORT( mailbox_full, time);
            output el.event_report( report);
            output cc.full_mailbox;
        end;
    end;
    else begin
        file_name := GET_NAME( mail_head, mail_number[link]);
        if file_name = empty_string then
            output mc[link].mail_num_resp( mail_number[link], 0x00000000,
             er_no_such_file_number);
        else begin
            rfile := OPEN_FILE( file_name);
            FILE_SEEK( 6, read_file);
            file_length := READ_FILE( 1, SIZEOF(ulong), rfile);
            if length[link] < > file_length then
                output mc[link].mail_num_resp( mail_number, 0x00000000,
                 er_bad_continue);
            else begin
                file_offset := GET_LENGTH( file_name) + 1;
                if file_offset >= length[link] then
                    output mc[link].mail_num_resp( mail_number[link], 0x00000000,
                     er_file_complete);
                else begin
                    if file_offset < 42 then file_offset := 0;
                    output mc[link].mail_num_resp( mail_number[link], file_offset,
                     no_error);
                end;
            end;
        end;
    end;
end;

from WAIT to same
when mc[link].mail_close_req
begin
    file_error := no_error;
    header_crc := 0x0000;
    file_name := GET_NAME( mail_head, mail_number[link]);
    if file_name = empty_string then file_error := er_no_room;
    else begin
        rfile := OPEN_FILE( file_name);
        FILE_SEEK( mn, rfile);
        mail_num := READ_FILE( 1, 4, rfile);
        file_length := READ_FILE( 1, 4, rfile);
        if GET_LENGTH( file_name) >= file_length then begin
            if mail_num = 0x00000000 then begin
                FILE_SEEK_SET( mn, rfile);
                WRITE_FILE( mail_number[link], 4, rfile);
                for i := 0 to 3 do
                    header_crc := header_crc + mail_number[link][i];
            end;
            time := GET_TIME( );
            FILE_SEEK_SET( ut, rfile);
            WRITE_FILE( time, 4, rfile);   { Write upload_time.                }
            for i := 0 to 3 do
                header_crc := header_crc + time[i];
            time := time + default_stay_time;
            WRITE_FILE( time, 4, rfile);   { Write expire_time.                }
            for i := 0 to 3 do
                header_crc := header_crc + time[i];
            WRITE_FILE( file_name[0..7], 8, rfile);
            for i := 0 to 7 do
                header_crc := header_crc + file_name[i];
            WRITE_FILE( file_name[9..11], 3, rfile);
            for i := 9 to 11 do
                header_crc := header_crc + file_name[i];
            num_dest := READ_FILE( 1, 1, rfile);
            if num_dest > 0 and num_dest < 8 then
                FILE_SEEK( num_dest*6, rfile);
            else if num_dest > 7 then
                FILE_SEEK( 42, rfile);
            j := READ_FILE( 1, 1, rfile); { Read title_length.                }
        FILE_SEEK( j, rfile);
        j := READ_FILE( 1, 1, rfile); { Read keyword_lenght.                 }
        FILE_SEEK( j, rfile);
        i := READ_FILE( 1, 2, rfile); { Read header check sum                }
        if i > 0 then begin
            header_crc := header_crc + i;
            FILE_SEEK( -2, rfile);
            WRITE_FILE( header_crc, 2, rfile);
        end;
    end;
    CLOSE_FILE( file_name);
    if req_resp[link] then begin
        if not HEADER_CHECK( file_name, mail_number[link], filetype, comptype)
        then file_error := er_bad_header;
        else begin
            rfile := OPEN_FILE( file_name);
            FILE_SEEK( bo, rfile);
            body_offset := READ_FILE( 1, 2, rfile);
            FILE_SEEK_SET( body_offset - 4, rfile);
            header_crc := READ_FILE( 1, 2, rfile);
            body_crc := READ_FILE( 1, 2, rfile);
            CLOSE_FILE( file_name);
            if not CRC_CHECKS_OUT( file_name, 0, body_offset, header_crc) then
                file_error := er_header_check;
            else if not CRC_CHECKS_OUT( file_name, body_offset, file_length,
            body_crc)
            then file_error := er_body_check;
        end;
        output mc[link].mail_close_resp( file_error);
        if file_error < > no_error then
            RE_INIT_FILE( file_name, mail_number[link]);
    end;
end;
```

```
from WAIT to same
when mc[link].mail_recv
begin
    file_name := GET_NAME( mail_head, mail_number[link]);
    if length[link] > MEM_SPACE( ) then
        COMPACT_MAIL( mail_head, first_let);
    if length[link] <= MEM_SPACE( ) and file_name < > empty_string then begin
        rfile := OPEN_FILE( file_name);
        FILE_SEEK( offset[link], rfile);
        WRITE_FILE( mail[link], length[link], rfile);
        CLOSE_FILE( file_name);
        output mc[link].mail_recv_resp( no_error);
    end;
    else
        if file_name = empty_string then
            output mc[link].mail_recv_resp( er_bad_continue);
        else begin
            output mc[link].mail_recv_resp( er_no_room);
            time := GET_TIME( );
            report := FORMAT_EVENT_REPORT( mailbox_full, time);
            output el.event_report( report);
            output cc.full_mailbox;
        end;
end;

from WAIT to same
when mc[link].mselect_req
begin
    file_error := no_error;
    if select_struct[link][0] = pan_sel then
        PANSAT_SELECTION( select_struct[link], client_call[link], mail_head, first_let,
        next_source_num, nsel, file_error);
    else DEFAULT_SELECT( client_call[link], mail_head, first_let, next_source_num,
    nsel);
    output mc[link].mselect_resp( nsel, file_error);
end;
```

```
from WAIT to same
when mc[link].mail_del_req
begin
    file_error := er_permission_denied;
    file_name = GET_NAME( mail_head, mail_number[link]);
    if file_name  = empty_string then
        file_error := er_no_such_file_number;
    else if client_call[link] = nps_call then
        file_error := no_error;
    else begin
        if file_name[2..7] < > client_call[link] then begin  { A client may only delete a}
            rfile := OPEN_FILE( file_name);  {   file which he has uploaded or which is}
            FILE_SEEK( nd, rfile);  {                    addressed to him.                }
            num_dest := READ_FILE( 1, 1, rfile);
            if num_dest = 0x01 then begin
                cs := READ_FILE( 1, SIZE_OF(Callsign_Type), rfile);
                if cs = client_call[link] then file_error := no_error;
            end;
            CLOSE_FILE( file_name);
        end;
        else file_error := no_error;
    if file_error = no_error then begin
        DELETE_FILE( this_file);
        DECREMENT_MSG( file_name[2..7], mail_head, first_let);
    end;
    output mc[link].mail_del_resp( file_error);
end;
```

```
from WAIT to same
when mc[link].dir_req
begin
    if mail_number[link] < > 0x00000000 then begin
        file_name := GET_NAME( mail_head, mail_number[link]);
        if file_name  = empty_string then
            output mc[link].directory( 0, er_no_such_file_number, false);
        else begin
            body_offset := 0;
            COPY_HEADER( file_name, dir_dat, body_offset);
            output mc[link].directory( body_offset, dir_dat, false);
        end;
    end;
    else begin
        j := client_call[link][0] - 0x41;
        temp1 := first_let[j];
        while temp1 < > null and temp->call < > client_call[link] do begin
            temp2 := temp1;     temp1 := temp1->next_call;
        end;
        if temp1 := null or temp1->selected.num_sel = 0x0000 then
            output mc[link].directory( 0, er_selection_empty, true);
        else begin
            j := 0;     file_length := 0;
            while temp1->next_dir <= temp1->selected.num_sel and j < 10 do begin
                mail_num := temp1->selected.sel[temp1->next_dir];
                file_name := GET_NAME( mail_head, mail_num);
                if file_name  < > empty_string then begin
                    COPY_HEADER( file_name, dir_dat, file_length);
                    j := j + 1;
                end;
                temp1->next_dir := temp1->next_dir + 1;
            end;
            if temp1->next_dir > temp1->selected.num_sel then begin
                done := true;
                if temp->next_mail > temp1->selected.num_sel then
                    temp1->selected.num_sel := 0x0000;
            end;
            else done := false;
            if file_length = 0 then
                output mc[link].directory( 0, er_selection_empty, true);
            else output mc[link].directory( file_length, dir_dat, done);
        end;
    end;
end;
```

```
from WAIT to same
when mc[link].mail_req
begin
  if mail_number[link] < > 0x00000000 then begin  { Particular file requested.            }
    file_name := GET_NAME( mail_head, mail_number[link]);
    if file_name = empty_string then
      output mc[link].mail_resp( er_no_such_file_number, mail_number[link], 0,
      false);
  end;
  else begin                    { "Next" file in selection list requested.             }
    j := client_call[link][0] - 0x41;
    templ := first_let[j];
    while templ < > null and temp->call < > client_call[link] do
      templ := templ->next_call;
    if ( templ = null or templ->selected.num_sel = 0x0000 or
    templ->next_mail > templ->selected.num_sel) then begin
      if templ = null or templ->selected.num_sel = 0x00 then
        done := true;
      else done := false;
      output mc[link].mail_resp( er_selection_empty, 0, 0, done);
    end;
    else begin                  { Active selection list found                          }
      mail_num := templ->selected.sel[templ->next_mail]
      file_name := GET_NAME( mail_head, mail_num);
      if file_name := empty_string then begin
        templ->next_mail := templ->next_mail + 1;
        if templ->next_mail > templ->selected.num_sel then
          if templ->next_dir > templ->selected.num_sel then begin
            templ->selected.num_sel := 0x0000;
            output mc[link].mail_resp( er_selection_empty, 0, 0, true);
          end;
          else output mc[link].mail_resp( er_selection_empty, 0, 0, false);
        else
          output mc[link].mail_resp( er_no_such_file_number, mail_num, 0,
          false);
      end;
      else begin                { Good file number found in select list.               }
        rfile := OPEN_FILE( file_name);
        FILE_SEEK( ml, rfile);
        file_length := READ_FILE( 1, 4, rfile);
        if file_length <= offset[link] then begin
          output mc[link].mail_resp( 0, mail_num, 0, false);
          CLOSE_FILE( file_name);
        end;
```

```
                else begin
                    FILE_SEEK_SET( offset[link], rfile);
                    if file_length - offset[link] > max_pdat then begin
                        dir_dat[0..max_pdat - 1] := READ_FILE( max_pdat, 1, rfile);
                        output mc[link].mail_resp( dir_dat, mail_num, max_pdat, false);
                    end;
                    else begin
                        file_offset := file_length - offset[link];
                        dir_dat[0.. file_offset - 1] := READ_FILE(  file_offset, 1, rfile);
                        output mc[link].mail_resp( dir_dat, mail_num, file_offset, false);
                    end;
                    CLOSE_FILE( file_name);
                end;
            end;
        end;
    end;
end;
```

```
from WAIT to same
when mc[link].dl_ack
begin
    j := client_call[link][0] - 0x41;
    templ := first_let[j];
    while templ < > null and temp->call < > client_call[link] do
        templ := templ->next_call;
    if ( templ < > null and templ->selected.num_sel < > 0x0000 and
    templ->next_mail < = templ->selected.num_sel) then
        if templ->selected.sel[templ->next_mail] = mail_number[link] then begin
            templ->next_mail := templ->next_mail + 1;
            if templ->next_mail > templ->selected.num_sel then
                if templ->next_dir > templ->selected.num_sel then
                    templ->selected.num_sel := 0x0000;
        end;
    file_name := GET_NAME( mail_head, mail_number[link]);
    if file_name < > empty_string then begin
        rfile := OPEN_FILE( file_name);
        FILE_SEEK_SET( dc, rfile);
        k := READ_FILE( 1, 1, rfile);
        done := false:
        FILE_SEEK_SET( nd, rfile);
        num_dest := READ_FILE( 1, 1, rfile);
        CLOSE_FILE( file_name);
        if num_dest = 0x00 or num_dest > 0x07 then done := true;
        else if MSG_TO( file_name, client_call[link]) then done := true;
        if done and k < 255 then begin
            rfile := OPEN_FILE( file_name);
            k := k + 1;
            FILE_SEEK( dc, rfile);
            WRITE_FILE( k, 1, rfile);
            FILE_SEEK( ds - sc, rfile);
            if num_dest < 0x08 then FILE_SEEK( num_dest*6, rfile);
            else FILE_SEEK( 42, rfile);
            j := READ_FILE( 1, 1, rfile); { Read title_length.                     }
            FILE_SEEK( j, rfile);
            j := READ_FILE( 1, 1, rfile); { Read keyword_length.                   }
            FILE_SEEK( j, rfile
            header_crc := READ_FILE( 1, 2, rfile);  { Read header check sum.        }
            if header_crc > 0 then begin
                header_crc := header_crc + 0x0001;
                FILE_SEEK( -2, rfile);
                WRITE_FILE( header_crc, 2, rfile);
            end;
            CLOSE_FILE( file_name);
        end;
    end;
end;
```

```
from WAIT to same
when cc.list_mail
begin
    num_files := 0;
    if bulletins then begin            { List all bulletins                                }
        file_name := GET_FIRST_FILE( name, "BULLETIN");
        while file_name < > empty_string do begin
            mail[num_files] := file_name;
            num_files := num_files + 1;
            file_name := GET_NEXT_FILE( name, file_name);
        end;
    end;
    if messages then
        if to then begin               { List all messages 'to' a certain callsign.        }
            file_name := GET_FIRST_FILE( date, empty_string);
            while file_name < > empty_string do begin
                if MSG_TO( callsign) then begin
                    mail[num_files] := file_name;
                    num_files := num_files + 1;
                end;
                file_name := GET_NEXT_FILE( date, file_name);
            end;
        end;
        else if from then begin        { List all messages 'from' a certain callsign.      }
            sname[0..1] := "  ";     sname[2..7] := callsign;
            file_name := GET_FIRST_FILE( name, sname);
            while file_name < > empty_string do begin
                mail[num_files] := file_name;
                num_files := num_files + 1;
                file_name := GET_NEXT_FILE( name, file_name);
            end;
        end;
        else begin                     { List all messages                                 }
            file_name := GET_FIRST_FILE( date, empty_string );
            while file_name < > emptry_string do begin
                if file_name[0..7] < > "BULLETIN" then begin
                    mail[num_files] := file_name;
                    num_files := num_files + 1;
                end;
                file_name := GET_NEXT_FILE( date, file_name);
            end;
        end;
    output cc.mail_list(num_files - 1, mail);
end;
```

```
from WAIT to same
when cc.post_bulletin
begin
   INCREMENT_MSG( "BULLETIN", next_source_num, mail_head, first_let,
   mail_num, ext);
   rfile := OPEN_FILE( bulletin);     { File has already been created and written by the}
   FILE_SEEK( mn, rfile);      {      command module.  Here, we just keep track      }
   WRITE_FILE( mail_num, 4, rfile);    { of how many bulletins are active, and       }
   FILE_SEEK( nd - ml, rfile);    {      write appropriate mail_number into the file  }
   num_dest := READ_FILE( 1, 1, rfile);   { header.  In this case, the mail_number   }
   if num_dest > 0x07 then        {      will not accurately reflect the extension of the}
      FILE_SEEK( 42, rfile);      {      file name, since this will be assigned by the  }
   j := READ_FILE( 1, 1, rfile); {      writing module.                              }
   FILE_SEEK( j, rfile);
   j := READ_FILE( 1, 1, rfile);
   FILE_SEEK( j, rfile);
   header_crc := READ_FILE( 1, 2, rfile);
   if header_crc > 0 then begin
      for i := 0 to 3 do
         header_crc := header_crc + mail_num[i];
      FILE_SEEK( -2, rfile);
      WRITE_FILE( header_crc, 2, rfile);
   end;
   CLOSE_FILE( file_name);
end;

from WAIT to same
when cc.delete_bulletin
begin
   DECREMENT_MSG( "BULLETIN", mail_head, first_let);
end;
```

```
from WAIT to same
when cc.purge_mail
begin
    if not all then begin
        if to then begin                { Purge all messages 'to' a certain callsign.           }
            file_name := GET_FIRST_FILE( date, empty_string);
            while file_name < > empty_string do begin
                if MSG_TO( callsign) then begin
                    if post_time > 0 then begin
                        rfile := OPEN_FILE( file_name);
                        FILE_SEEK( ut, rfile);
                        ul_time := READ_FILE( 1, 4, rfile);
                        CLOSE_FILE( file_name);
                        if ul_time < post_time then begin
                            DELETE_FILE( file_name);
                            DECREMENT_MSG( file_name[2..7], mail_head, first_let);
                        end;
                    end;
                    else begin
                        DELETE_FILE( file_name);
                        DECREMENT_MSG( file_name[2..7], mail_head, first_let);
                    end;
                end;
                file_name := GET_NEXT_FILE( date, file_name);
            end;
        end;
        else if from then begin  { Purge all messages 'from' a certain callsign.           }
            sname[0..1] := "  ";
            sname[2..7] := callsign;
            file_name := GET_FIRST_FILE( name, sname);
            while file_name < > empty_string do begin
                if post_time > 0 then begin
                    rfile := OPEN_FILE( file_name);
                    FILE_SEEK( ut, rfile);
                    ul_time := READ_FILE( 1, 4, rfile);
                    CLOSE_FILE( file_name);
                    if ul_time < post_time then begin
                        DELETE_FILE( file_name);
                        DECREMENT_MSG( callsign, mail_head, first_let);
                    end;
                end;
                else begin
                    DELETE_FILE( file_name);
                    DECREMENT_MSG( callsign, mail_head, first_let);
                end;
                file_name := GET_NEXT_FILE( name, file_name);
            end;
        end;
        else begin                           { Purge all messages.                    }
            file_name := GET_FIRST_FILE( date, empty_string);
            while file_name < > empty_string do begin
                if file_name[0..7] < > "BULLETIN"
                and file_name[0..7] < > "USRTELEM" then begin
                    if post_time > 0 then begin
                        rfile := OPEN_FILE( file_name);
                        FILE_SEEK( ut, rfile);
                        ul_time := READ_FILE( 1, 4, rfile);
                        CLOSE_FILE( file_name);
                        if ul_time < post_time then begin
                            DELETE_FILE( file_name);
                            DECREMENT_MSG( file_name[2..7], mail_head, first_let);
                        end;
                    end;
                    else begin
                        DELETE_FILE( file_name);
                        DECREMENT_MSG( file_name[2..7], mail_head, first_let);
                    end;
                end;
                file_name := GET_NEXT_FILE( date, file_name);
            end;
        end;
    end;
end;
```

```
    from WAIT to same
    when ts.store_usr_telem
    begin
        INCREMENT_MSG( "USRTELEM", next_source_num, mail_head, first_let,
        mail_num, ext);
        rfile := OPEN_FILE( telem);  { File has already been created and written by the   }
        FILE_SEEK( mn, rfile);       {      telemetry module.  Here, we just keep track    }
        WRITE_FILE( mail_num, 4, rfile);    { of how many usr telem files are active, and }
        FILE_SEEK( nd - ml, rfile);   {      write appropriate mail_number into the file   }
        num_dest := READ_FILE( 1, 1, rfile);   { header.  In this case, the mail_number    }
        if num_dest > 0x07 then       {      will not accurately reflect the extension of the}
            FILE_SEEK( 42, rfile);    {      file name, since this will be assigned by the  }
        j := READ_FILE( 1, 1, rfile); {      writing module.                                }
        FILE_SEEK( j, rfile);
        j := READ_FILE( 1, 1, rfile);
        FILE_SEEK( j, rfile);
        header_crc := READ_FILE( 1, 2, rfile);
        if header_crc > 0 then begin
            for i := 0 to 3 do
                header_crc := header_crc + mail_num[i];
            FILE_SEEK( -2, rfile);
            WRITE_FILE( header_crc, 2, rfile);
        end;
        CLOSE_FILE( telem);
    end;

    from WAIT to same
    when ts.delete_user_telem          { Telemetry module has already deleted the file and }
    begin                              {  is only notifying the mail box module.           }
        DECREMENT_MSG( "USRTELEM", mail_head, first_let);
    end;

end;                                   {  of MAILBOX_CONTROL_BODY                          }
```

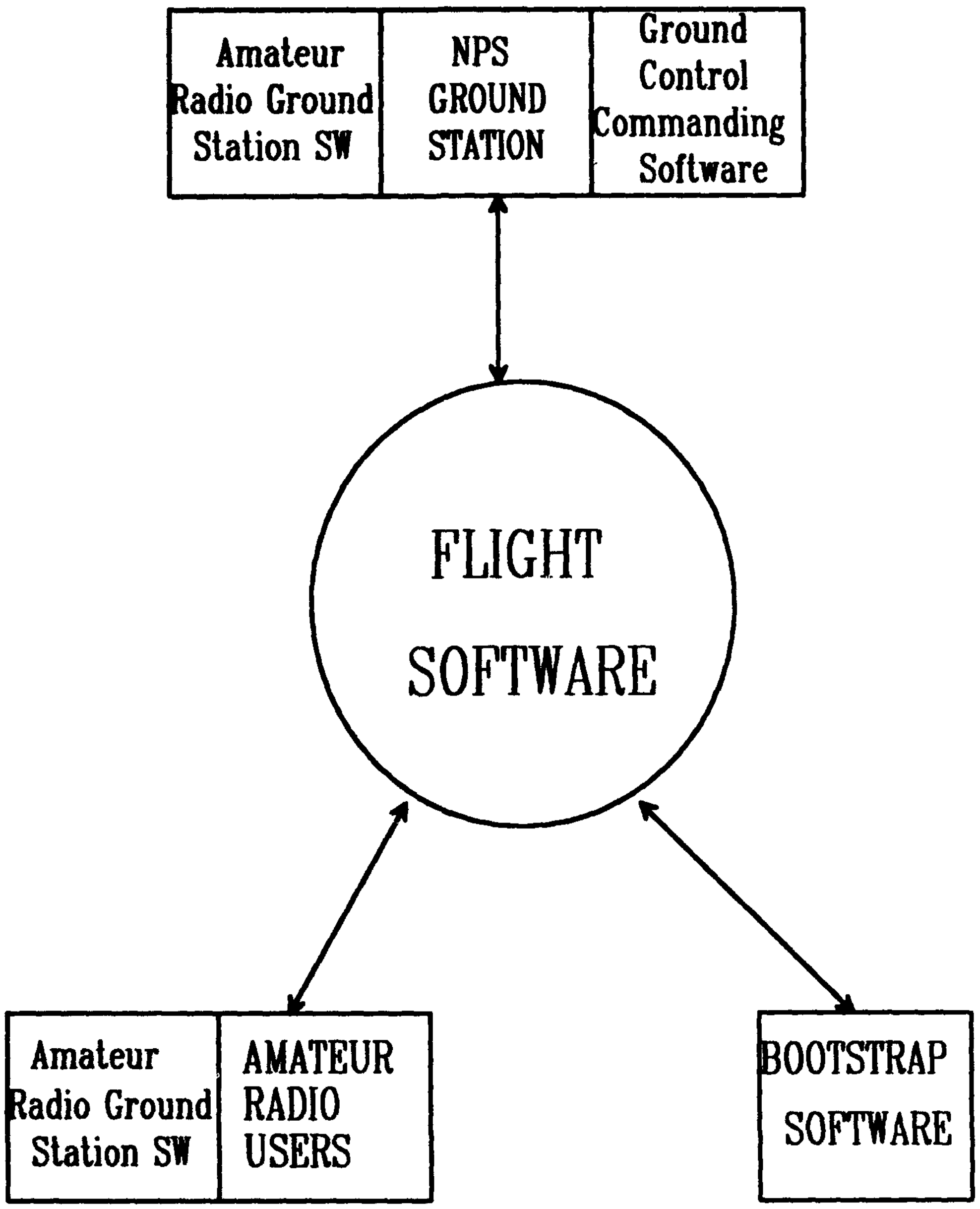
```
body PASSWORD_CONTROL_BODY for PASSWORD_CONTROL_TYPE; external;

body AUTO_CONTROL_BODY for AUTO_CONTROL_TYPE;         external;

body GROUND_CONTROL_BODY for GROUND_CONTROL_TYPE;     external;

body PRIMITIVE_SW_LOADER for PRIMITIVE_SW_LOADER_TYPE; external;

body TELEMETRY_GATHER_BODY for TELEMETRY_GATHER_TYPE; external;

body A/D_DRIVER_BODY for A/D_DRIVER_TYPE;             external;

body EVENT_LOGGER_BODY for EVENT_LOGGER_TYPE;         external;

body EPS_DRIVER_BODY for EPS_DRIVER_TYPE;             external;

body COMM_DRIVER_BODY for COMM_DRIVER_TYPE;           external;

body DCS_DRIVER_BODY for DCS_DRIVER_TYPE;             external;
```

```
                                 {Module instantiation and channel connection    }
modvar                           {  section for Flight  Software Specification.   }
   Primitive_AX25:        PRIMITIVE_AX25_TYPE;
   Primitive_SW_Loader:   PRIMITIVE_SW_LOADER_TYPE;
   Data_Transfer:         DATA_TRANSFER_TYPE;
   Packet_Transfer:       array[maxlinks] of PACKET_TRANSFER_TYPE;
   Mailbox_Control:       MAILBOX_CONTROL_TYPE;
   Ground_Control:        GROUND_CONTROL_TYPE;
   Auto_Control:          AUTO_CONTROL_TYPE;
   Event_Logger:          EVENT_LOGGER_TYPE;
   Password_Control:      PASSWORD_CONTROL_TYPE;
   Telemetry_Gather:      TELEMETRY_GATHER_TYPE;
   A/D_Driver:            A/D_DRIVER_TYPE;
   EPS_Driver:            EPS_DRIVER_TYPE;
   Comm_Driver:           COMM_DRIVER_TYPE;
   DCS_Driver:            DCS_DRIVER_TYPE;

initialize                       { Initialization Part of the Specification      }
begin
   init Primitive_AX25 with PRIMITIVE_AX25_BODY;
   init Primitive_SW_Loader with PRIMITIVE_SW_LOADER_BODY;
   init Data_Transfer with DATA_TRANSFER_BODY;
   init Mailbox_Control with MAILBOX_CONTROL_BODY;
   init Ground_Control with GROUND_CONTROL_BODY;
   init Auto_Control with AUTO_CONTROL_BODY;
   init Event_Logger with EVENT_LOGGER_BODY:
   init Password_Control with PASSWORD_CONTROL_BODY;
   init Telemetry_Gather  with TELEMETRY_GATHER_BODY;
   init A/D_Driver with A/D_DRIVER_BODY;
   init EPS_Driver with EPS_DRIVER_BODY;
   init Comm_Driver with COMM_DRIVER_BODY;
   init DCS_Driver with DCS_DRIVER_BODY;
   all link: Link_Type  do begin
      init Packet_Transfer[link] with PACKET_TRANSFER_BODY;
      connect Data_Transfer.pc[link] to Packet_Transfer[link].pc;
      connect Mailbox_Control.mc[link] to Packet_Transfer[link].mc;
      connect Event_Logger.el[link] to Packet_Transfer[link].el;
   end;
   connect Primitive_AX25.bax[0] to Data_Transfer.bax;
   connect Primitive_AX25.bax[1] to Ground_Control.bax;
   connect Primitive_AX25.bax[2] to Auto_Control.bax;
   connect Primitive_AX25.bax[3] to Primitive_SW_Loader.bax;
   connect Ground_Control.ccd to Data_Transfer.cc[0];
   connect Ground_Control.cct to Telemetry_Gather.cc[0];
```

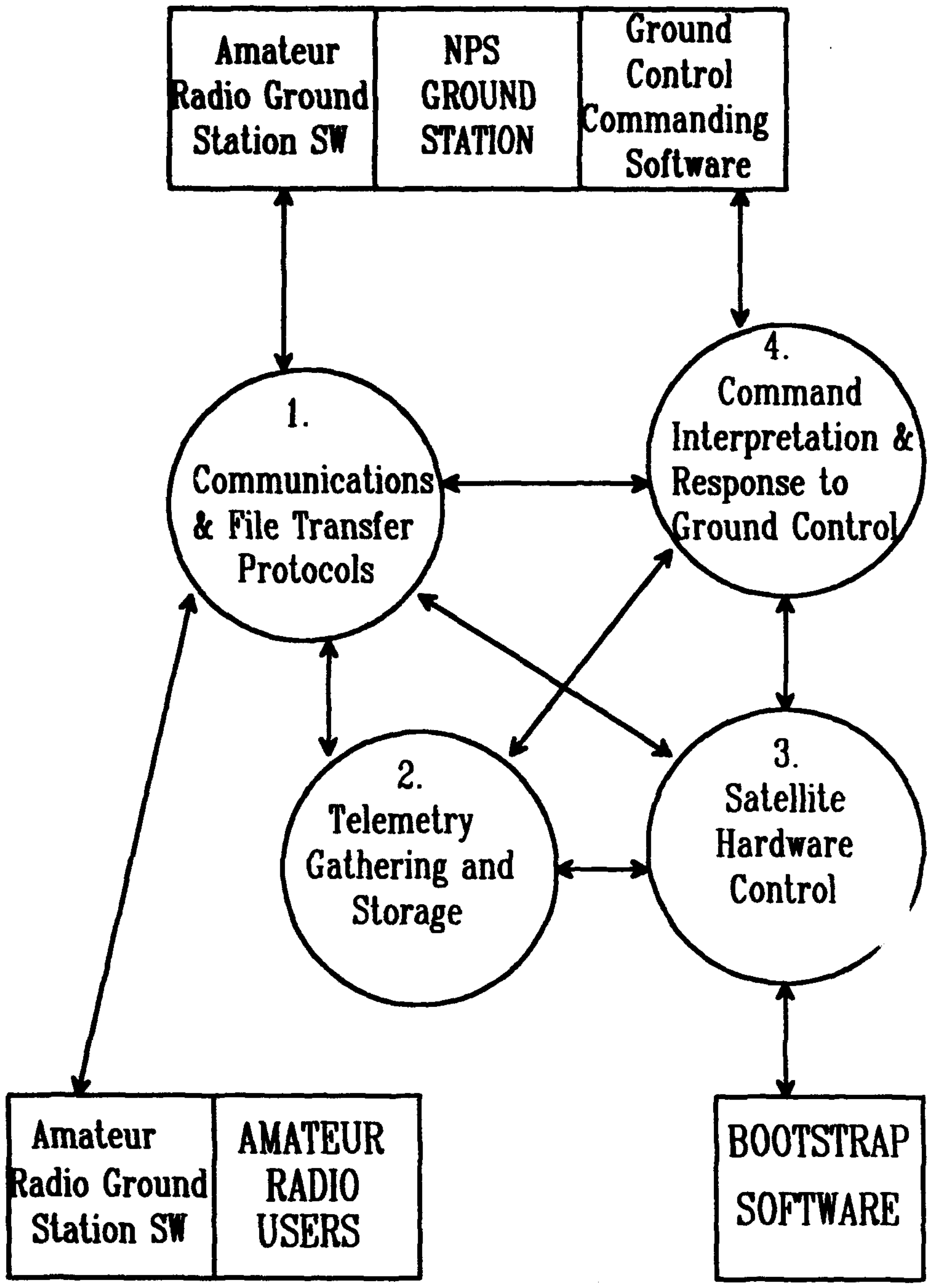
```
    connect Ground_Control.ccl to Primitive_SW_Loader.cc[0];
    connect Ground_Control.ccp to Password_Control.cc[0];
    connect Ground_Control.ccm to Mailbox_Control.cc[0];
    connect Ground_Control.cce to EPS_Driver.cc[0];
    connect Ground_Control.ccom to Comm_Driver.cc[0];
    connect Ground_Control.ccdc to DCS_Driver.cc[0];
    connect Ground_Control.el to Event_Logger.el[maxlinks];
    connect Ground_Control.sc to Auto_Control.sc;
    connect Auto_Control.acd to Data_Transfer.cc[1];
    connect Auto_Control.act to Telemetry_Gather.cc[1];
    connect Auto_Control.acp to Password_Control.cc[1];
    connect Auto_Control.acm to Mailbox_Control.cc[1];
    connect Auto_Control.ace to EPS_Driver.cc[1];
    connect Auto_Control.acom to Comm_Driver.cc[1];
    connect Auto_Control.acdc to DCS_Driver.cc[1];
    connect Auto_Control.el to Event_Logger. el[maxlinks + 1];
    connect Telemetry_Gather.ad to A/D_Driver.ad;
    connect Telemetry_Gather.el to Event_Log.el[maxlinks + 2];
    connect Telemetry_Gather.ts to Mailbox_Control.ts;
    connect Mailbox_Control.el to Event_Logger.el[maxlinks + 3];
    connect Data_Transfer.el to Event_Logger.el[maxlinks + 4];
    connect Password_Control.el to Event_Logger.el[maxlinks + 5];
end;

end.                                    { of Flight Software Specification            }
```

APPENDIX B - DATA FLOW DIAGRAMS

DFD: CONTEXT DIAGRAM

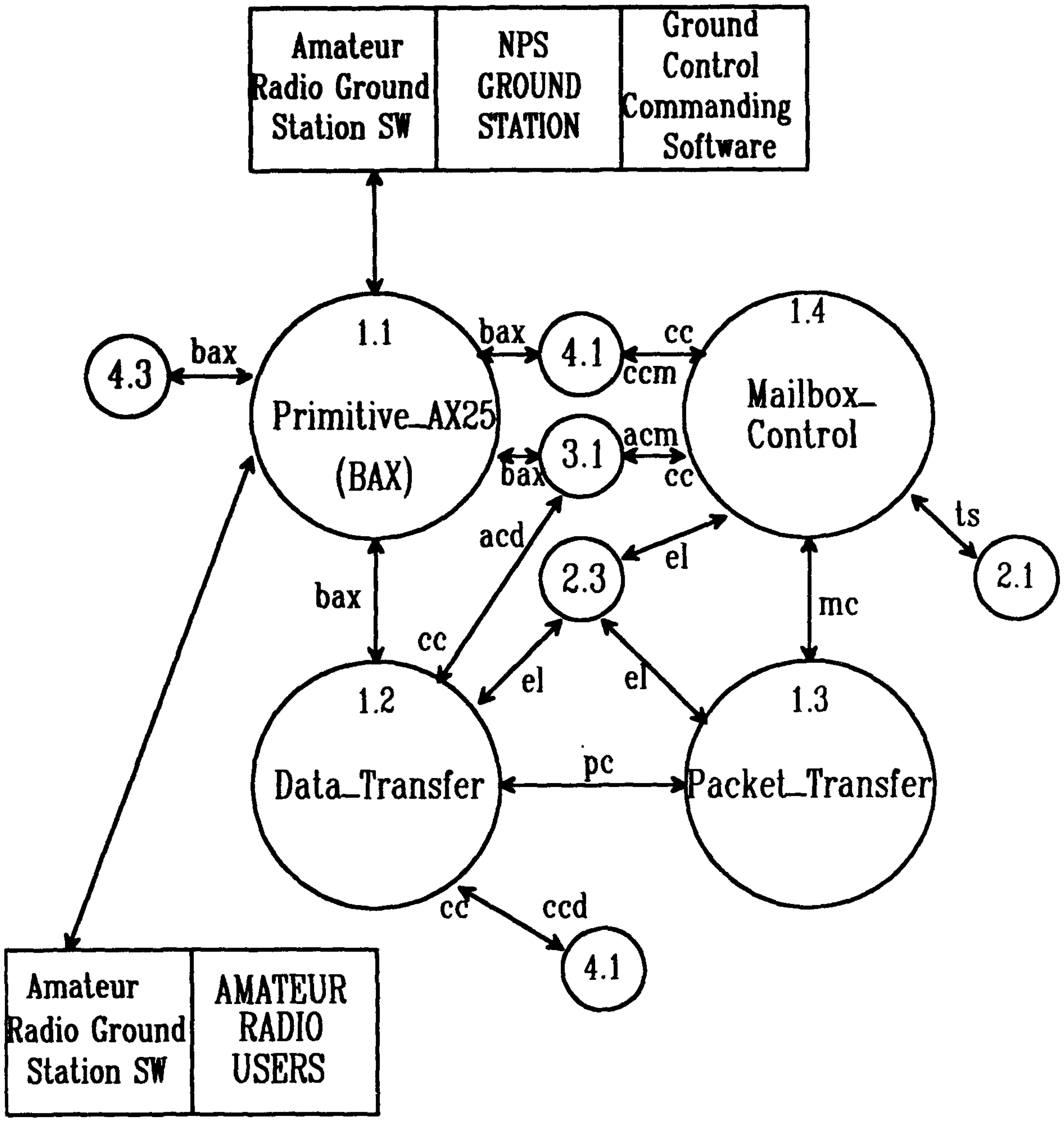

Amateur Radio Ground Station SW
NPS GROUND STATION
Ground Control Commanding Software
FLIGHT SOFTWARE
Amateur Radio Ground Station SW
AMATEUR RADIO USERS
BOOTSTRAP SOFTWARE

DFD 0 - FLIGHT SOFTWARE

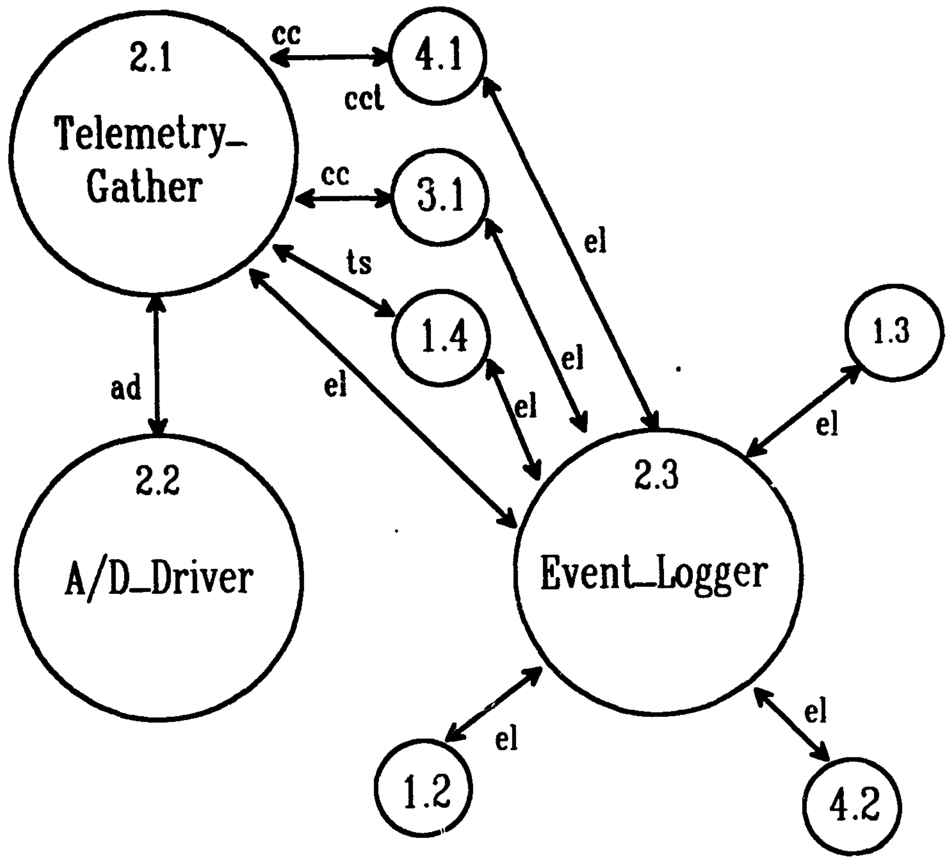


Amateur Radio Ground Station SW
NPS GROUND STATION
Ground Control Commanding Software
1. Communications & File Transfer Protocols
4. Command Interpretation & Response to Ground Control
2. Telemetry Gathering and Storage
3. Satellite Hardware Control
Amateur Radio Ground Station SW
AMATEUR RADIO USERS
BOOTSTRAP SOFTWARE

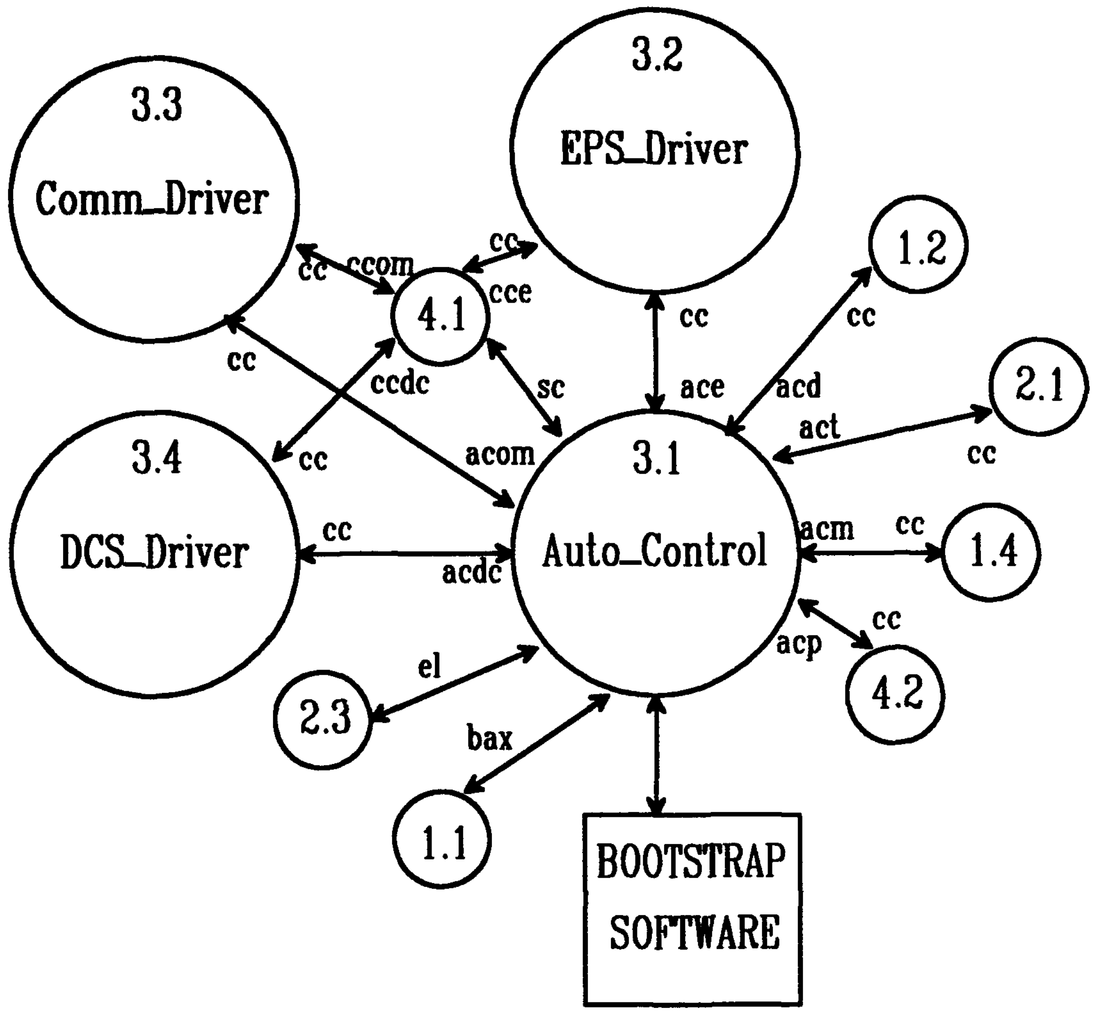
DFD 1 - Communications & File Transfer Software



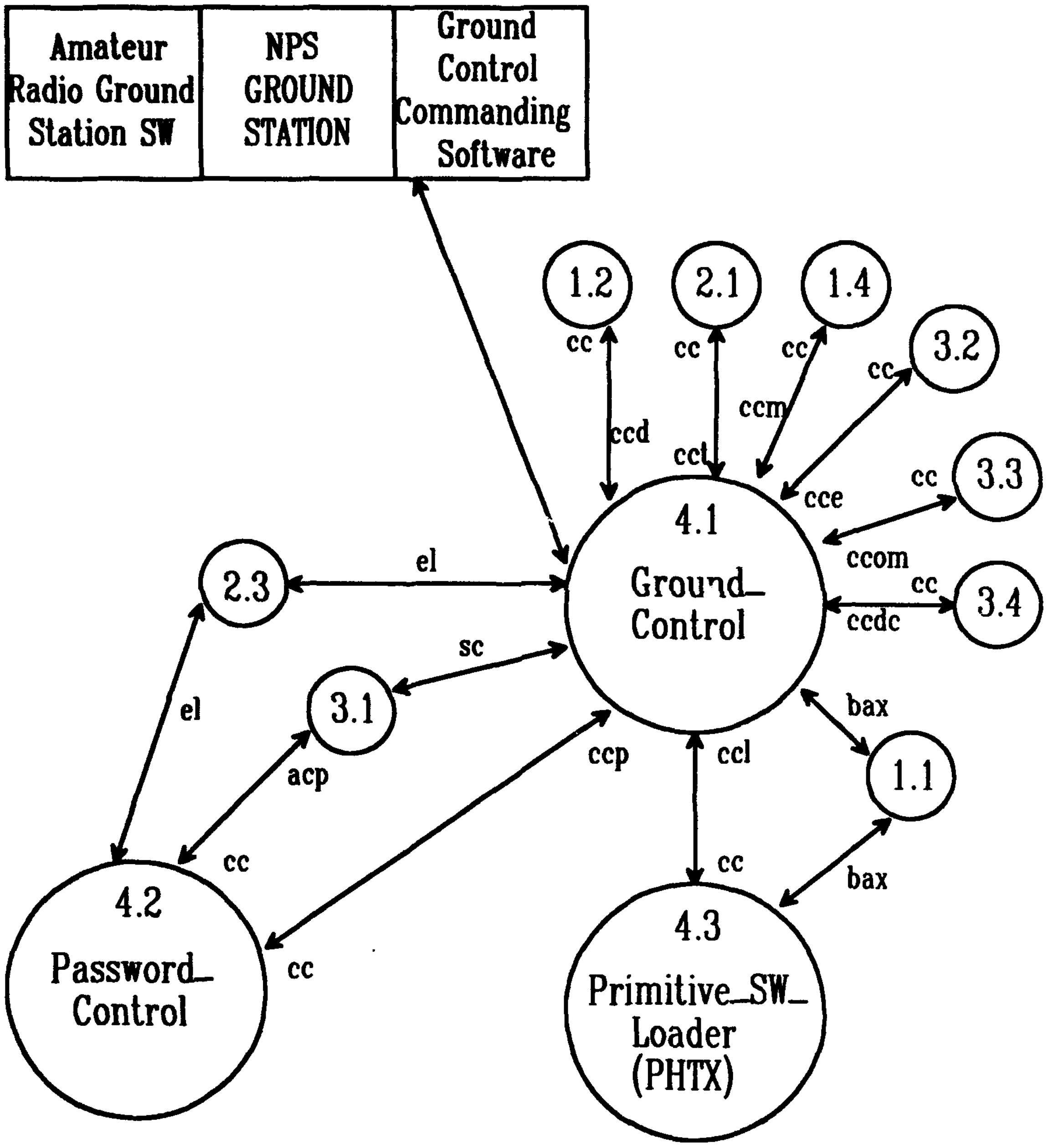
DFD 2 - Telemetry Gathering & Storage



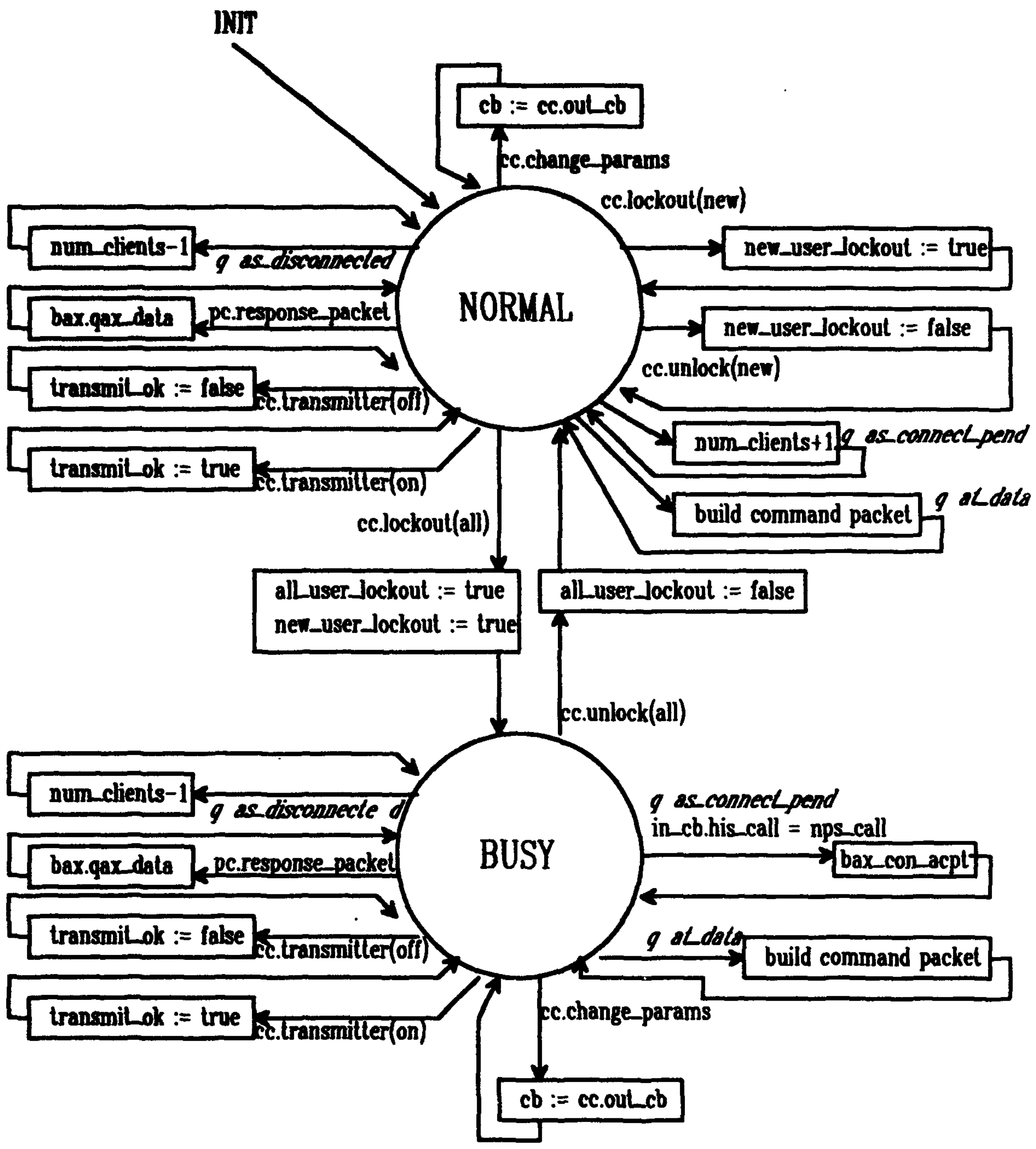
DFD 3 – Satellite Hardware Control



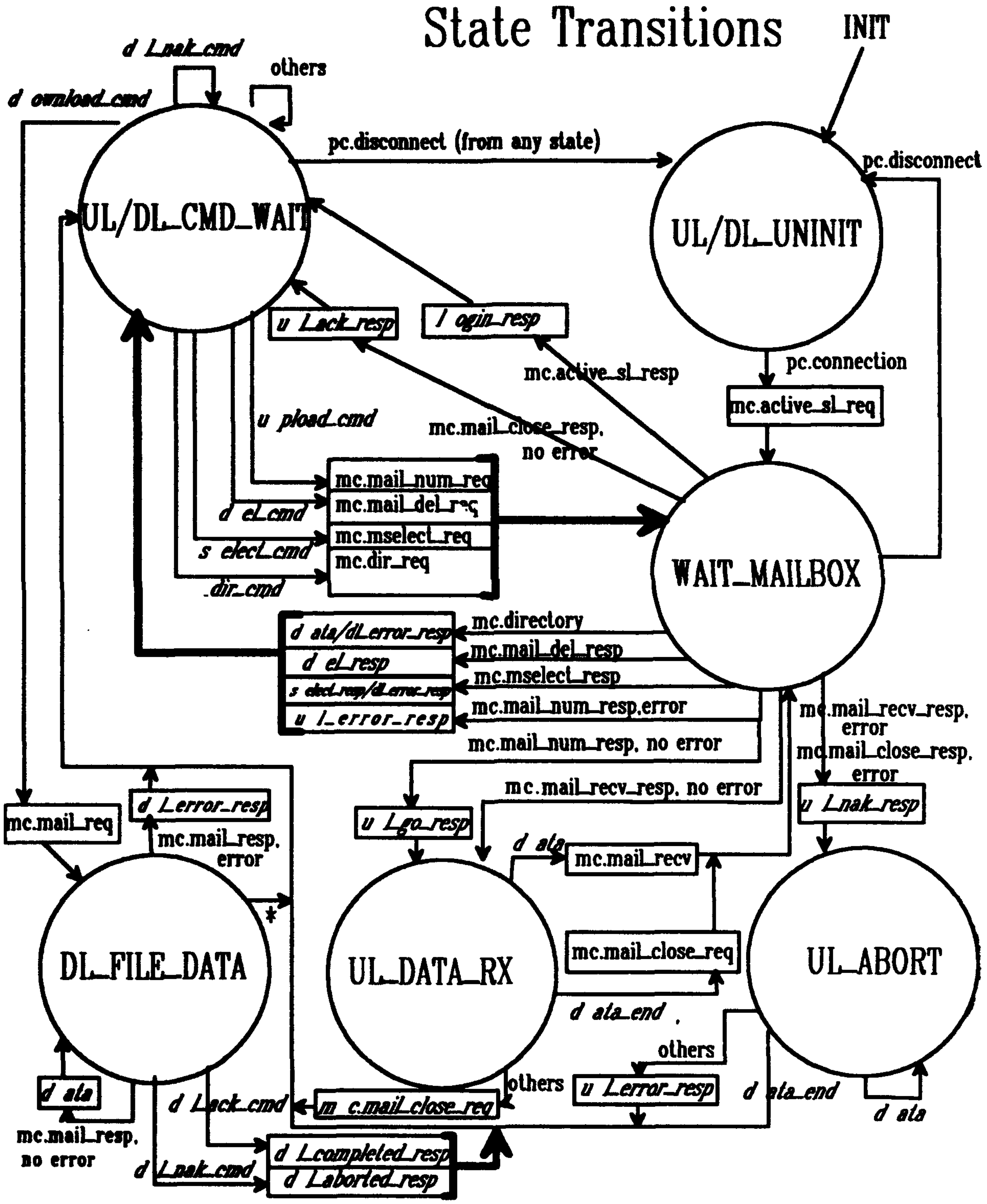
DFD 4 - Command Interpretation & Response to Ground Control



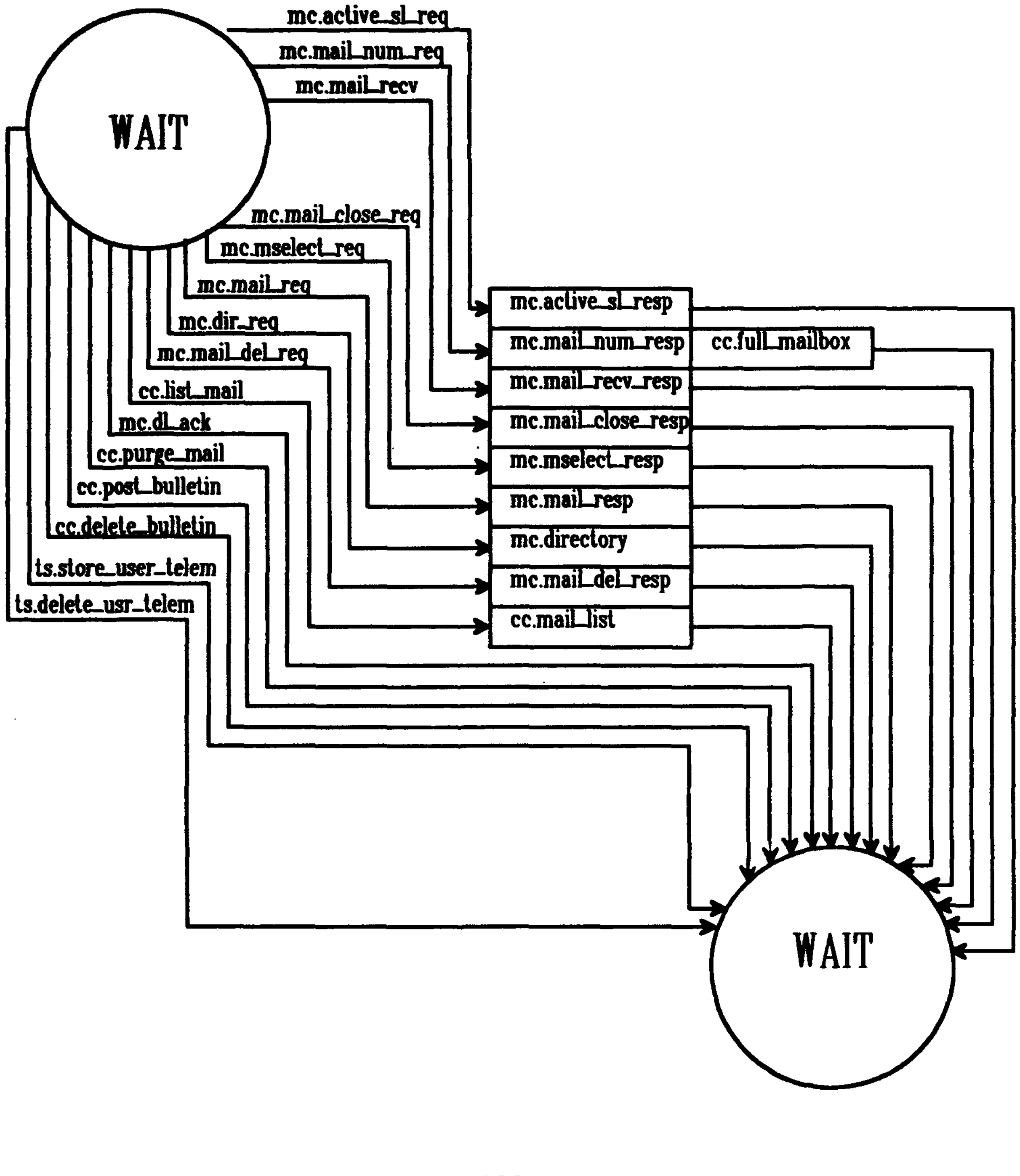
DFD 1.2 - Data Transfer Module State Transitions



DFD 1.3 - Packet_Transfer Module State Transitions



DFD 1.4 - Mailbox_Control Module Response to Messages



APPENDIX C - ESTELLE SYNTAX

Table C.1 contains the subset of Pascal syntax which was utilized in Appendix A. Note that cells with double lines on top and bottom contain definitions somewhat modified from those found in [Ref. 8]. A more complete lexicon and construction rules can be found in Annex C of [Ref. 8].

Key:

1) Each definition ends with a period, "." .

2) The or symbol, "|", denotes a choice among options.

3) Components enclosed by square brackets, "[ ]", are optional.

4) Parentheses, "( )", are used for grouping components in order to clarify definitions.

5) Components enclosed by curly brackets, "{ }", may be included zero or more times.

6) Symbols shown within quotes, " ", must be typed exactly as they appear. (They will be found in bold-face type within the specification.)

TABLE C.1 PASCAL SYNTAX USED IN APPENDIX A
letter = "a" | "b" | ... | "z"| "A" | "B" | ... | "Z".
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "A" | "B" | "C" | "D" | "E" | "F".
special-symbol = "+" | "-" | "*" | "=" | "<" | ">" | "[" | "]" | "(" | ")" | ";" | "^" | "< >" | "<=" | ">=" | ":=" | ".." | word-symbol.

TABLE C.1 PASCAL SYNTAX USED IN APPENDIX A

```
word-symbol = "and" | "array" | "begin" | "case" | "const" | "do" | "downto"
              | "else" | "end" | "false" | "for" | "function" | "if" | "in" |
              "not" | "null" | "of" | "or" | "procedure" | "record" |
              "repeat" | "then" | "to" | "true" | "type" | " until" | "var" |
              "while".

identifier = letter { letter | digit }.

unsigned-integer = digit {digit} | "0x" digit {digit}.

character-string = " ' " string-character { string-character } " ' ".

comment = "{" any-sequence-of-characters-and-separations-of-lines-not-
              containing-right-brace "}".

block =       constant-definition-part
              type-definition-part
              variable-declaration-part
              procedure-and-function-declaration-part
              statement-part.

constant-definition-part = [ "const" constant-definition ";" { constant-definition
              ";" } ].

type-definition-part = [ "type" type-definition ";" { type-definition ";" } ].

variable-declaration-part = [ "var" variable-declaration ";" { variable-declaration
              ";" } ].

procedure-and-function-declaration-part = { ( procedure-declaration | function-
              declaration ) ";" }.

statement-part = compound-statement.

constant-definition = identifier "=" constant.

constant = [sign] ( unsigned-integer | constant-identifier ) | character-string.

constant-identifier = identifier.

type-definition = identifier "=" type-denoter.

type-denoter = ordinal-type | new-type.

new-type = new-ordinal-type | new-structured-type | new-pointer-type.
```

TABLE C.1 PASCAL SYNTAX USED IN APPENDIX A
structured-type-identifier = type-identifier.
pointer-type-identifier = type-identifier.
type-identifier = identifier.
ordinal-type = new-ordinal-type | ordinal-type-identifier.
new-ordinal-type = enumerated-type | subrange-type.
ordinal-type-identifier = uchar | uint | ulong | int | boolean.
uchar = 8-bits-binary-data-or-1-byte-unsigned-integer-or-1-ascii-character.
uint = 2-byte-unsigned-integer.
ulong = 4-byte-unsigned-integer.
int = [sign] unsigned-integer.
sign = "+" | "-".
boolean = "true" | "false".
enumerated-type = "(" identifier-list ")".
identifier-list = identifier ("," identifier).
subrange-type = constant ".." constant.
structured-type = new-structured-type | structured-type-identifier.
new-structured-type = array-type | record-type.
array-type = "array" "[" number-components "]" { "[" number-components "]" } "of" component-type.
number-components = unsigned-integer.
component-type = type-denoter.
record-type = "record" field-list "end".
field-list = record-section { ";" record-section }.

TABLE C.1 PASCAL SYNTAX USED IN APPENDIX A
record-section = identifier-list ":" type-denoter.
pointer-type = new-pointer-type | pointer-type-identifier.
new-pointer-type = "^" type-identifier.
variable-declaration = identifier-list ":" type-denoter.
variable-access = entire-variable | component-variable | identified-variable.
entire-variable = variable-identifier.
variable-identifier = identifier.
component-variable = indexed-variable | field-designator.
indexed-variable = array-variable "[" index-expression "]" { "[" index-expression "]" }.
array-variable = variable-access.
index-expression = expression.
field-designator = record-variable "." field-specifier | field-identifier.
record-variable = variable-access.
field-specifier = field identifier.
field-identifier = identifier.
identified-variable = "^" pointer-variable.
pointer-variable = variable-access.
procedure-declaration = procedure-heading ";" directive | procedure-identification ";" procedure-block | procedure-heading ";" procedure-block.
procedure-heading = "procedure" identifier [formal-parameter-list].
procedure-identification = "procedure" procedure-identifier.
procedure-identifier = identifier.
procedure-block = block.

TABLE C.1 PASCAL SYNTAX USED IN APPENDIX A
function-declaration = function-heading ";" directive | function-identification ";" function-block | function-heading ";" function-block.
function-heading = "function" identifier [formal-parameter-list] ":" result-type.
function-identification = "function" function-identifier.
function-identifier = identifier.
function-block = block.
result-type = type-denoter.
formal-parameter-list = "(" formal-parameter-section { ";" formal-parameter-section } ")".
formal-parameter-section = value-parameter-specification variable-parameter-specification.
value-parameter-specification = identifier-list ":" type-identifier.
variable-parameter-specification = "var" identifier-list ":" type-identifier.
expression = simple-expression [relational-operator simple-expression].
simple-expression = [sign] term { adding-operator term }.
term = factor { multiplying-operator factor }.
factor = variable-access | unsigned-constant | function-designator | set-constructor | "(" expression ")" | "not" factor.
unsigned-constant = unsigned-number | character-string | constant-identifier | "null".
set-constructor = "[" [member-designator { "," member-designator }] "]".
member-designator = expression [".." expression].
multiplying-operator = "*" | "and".
adding-operator = "+" | "-" | "or".
relational-operator = "=" | "< >" | "<" | ">" | "<=" | ">=" | "in".
boolean-expression = expression.

TABLE C.1 PASCAL SYNTAX USED IN APPENDIX A
function-designator = function-identifier [actual-parameter-list].
actual-parameter-list = "(" actual-parameter { "," actual-parameter } ")".
actual-parameter = expression | variable-access | procedure-identifier | function-identifier.
statement = (simple-statement | structured-statement).
simple-statement = empty-statement | assignment-statement | procedure-statement.
empty-statement = .
assignment-statement = (variable-access | function identifier) ":=" expression.
procedure-statement = procedure-identifier ([actual-parameter-list]).
structured-statement = compound-statement | conditional-statement | repetitive-statement.
statement-sequence = statement { ";" statement }.
compound-statement = "begin" statement-sequence "end".
conditional-statement = if-statement | case-statement.
if-statement = "if" boolean-expression "then" statement [else-part].
else-part = "else" statement.
case-statement = "case" case-index "of" case-list-element { ";" case-list-element } [";"] "end".
case-list-element = case-constant-list ":" statement.
case-constant-list = case-constant { "," case-constant }.
case-constant = constant.
case-index = expression.
repetitive-statement = repeat-statement | while-statement | for-statement.
repeat-statement = "repeat" statement-sequence "until" boolean-expression.
while-statement = "while" boolean-expression "do" statement.

TABLE C.1 PASCAL SYNTAX USED IN APPENDIX A
for-statement = "for" control-variable ":=" initial-value ("to" | "downto") final-value "do" statement.
control-variable = entire-variable.
initial-value = expression.
final-value = expression.

The following table lists Estelle-specific reserved words which have been used in Appendix A. The expected location and function associated with each is also indicated.

TABLE C.2 ESTELLE-SPECIFIC RESERVED WORDS

Reserved Word	Location	Function
specification	Beginning of entire specification block.	Identifies the name of the specification.
any	In a constant declaration.	Declares that a value of the indicated type must be chosen during implementation.
. .	By itself, on the right-hand side of a type definition.	Indicates that the actual internal details of the type have not yet been determined. The final definition may be implementation-dependant.
channel	In the **channel** definition section, which immediately follows the global constant, type and and variable declaration sections.	Indicates the beginning of a channel definition. Followed by the channel name, and then, within parentheses, the end-point names.

TABLE C.2 ESTELLE-SPECIFIC RESERVED WORDS

Reserved Word	**Location**	**Function**
by	Within a **channel** definition.	'by' is followed by one of the **channel** end point names and then by a list of the messages which can be sent from that end point. Following each message name, in parentheses, is a list of the parameters for that message type.
module	In the module header definition section, which follows the global function and procedure declarations which follow the **channel** definition section.	Indicates the beginning of a module header definition. "**module**" is followed by the name of the module type. The module header definition defines the interfaces with the module.
system process	In the module header definition following the name of the module type.	Indicates that the module is an autonomous process, not a subprocess enclosed within another.
ip	In the module header definition.	Indicates the beginning of the list of interface points for the module. It is followed by the **channel** names, each of which is given a channel type from among those defined in the channel definition section. In parentheses is indicated which end point this module plays the role of. That in turn, defines the type of messages which can be sent by this module.
individual queue	In an interface point declaration within a module header definition.	Indicates that all messages to or from this module via the indicated **channel** will be maintained in an **individual queue** for that module alone.

TABLE C.2 ESTELLE-SPECIFIC RESERVED WORDS

Reserved Word	**Location**	**Function**
common queue	In an interface point declaration within a module header definition.	Indicates that the module will share the queue for this **channel** with all other "**common queue**" modules playing the role of the same kind of end point for the same kind of channel. Or, if a module has an **array** of **channels** of the same type, all of the **channels** may use a **common queue.**
body	In the module **body** definition section, which follows the module header definition section.	Indicates the beginning of a module **body** definition. This is where the actual behavior of the module is defined.
external	In the module body definition section, following the body name and the module type.	Indicates that the **body** definition for this module is **external** to the current specification. It may not yet have been developed, may be under development by another team, or it may be completely **external** to the system at hand, with only the interface defined by the module header definition being of any importance.
state	Within a module body definition, following the local **const**, **type** and **var** declaration sections.	Marks the beginning of the list of state names for this module.

TABLE C.2 ESTELLE-SPECIFIC RESERVED WORDS

Reserved Word	**Location**	**Function**
stateset	Within a module body definition, following the list of state names.	Defines a common name to be used for several states, when they have similar transitions. If the module state machine reacts the same way to a particular stimulus when in any one of several different states, these states may be grouped together by a stateset so that the behavior need only be written out once for all the affected states.
primitive	In a function or procedure declaration.	Indicates that the algorithmic details of the function or procedure are not included in the present specification. The function or procedure may be deemed to be commonly understood or readily available from the operating system, or the details may simply not be relevant to the aspect of the system currently under consideration.
initialize	Following the last local function or procedure declaration within a module body definition.	Indicates the initial state of the module state machine when it is instantiated. Statements between the "**begin**" and "**end**" keywords may be used to set up initial variable values, and to take any other automatic "start-up" actions.
trans	Following the initialization section in the module body definition.	Indicates the beginning of the state transition section of the module body. All possible state transitions will be listed within this section.
from	In the transition section of the module body definition.	Indicates the state from which the transition takes place.

TABLE C.2 ESTELLE-SPECIFIC RESERVED WORDS

Reserved Word	**Location**	**Function**
to	In the transition section of the module body definition.	Indicates the state in which the module will be, following the transition.
when	Following the "**from - to**" clause of the transition statement.	Identifies the stimulus which may trigger the transition. "**when**" is usually followed by the name of an interface point, with a period, ".", and the kind of message from that channel which could trigger the transition. Any parameters associated with the incoming message may be referenced directly by their value-parameter names within the "**begin - end**" block of the transition statement.
provided	Following the "**when**" clause of the transition statement.	Indicates any further conditions for the transition to occur. '**provided**' is usually followed by an expression in which one of the parameters of the "**when message**" is compared with a necessary condition. The transition only occurs when the module is in the "**from state**", the "**when message**" arrives, and the message parameter meets the necessary condition. When the transition occurs, all of the statements between "**begin**" and "**end**" are executed before the module enters the "**to state**" and waits for the next stimulus.

TABLE C.2 ESTELLE-SPECIFIC RESERVED WORDS

Reserved Word	**Location**	**Function**
output	Within the "**begin** - **end**" block of a transition statement.	Indicates that a message is to be sent out at the interface point indicated. The interface point name is shown on the left side of a period, with the type of message on the right side. If there are any message parameters, variables of the appropriate types must be prepared with the proper values, and must be included in parentheses following the message name.
modvar	Following all module body definitions.	Indicates the beginning of the module instantiation and **channel** connection section of the specification.
initialize	In the **modvar** section.	Indicates the beginning of the module instantiation section, which will define exactly how many copies of each module type will be created, using which module bodies.
init	In the module instantiation section.	"**init**" indicates initialize, but really means instantiate. Use the module body indicated by the "**with**" clause. (there could be more than one module body for a particular module type).
connect	In the **modvar** section, following the module instantiations.	Indicates that the interface points of two modules will be connected together. Further defines which specific **channels** go between which specific module instantiations.

LIST OF REFERENCES

1. Price, H. E., *BekTek Spacecraft Operating System SCOS Reference Manual*, Bethel Park, PA, March, 1992.

2. Fox, T., "AX.25 Level 2 Protocol," Proceedings of the First ARRL Amateur Radio Computer Networking Conference, pp. 2.4-2.14, ARRL, Newington, CT, October, 1981.

3. Stallings, W., *Data and Computer Communications*, pp.326-335, Macmillan Publishing Company, 1994.

4. Price, H. E., *BekTek AX.25 Protocol System BAX Reference Manual*, Bethel Park, PA, March, 1992.

5. Price, H. E., Ward, J., "PACSAT Broadcast Protocol," Proceedings of the ARRL Amateur Radio 9th Computer Networking Conference, pp. 232-244, ARRL, Newington, CT, September, 1990.

6. Price, H. E., Ward, J., "Pacsat Protocol: File Transfer Level 0," Proceedings of the ARRL Amateur Radio 9th Computer Networking Conference, pp. 209-231, ARRL, Newingtion, CT, September, 1990.

7. Price, H. E., Ward, J., "Pacsat File Header Definition," Proceedings of the ARRL Amateur Radio 9th Computer Networking Conference, pp. 245-252, ARRL, Newingtion, CT, September, 1990.

8. ISO/TC 97, *Information Processing Systems - Open Systems Interconnection - Estelle: A Formal Description Technique Based on an Extended State Transition Model*, ISO 9074:1989(E), Genève, Switzerland, July, 1989.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria VA 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey CA 93943-5101	2
3.	Department Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5121	1
4.	Professor Douglas J. Fouts, Code EC/Fs Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5121	2
5.	Professor Frederick W. Terman, Code EC/Tz Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5121	1
6.	Professor Michael Ross, Code AA/Ro PANSAT Project Lead Naval Postgraduate School Monterey, California 93943-5121	4
7.	Mr. Jim Horning, Code SP/Jh Space Systems Academic Group Naval Postgraduate School Monterey, California 93943-5121	2
8.	Teresa O. Ford, LT, USN 2101 Crystal Plaza Arcade #258 Arlington, VA 22202	1