

Pilot Programmer's Manual

XEROX

610E00160
September 1985

**Xerox Corporation
Office Systems Division
2100 Geng Road
MS 5827
Palo Alto, California 94303**

Copyright © 1984, Xerox Corporation. All rights reserved.
XEROX®, 8010, and XDE are trademarks of XEROX CORPORATION
Printed in U.S.A.



Preface

This document is one of a series of manuals written to aid in programming and operating the Xerox Development Environment (XDE).

Comments and suggestions on this document and its use are encouraged. The form at the back of this document has been prepared for this purpose. Please send your comments to :

Xerox Corporation
Office Systems Division
XDE Technical Documentation, M/S 5827
2300 Geng Road
Palo Alto, California 94303

Preface



Table of contents

1 Introduction

1.1	General structure of system software	1-1
1.2	Files	1-2
1.3	General characteristics of Pilot	1-2
1.3.1	Processes, monitors, and synchronization	1-4
1.3.2	Virtual memory, files, and volumes.	1-5
1.3.3	Stream, device, and communication interfaces	1-6
1.4	Pilot concepts	1-7
1.4.1	Stateless enumerators.	1-7
1.4.2	Synchronous and asynchronous operations	1-8
1.5	Notation and conventions	1-8
1.6	Common Software.	1-9
1.7	What follows	1-10

2 Environment

2.1	Processor environment	2-1
2.1.1	Basic types and constants	2-1
2.1.2	Device numbers and device types	2-4
2.2	Processor interface	2-5
2.2.1	BitBlt	2-5
2.2.2	TextBlt	2-10
2.2.3	Checksum	2-14
2.2.4	ByteBlt	2-14
2.2.5	Other Mesa machine operations	2-15
2.3	System timing and control facilities	2-18
2.3.1	Universal identifiers	2-18
2.3.2	Network addresses	2-19
2.3.3	Timekeeping facilities	2-20
2.3.4	Control of system power	2-23

Table of contents

2.3.5	Pilot's state after booting	2-24
2.4	Mesa run-time support	2-26
2.4.1	Processes and monitors.	2-26
2.4.2	Programs and configurations	2-31
2.4.3	Traps and signals	2-35
2.4.4	Calling the debugger or backstop	2-36
2.5	Client startup.	2-37
2.6	Coordinating subsystems' acquisition of resources	2-37
2.6.1	Use of the Supervisor	2-38
2.6.2	Supervisor facilities	2-39
2.6.3	Exception handling	2-42
2.7	General object allocation	2-42
2.7.1	Basic types	2-43
2.7.2	Basic procedures and errors	2-43

3 Streams

3.1	Semantics of streams	3-2
3.2	Operations on streams	3-3
3.2.1	GetBlock and PutBlock	3-3
3.2.2	Additional data transmission operations	3-6
3.2.3	Subsequence types	3-7
3.2.4	Attention flags	3-8
3.2.5	Timeouts	3-9
3.2.6	Stream positioning.	3-9
3.3	Creating streams	3-9
3.4	Control over physical record characteristics	3-11
3.5	Transducers, filters, and pipelines	3-13
3.5.1	Representing filters and transducers	3-14
3.5.2	Stream component managers	3-18

4 File Storage and Memory

4.1	Physical volumes	4-1
4.1.1	Physical volume name and size.	4-2
4.1.2	Physical volume errors.	4-2
4.1.3	Drives and disks	4-3
4.1.4	Disk access, Pilot volumes, and non-Pilot volumes	4-4
4.1.5	Physical volume creation	4-6
4.1.6	Scavenging	4-6
4.1.7	Logical volume operations on physical volumes	4-8
4.1.8	Miscellaneous operations on physical volumes	4-9

4.2	Logical volumes	4-10
4.2.1	Volume name and size	4-10
4.2.2	Logical and physical volumes	4-11
4.2.3	Volume error conditions	4-12
4.2.4	Creating and erasing logical volumes	4-12
4.2.5	Volume status and enumeration	4-13
4.2.6	Opening and closing volumes	4-14
4.2.7	Volume attributes	4-15
4.2.8	Volume root directory	4-16
4.3	Files	4-17
4.3.1	File naming	4-17
4.3.2	Addressing within files	4-18
4.3.3	File types	4-18
4.3.4	File error conditions	4-20
4.3.5	File creation and deletion	4-21
4.3.6	File attributes	4-21
4.4	Scavenging	4-22
4.4.1	Scavenging a volume	4-23
4.4.2	Scavenger log file	4-24
4.4.3	Operations on log files	4-26
4.4.4	Investigating and repairing damaged pages	4-27
4.5	Virtual memory management	4-29
4.5.1	Fundamental concepts of virtual memory	4-29
4.5.2	Mapping files to virtual memory intervals	4-32
4.5.3	Explicitly reading and writing virtual memory	4-35
4.5.4	Swapping	4-37
4.5.5	Access control	4-39
4.5.6	Explicit allocation of virtual memory and special intervals	4-39
4.5.7	Map unit and swap unit attributes, utility operations	4-42
4.6	Pilot memory management	4-43
4.6.1	Zones	4-44
4.6.2	Heaps	4-49
4.7	Logging	4-55
4.7.1	Writing into the log file	4-55
4.7.2	Reading a log file	4-58

5 I/O Devices

5.1	Channel structure and initialization	5-1
5.1.1	Data transfer	5-2
5.1.2	Device specific commands	5-5
5.1.3	Device status	5-5
5.2	Keyset, keyboards, and mouse	5-6

Table of contents

5.3	The user terminal	5-11
5.3.1	The display image	5-11
5.3.2	Smooth scrolling	5-13
5.3.3	The keyboard and keyset	5-15
5.3.4	The mouse	5-15
5.3.5	The sound generator	5-15
5.4	Floppy disk channel	5-16
5.4.1	Drive characteristics	5-16
5.4.2	Diskette characteristics	5-17
5.4.3	Status	5-17
5.4.4	Transfer operations	5-18
5.4.5	Non-transfer operations	5-19
5.5	Floppy file system	5-20
5.5.1	Accessing files on the diskette	5-20
5.5.2	Snapshotting and replication of the floppy volume	5-23
5.5.3	Managing the floppy volume	5-24
5.6	TTY Port channel	5-28
5.6.1	Creating and deleting the TTY Port channel	5-28
5.6.2	Data transfer	5-29
5.6.3	Data transfer status	5-29
5.6.4	TTY Port operations	5-30
5.6.5	Device status	5-31
5.7	TTY Input/Output	5-32
5.7.1	Starting and stopping	5-32
5.7.2	Signals and errors	5-33
5.7.3	Output	5-33
5.7.4	Utilities	5-34
5.7.5	String input operations	5-35
5.7.6	String output operations	5-36
5.7.7	Numeric input operations	5-37
5.7.8	Numeric output operations	5-38

6 Communication

6.1	Well known sockets	6-2
6.2	Packet exchange	6-4
6.2.1	Types and constants	6-5
6.2.2	Signals and errors	6-6
6.2.3	Procedures	6-7
6.3	Network streams	6-9
6.3.1	Types and constants	6-10
6.3.2	Creating network streams	6-12

6.3.3	Signals and errors	6-14
6.3.4	Utilities	6-17
6.3.5	Attributes of Network streams	6-19
6.4	Routing	6-23
6.4.1	Types and constants	6-23
6.4.2	Signals and errors	6-24
6.4.3	Procedures	6-24
6.5	RS232C communication facilities	6-27
6.5.1	Correspondents	6-27
6.5.2	Environment	6-29
6.5.3	RS232C channel	6-32
6.5.4	Procedures for starting and stopping the channel	6-43
6.5.5	Auto-dialing	6-43
6.6	Courier	6-46
6.6.1	Definition of terms	6-46
6.6.2	Binding	6-46
6.6.3	Remote procedure calling	6-49
6.6.4	Errors	6-53
6.6.5	Bulk data	6-58
6.6.6	Description routines	6-59
6.6.7	Miscellaneous	6-65

7 Editing and Formatting

7.1.	ASCII character definitions	7-1
7.2	Formatting	7-2
7.2.1	Binding	7-2
7.2.2	Specifying the destination of the output	7-2
7.2.3	String editing	7-2
7.2.4	Editing numbers	7-3
7.2.5	Editing dates	7-4
7.2.6	Editing network addresses	7-5
7.3	Strings	7-5
7.3.1	Sub-strings	7-6
7.3.2	Overflowing string bounds	7-6
7.3.3	String operations	7-6
7.4	Time	7-10
7.4.1	Binding	7-10
7.4.2	Operations	7-10
7.5	Memory stream	7-12
7.5.1	Errors	7-12
7.5.2	Procedures	7-12

Table of contents

8	System Generation and Initialization	
8.1	System components	8-1
8.2	Pilot initialization	8-2
8.3	Volume initialization	8-3
8.3.1	Formatting physical volumes	8-5
8.3.2	Checking drives for bad pages	8-6
8.3.3	Microcode and boot files	8-6
8.3.4	Miscellaneous operations	8-9
8.4	Communication initialization	8-11
8.5	Booting	8-12
8.5.1	Creating a boot file	8-13
8.5.2	Writing the contents of a boot file	8-13
8.5.3	Making a boot file bootable	8-13
8.5.4	Installing a boot file	8-14
8.5.5	Booting a boot file	8-14
8.5.6	Updating a boot file	8-14
8.5.7	Atomic saving and restoring of Pilot instances	8-15
9	The Backstop	
9.1	Implementing a backstop	9-1
9.1.1	Initializing a backstop log file	9-2
9.1.2	Control flow	9-2
9.1.3	Logging errors	9-2
9.2	Reading backstop log files	9-4
10	Online Diagnostics	
10.1	Communication Diagnostics	10-1
10.1.1	Ethernet echo testing	10-2
10.1.2	Gathering Ethernet statistics	10-7
10.1.3	RS232C testing	10-9
10.1.4	Dialer testing	10-13
10.2	Bitmap Display, Keyboard, and Mouse Diagnostics	10-14
10.3	Lear Siegler Diagnostics	10-17
10.4	Floppy Diagnostics	10-18
Appendices		
A	Performance Criteria	
A.1	Physical memory requirements of Pilot	A-1
A.2	Execution speed and client program profile	A-2
A.2.1	Memory management	A-2

A.2.2	File management	A-3
A.2.3	Communication via the Ethernet	A-3
A.2.4	Processes	A-3
B	Assigning and Managing File Types	B-1
C	Pilot's Interrupt Key	C-1
D	UtilityPilot	D-1
E	Multi-national Considerations	E-1
F	References	
F.1	Mandatory references.	F-1
F.2	Informational references	F-1
Index		I-1

Table of contents

Introduction

This document defines and describes the external structure, appearance, and interfaces of *Pilot*, the operating system for the Mesa processor, and the other packages released with it. The description is primarily intended for the designers and implementors of *client programs* of Pilot, *i.e.*, applications, certain development and production tools, test programs, *etc.* It provides sufficient information to allow the programmer to understand the facilities available and to write procedure calls in the Mesa language to invoke them. For each of the facilities of Pilot, this manual lists the procedure names, parameters, results, the data types of each of the arguments, and the possible signals which can be generated. These are captured in the Mesa **DEFINITIONS** modules which are part of each release.

This manual is a reference manual for programmers, who are assumed to be familiar with the Mesa programming language. It is not a tutorial on how to write programs which use Pilot. The order of information presented, insofar as possible, tries to minimize the number of forward references. Cross referencing within the text has been abandoned for a more comprehensive referencing via the index. It is expected that the reader will go to the index to locate the description of terms or concepts encountered. References in the text of the form §1.2.3 refer to section 1.2.3. Deviations from the descriptions given here and the currently released version of Pilot are noted in the documentation which accompanies the release.

The specification presented here is adequate for the majority of programs which need to interface with Pilot and make use of its facilities. In some cases, however, supplementary facilities will be required in order to permit certain applications to make effective use of the Mesa hardware and processor. Such facilities, if made generally available, could lead to degraded performance or degraded reliability of both Pilot and the whole Mesa system. Therefore, they are not described here but are in supplementary documents which are made available, along with the corresponding **DEFINITIONS** modules, only as required.

1.1 General structure of system software

It is important to understand the relationship of the various kinds of software found in a Mesa processor. There are the following major categories:

Faces, Heads, and Microcode: A face is a Mesa interface that embodies some aspects of the processor, defined in the *Mesa Processor Principles of Operation*, and of its I/O devices. Each face is implemented by a combination of Mesa code, called a head, lower level machine code, called microcode, and the underlying hardware. The collection of heads and microcode provides a machine-independent environment in which Pilot and its clients execute.

Pilot: Pilot is the operating system which manages the hardware resources of, and provides the run-time support for, all Mesa programs on a machine. Pilot is written in the Mesa language. Its facilities are explicitly invoked by means of procedure calls from, or exceptions generated by, client programs.

Common Software: These are collections of modules and configurations which provide services often useful to applications. They are written in Mesa and call upon Pilot facilities. Some are released with Pilot while others are released separately.

Applications: Application software actually performs the functions we are marketing. These programs are written in Mesa and may call upon Pilot and Common Software for support.

In addition, there are other categories of software which are important but which will not appear in a final product as delivered to a customer. These include the Mesa compiler and binder, a number of development tools, test programs, etc. These are designed to operate in the environment of Pilot.

This document deals with Pilot, and the Common Software released with it. However, it is not possible to consider Pilot in isolation, and frequent reference must be made to documents describing the other categories of software. In particular, the Pilot facilities described here would be inadequate for supporting a modern software development project in the absence of the Mesa facilities.

1.2 Files

The basic facilities of Pilot are incorporated in the object file `PilotKernel.bcd`. There is also a special version of Pilot in the object file `UtilityPilotKernel.bcd`; this version is intended to support small applications and utilities which must run in real memory (see Appendix D for more details). Some of the facilities described in this manual are implemented in their own object files. In those cases, the name of the object file will be mentioned in the section that describes the facility.

There is no explicit mention made in this document of the location of files. That information is contained in the documentation that is issued in conjunction with each release of Pilot. Readers should consult that documentation to ascertain where files are located.

1.3 General characteristics of Pilot

Pilot is *not* a general purpose operating system. Instead, it is a nucleus of software which serves as an interface between a Mesa processor and all other software. In particular, Pilot defines a "Basic Machine" which is an abstraction of the physical resources provided by the hardware. The purpose of this Basic Machine is to define a standard interface

which is relatively independent of the size, speed, particular model, and configuration upon which it is operating. It thus provides a uniform environment in which clients can be designed and programmed. Furthermore, it insulates these as much as possible from variations in hardware configuration from site to site and from time to time.

In general, Pilot is designed around the notion that its clients are a cooperative system of programs all serving a common purpose. Thus, it is far more tolerant and permissive than most operating systems. It delegates much more control of system resources to its users. It permits programs and subsystems to recover gracefully from errors, but it also places more responsibility on them to ensure the overall well-being of the machine and of the networks to which it is connected.

The major facilities of the Basic Machine can be regarded as falling roughly into three main categories:

Mesa run-time support including processes, monitors, and synchronization facilities

Virtual memory, files, and volumes

Stream, device, and communication interfaces

Each of these categories are described below in some detail.

Some facilities and concepts normally associated with operating systems have been deliberately omitted from Pilot. For example,

Master Mode and Protection: There is no "ironclad" mechanism which protects Pilot from errant or malicious client programs, or even which protects client programs from each other. Instead, Pilot consists simply of a group of Mesa modules, and relies on such facilities as Mesa type-checking to provide the redundancy necessary to detect errors. The protection relationship between Pilot and its clients is the same as that between any two systems built in Mesa.

Job Control: Since product systems have no explicit concept of "job", Pilot provides no job control facilities. Instead, groups of related processes which support a particular application control themselves and their use of resources in response to external stimuli from the human user, or from other system elements via the Network Services (NS) Communication System.

Billing and Accounting Functions: Since the product architecture is designed around the concept of a distributed network of low cost system elements, there is no need to do detailed billing or to account for the use of resources within a single system element. In those few applications where economic management of resources is required or desired, such as in central file servers, this function is performed at a higher level, not within Pilot.

Competitive Allocation of Resources: The allocation of major system resources will generally be on a cooperative rather than a competitive basis. Thus, Pilot does not contain elaborate resource allocation functions. Instead, resources and resource management can often be planned statically when systems are configured. Where dynamic resource control is required, such as in the sharing of physical memory, Pilot provides facilities which allow the applications to state their current requirements.

Complex Services: Pilot does not provide very complex services or facilities such as directories, display and keyboard management routines, command languages, or human-engineered interfaces. These are all provided by client programs, and are likely to vary across the product lines.

1.3.1 Processes, monitors, and synchronization

Within a system element, there will almost always be several activities occurring concurrently. For example, the display will be updated at the same time as the human user is typing on the keyboard, and perhaps both of these will take place at the same time files are being read, text is being edited, or documents are being transferred to other system elements. To support this kind of concurrent activity, Mesa (with the help of the Mesa processor and Pilot) provides the following facilities:

Processes, which represent asynchronous activities,

Monitors, which arbitrate access to shared resources, and

Condition variables, which provide flexible interprocess synchronization.

These facilities are actually features of the Mesa language, but are described here for completeness.

The concept of process is a fundamental architectural concept in all Mesa software. Mesa processes are intentionally "lightweight". They are much more like Mesa procedures than, say, entire application programs. A process is instantiated in much the same way that a Mesa procedure is called. When this is done, the result is a separate, independently executing thread of control, with its own local data (if any). A process has the same status as a procedure. A process may call procedures, access local or global data, and spawn new instances of processes, subject to the standard Mesa name scoping constraints. A typical application may utilize many processes, and the whole processor may contain hundreds of process instances at one time. These can be created and deleted frequently (tens, or even hundreds of times per second if this proves useful).

The general philosophy of programming with processes in Mesa is that one or a collection of modules manages a particular resource or common data structure. Each process which needs to access that resource or data structure calls the procedures defined in those modules. To impose order on the possible chaos which could result from asynchronous manipulation of the data, the concept of *monitor lock* is provided. A monitor lock is a data structure which contains the interlocks sufficient to guarantee that only one process at a time may gain access to the data. It serves as an orderly "meeting ground" through which otherwise asynchronous processes may synchronize their activities and ensure the consistency of the data or resource which they are sharing.

In many cases, the exclusive access guarantee of the monitor mechanism is not sufficient to express the desired pattern of coordination among cooperating processes. The *condition variable* facility provides additional flexibility in synchronizing such interactions, by allowing one process to wait for some event, and another process to notify it when the event occurs. Condition variables also provide the basic means in Pilot and Mesa by which a process may wait for an event and time out after a specified period of elapsed time if that event does not occur.

In Pilot, the interfaces to sharable system resources are presented as procedures which client programs may call. These procedures almost always define *synchronous* operations, even when they involve the operation of an asynchronously operating device connected to the Mesa processor. Thus, some of them may take a long time to complete. In general, if an application program cannot tolerate such a long wait, or could make better use of its time, it should fork a new process instance to call the Pilot procedure and do the waiting for it. Later, when the results are actually required, the two process instances can be synchronized and one of them deleted. This is the general mechanism by which asynchronous activity is managed by both Pilot and client programs. The single exception to this is in the area of direct control of physical devices, in which Pilot provides a more primitive means of implementing overlapped, concurrent activity. Very few clients will be directly involved with this interface to Pilot.

1.3.2 Virtual memory, files, and volumes

Pilot provides an integrated system for managing main memory and file storage. In particular, it implements a single, monolithic, page-oriented, virtual memory shared by all Mesa software, including Pilot itself. This virtual memory consists of 2^{20} to 2^{32} 16-bit words, depending upon the hardware processor. The memory is organized into 256-word pages. To complement the virtual memory, Pilot provides a system of files, each of which may contain up to 2^{23} pages (*i.e.*, 2^{32} bytes). Files are aggregated into volumes each of which also may contain up to 2^{23} pages. Files are accessed via the virtual memory swapping mechanism, as described below.

Traditionally, virtual memories are implemented in operating systems by swapping the contents of virtual pages between real memory and some form of backing store. In Pilot, the files serve the role of backing store. Any page of virtual memory which contains information must have associated with it a page from a file to and from which it can be swapped. In the case of pages containing Mesa object code (which are always read-only), the backing file is just the object code file output by the Mesa system. In the case of virtual memory which "buffers" the contents of files containing long-term data, the files themselves act as the backing store. Finally, for pages containing temporary data which is purely internal to the current execution of the program, Pilot provides private, temporary, anonymous files for backing storage. In UtilityPilot based systems, pages for temporary data are only supplied from the processor's real memory.

Files are associated with virtual memory by *mapping* a file or portion of a file to virtual memory. The *interval* of virtual memory used is normally allocated as part of the mapping operation. Each *map unit*, or mapped interval, is typically subdivided into *swap units*, for swapping purposes, as described in the next paragraph. Pilot also provides operations to remove the mapping when it is no longer required.

Whenever a process attempts to reference (*i.e.*, fetch or store) a virtual memory location within a map unit, the page containing that location may not be present in real memory. If it is not, Pilot must read it into real memory. Execution of the process is suspended until the swapping is completed. Pilot provides swapping in two ways:

under the control of the client program, in the form of swapping commands -- these are commands by which the client program informs Pilot that certain intervals of virtual memory will be needed in the immediate future and that swapping should be initiated as soon as possible; An interval is no longer needed and should be swapped out; An

interval is not likely to be referenced soon, so Pilot should write it out and release the real memory allocated to it.

on demand -- if the page referenced is neither in real memory nor the subject of a recent swapping command to bring it in, Pilot will itself initiate a swapping action to bring in the that page and any adjoining swapped-out pages of the containing swap unit.

Typically, intervals containing code, and intervals containing local and global frames will be swapped on demand, while those which contain the major client data structures and data from files will be swapped under client program control. Swapping performance can be improved by organizing the Mesa code file(s) so that related procedures are located in the same interval of virtual memory, typically by use of the *packager*. Pilot further improves performance by attempting to allocate the pages of a file contiguously on the file storage medium so that an interval can be swapped in a single I/O operation.

A client which wishes to read from a file will map that file into a virtual memory interval and then use explicit or demand swapping to cause it to be swapped into real memory. If the file is being updated in place, the client will simply store into the relevant locations of virtual memory. Subsequently, when the interval is unmapped or otherwise swapped out of real memory, the file will reflect the new contents. If, on the other hand, the file is not being updated in place, the client program can copy the contents of a virtual memory interval to a portion of a file, and copy a portion of a file to a virtual memory interval, without altering the mapping of the interval.

Pilot supports access to files on local volumes. Each existing file is uniquely defined within that volume. If that volume is implemented on a removable medium, it (and all of its files) may be removed and remounted on another system element.

Files are identified by *file ids*. When a new file is created, a new *file id* is issued. The file is uniquely identified to Pilot by presenting Pilot with its *id* and the *id* of the containing volume. Client may not generate *file ids*, but they may store them, copy them, and pass them to other programs.

An important interval of virtual memory recognized by the Mesa processor and the Mesa system is the *main data space (MDS)*. This is a contiguous subset of virtual memory consisting of 2^{16} words (256 pages), any part of which may be addressed by a sixteen-bit Mesa **POINTER**. An MDS contains the low-level data structures and mechanisms, such as local and global frames and trap handlers, necessary for executing Mesa processes. Conversely, each process is associated with one and only one MDS. Although the Mesa processor supports multiple coexisting MDS's, Pilot does not. Thus, any Pilot-based system has only one MDS, which is shared by all of the system's processes.

1.3.3 Stream, device, and communication interfaces

Pilot supports a sophisticated, packet-switched, communication system. The heart of this system is a software package called the *router*.

Information received from one Pilot client for transmission to another Pilot client (on the same or another system element) is broken into *packets* for delivery. These packets, encapsulated in the *Xerox Internet Transport Protocols* and including both source and

destination addresses, are passed to the router. If the destination client is on the local machine, the packet is passed to that client.

For remote destination clients, the router determines if there is a communication path from the local machine to the final destination machine. If no path exists, the packet cannot be transmitted, and an appropriate status is set. Otherwise the best available path is selected and the packet is transmitted via the first *communication link* of the path on route to its final destination. This physical transmission may take place on any one of a number of communication devices, including the ethernet or telephone lines.

The router sends and receives packets via ethernet device drivers and by other communication device drivers which may be added in the future. On the Pilot client side, the router is accessed by the **NetworkStream** and **PacketExchange** interfaces (see Chapter 6).

Pilot establishes a style and some standards for the construction of I/O device drivers by defining the notion of *channel*. This makes the style of usage of the various I/O drivers similar enough to be somewhat predictable and standard enough that a client constructed I/O device driver can be included in Pilot without a formal integration. All of the Pilot-supplied and Pilot-required device drivers conform to this style and these standards.

One such Pilot-supplied device driver is the ethernet device driver. The ethernet device driver not only may be used to transmit Internet Transport Protocol packets through the router as described above, but may also be used as an ordinary device driver for non-NS communication with non-NS stations.

When *sequential* data is to be transported between a Pilot client and an I/O device or another Pilot client, it is usually possible to do this in a device and format independent way. The Pilot *Stream Package* accomplishes this. The mechanism for transcribing a sequential stream of data on or off an I/O device is provided by a client written or Pilot-supplied *transducer*. Modifications to the data stream (e.g., code conversion) are accomplished by a client or Pilot *filter*. The stream package provides a basic set of transducers and filters and, more important, a way of assembling them sequentially into processing and transmitting *pipelines*.

One kind of stream supported directly by Pilot is the Network stream referred to above. This kind of stream is capable of receiving data from a Pilot client on one machine and transmitting it to another client on a different machine.

1.4 Pilot concepts

There are methodologies which are used repeatedly in the design of the Pilot functions. They are described here.

1.4.1 Stateless enumerators

Many Pilot functions return information to the client of the form of a list of items whose length cannot be *a priori* known. Consequently, Pilot functions that supply this type of information do so by passing back an item of the list for each call for the information. These functions are created in a very stylized way.

The basic idea is that the client, on its first call to such a function, supplies a value which no item of the list can have. This item is usually has a name of the form **nullobject**, for

whatever object is being enumerated. The function returns a member of the list. If the client, on its next call on the list function, supplies the previously returned value, Pilot will return another member of the list. This goes on until the list is exhausted where upon Pilot returns `nullobject`, indicating the end of the list.

These types of functions are called *stateless enumerators*. A reference to a stateless enumerator will always be accompanied by the beginning and ending values. Usually the items of the list are not returned in any particular order. If there is some order imposed, this will be pointed out in the description of the function.

1.4.2 Synchronous and asynchronous operations

When a Pilot function is called, it may or may not return before the requested operation has been completed. If Pilot waits until the operation is done (the usual case), the operation is called *synchronous*. If the operation queues the operation and returns before it has completed, it is dubbed *asynchronous*. If no mention is made of the type of a particular operation, the operation is synchronous. Almost all Pilot operations are synchronous.

1.5 Notation and conventions

At the beginning of each section are listed the names of the **DEFINITIONS** modules containing the Pilot facilities described in that section. The procedure and type definitions contained in each of the interface modules are presented in this document as pseudo-Mesa declarations of the form:

ModuleName.TypeName: TYPE = ...;

ModuleName.ProcedureName: PROCEDURE [ParameterList] RETURNS [ResultsList];

ModuleName.SignalName: SIGNAL [ParameterList] RETURNS [ResultsList];

That is, each definition is listed with its own name qualified by the **DEFINITIONS** module name. Any Mesa program which invokes the facilities of Pilot must list the names of the relevant **DEFINITIONS** modules in its **DIRECTORY** clause. It may then refer to one of these variables, procedures, types, or signals by its fully, qualified name. This style of explicit qualification is *strongly recommended* (*i.e.*, as opposed to opening the scope of the **DEFINITIONS** module by an **OPEN** clause, and using the unqualified name).

Accompanying these Mesa declarations is the explanation of the function of each procedure, the conditions under which it may be invoked, and the **SIGNALS** and **ERRORS** it can raise. In this explanatory text, the explicit interface qualification is usually dropped, since it is clear from the context.

The following rules apply to all the operations discussed in this manual. Exceptions to the rules will be mentioned explicitly.

- 1) If the explanatory text of an operation does not explicitly say that a specific error is raised, then the operation does not raise the error.

- 2) If an operation returns by raising an error, then the operation will appear to have only raised the error.
- 3) If an operation is to operate on a object already operated on (e.g., Space.MakeReadOnly on a read-only object), then the operation will return successfully. That is, most operations are idempotent.
- 4) All operations that may be performed outside the body of a catch phrase, may be performed within the body of the catch phrase (e.g., Pilot holds no monitor locks while raising a signal or error).
- 5) Invoking an operation with a count parameter of zero, is equivalent to invoking the operation with a count of one minus one (i.e., zero is not a special case).

Note: A paragraph in this form headed by the word "Note" contains additional information about how the operations are intended to be used. These are included to help the programmer to design his program to take best advantage of the Pilot facilities. Ignoring these notes will not produce incorrect programs, but it may produce programs that execute slowly, or require excessive amounts of system resources.

Caution: Paragraphs labeled with "Caution" are intended as warnings to programmers. In general, these apply to features or aspects of Pilot which can be easily misused, and which will result in incorrect or inconsistent operation if they are misused. *In particular, Pilot is not likely to be able to detect errors cautioned against in these paragraphs. It is the programmer's responsibility to avoid making these mistakes.*

For example, an error which Pilot cannot detect is the "dangling reference" problem. In many cases, Pilot defines a class of abstract objects and provides client programs *handles* for accessing such objects. If one client program should request Pilot to destroy a particular object, then later another client program requests Pilot to create a new one of the same type, Pilot *may* reuse the handle of the old, destroyed one. If the first client program inadvertently retains and uses copies of the old handle, these will now look like legitimate handles for the new object. Pilot may not be able to detect the condition and chaos is likely to ensue.

Metasymbols are indicated with italics. It is expected that some specific instance will be filled in for the metasymbol, such as in the case of *nullObject* in the preceding section. A possible instance of a *nullObject* might be *nullHandle*.

1.6 Common Software

This manual also includes descriptions of the Common Software. Common Software is not included in PilotKernel.bcd, but is made available as separate object files. Clients which make no use of Common Software need not be burdened with its presence. Common Software comes in two varieties, Product and Development. Those Common Software packages denoted as Product Common Software are intended to be used in products. Development Common Software consists of packages that are used internally, in the development environment; they should not be used in product systems. Only Product Common Software is described in this manual.

Because the Common Software packages are not included in PilotKernel.bcd, the name of the implementing object file, how to bind, etc. is presented at the beginning of each section describing a Common Software package.

1.7 What follows

The rest of the manual describes the interfaces to Pilot and the Common Software packages in terms of the Mesa data types and procedures used by clients. These types and procedures are embodied in one or more Mesa interfaces (**DEFINITIONS** modules) made available to programmers of client software. The description is organized according to the major resources managed by Pilot.

Chapter 2 describes the interface provided by Pilot to various Mesa processor features. Described are the various constants and types associated with the processor. It also describes the run-time support needed to execute Mesa programs. This chapter includes the descriptions of facilities to support the Mesa concepts of *process*, *monitor*, and *condition variable* and the various traps, procedures, and signals defined by the Mesa language. It describes some basic, low-level system facilities provided by Pilot. These include: *universal identifiers*, by which volumes and other objects are named; *network addresses*, which control communication via the Xerox Internet Transport Protocols; several forms of timekeeping facilities; and facilities for controlling system electrical power.

In Chapter 3, the general concept of a *stream* is introduced. Streams may be superimposed upon files, communication facilities, and devices in order to achieve a high level, medium independent means of accessing and distributing information.

Chapter 4 describes the file management and virtual memory facilities of Pilot.

Chapter 5 describes the facilities by which client software can exercise control over hardware devices. These facilities are meant primarily for situations in which streams are not suitable. This chapter is a model for individual device interfaces, some of which are described in this manual, and others of which are implemented by clients.

Chapter 6 describes the communication facilities of Pilot.

Chapter 7 describes miscellaneous editing and formatting packages.

Chapter 8 describes how to initialize the system, and how to get a client to start execution.

Chapter 9 describes facilities for automatically handling system errors and signals. The processing of error conditions is done by a separate program referred to generically as a *backstop*.

Environment

This chapter describes the constants, types, and procedures, available to the Pilot programmer, which describe the system element and make available at the client level, certain features of the abstract machine. It contains the basic levels of the system.

2.1 Processor environment

Environment: **DEFINITIONS** . . . ;

This section captures all of the basic constants describing the processor and peripherals. The first section describes the processor and the second defines the constants pertinent to the peripheral devices attached to the processor.

2.1.1 Basic types and constants

Pilot is specifically designed to execute on system elements defined by the *Mesa Processor Principles of Operation*. For convenience, the basic types and constants of that architecture are captured symbolically in the **DEFINITIONS** module **Environment**.

The following definitions define the basic word, byte and character sizes of the Mesa processor.

Environment.Byte: **TYPE** = [0..255];

Environment.Word: **TYPE** = [0..65535];

Environment.bitsPerWord: **CARDINAL** = 16;

Environment.bitsPerByte, Environment.bitsPerCharacter: **CARDINAL** = 8;

Environment.logBitsPerWord: **CARDINAL** = 4;

Environment.bytesPerWord, Environment.charsPerWord: **CARDINAL** =
bitsPerWord / bitsPerCharacter;

Environment.logBitsPerByte, Environment.logBitsPerChar: **CARDINAL** = 3;

Environment.logBytesPerWord, Environment.logCharsPerWord: CARDINAL = 1;

All constants of the form **log...** are base 2 logarithms of their respective quantities. The following type is a general purpose descriptor for a sequence of bytes in virtual memory (see section §4.5 for a description of virtual memory).

```
Environment.Block: TYPE = RECORD[
  blockPointer: LONG POINTER TO PACKED ARRAY [0..0] OF Environment.Byte,
  startIndex, stopIndexPlusOne: CARDINAL];
```

The following constant defines an empty block.

Environment.nullBlock: Environment.Block = [NIL, 0, 0];

The following definitions characterize the basic page size of the Mesa processor.

Environment.wordsPerPage: CARDINAL = 256;

Environment.bytesPerPage, Environment.charsPerPage: CARDINAL = wordsPerPage * bytesPerWord;

Environment.logWordsPerPage: CARDINAL = 8;

Environment.logBytesPerPage, Environment.logCharsPerPage: CARDINAL = logWordsPerPage + logBytesPerWord;

The following definitions characterize the maximum virtual memory address space available to Pilot clients.

Environment.maxPagesInVM: CARDINAL = Environment.lastPageCount;

This is one less than the number of VM pages provided by the hardware. The highest numbered VM page is reserved for system purposes.

Environment.maxPagesInMDS: CARDINAL = 256;

Environment.PageNumber: TYPE = LONG CARDINAL; --[0..2²⁴-1]--

Environment.firstPageNumber: Environment.PageNumber = 0;

Environment.lastPageNumber: Environment.PageNumber = 16777214; --2²⁴-2--

Note: Because **LONG** subrange types are not implemented in the current version of Mesa, the current version of Pilot defines **PageNumber** as a **LONG CARDINAL** and defines the constants **firstPageNumber** and **lastPageNumber** to specify **FIRST[PageNumber]** and **LAST[PageNumber]**. Similarly for **PageCount** and **PageOffset** below.

Environment.PageCount: TYPE = LONG CARDINAL --[0..2²⁴-1]--;

Environment.firstPageCount: Environment.PageCount = 0;

Environment.lastPageCount: Environment.PageCount = lastPageNumber + 1; -- 2²⁴-1

Environment.PageOffset: TYPE = Environment.PageNumber;

Environment.firstPageOffset: Environment.PageOffset = 0;

Environment.lastPageOffset: Environment.PageOffset = lastPageNumber;

Caution: Substituting LAST[Environment.PageNumber] or LAST[Environment.PageCount] for the above constants will yield incorrect results.

Environment.Base: TYPE = LONG BASE POINTER;

Environment.first64K: Environment.Base = ...;

first64K is the base pointer to the first 64K of virtual memory.

Environment.maxINTEGER: INTEGER = LAST[INTEGER];

Environment.minINTEGER: INTEGER = FIRST[INTEGER];

Environment.maxCARDINAL: INTEGER = LAST[CARDINAL];

Environment.maxLONGINTEGER: INTEGER = LAST[LONG INTEGER];

Environment.minLONGINTEGER: INTEGER = FIRST[LONG INTEGER];

Environment.maxLONGCARDINAL: INTEGER = LAST[LONG CARDINAL];

The following types allow direct manipulation of long values.

Environment.Long, Environment.LongNumber: TYPE = MACHINE DEPENDENT

```
RECORD [SELECT OVERLAID * FROM
  lc => [lc: LONG CARDINAL],
  li => [li: LONG INTEGER],
  lp => [lp: LONG POINTER],
  lu => [lu: LONG UNSPECIFIED],
  num => [lowbits, highbits: CARDINAL],
  any => [low, high: UNSPECIFIED],
  ENDCASE];
```

The following structure is used to address bits (used principally by BitBlt).

Environment.BitAddress: TYPE = MACHINE DEPENDENT RECORD [

```
word: LONG POINTER,
reserved: [0..LAST[WORD]/Environment.bitsPerWord) ← 0,
bit: [0..Environment.bitsPerWord)];
```

Note that the reserved field must be zero.

The following operation returns a LONG POINTER to the first word of a page.

Environment.LongPointerFromPage: PROCEDURE [page: Environment.PageNumber]
 RETURNS [LONG POINTER];

The following operation returns the number of the page containing pointer. If pointer is **NIL**, the value returned is undefined—no signal is raised.

```
Environment.PageFromLongPointer: PROCEDURE [pointer: LONG POINTER]
RETURNS [Environment.PageNumber];
```

2.1.2 Device numbers and device types

```
Device: DEFINITIONS . . . ;
```

```
DeviceTypes: DEFINITIONS . . . ;
```

Definitions are provided for devices and classes of devices attached to the system element. These constants are defined in the interfaces **Device** and **DeviceTypes**. Definitions in the interface **Device** serve to identify the individual devices attached to the system element.

```
Device.Type: TYPE = RECORD [CARDINAL];
```

```
Device.nullType: Device.Type = [0];
```

```
Device.Ethernet: TYPE = CARDINAL [5..16];
```

```
Device.PilotDisk: TYPE = CARDINAL [64..1024];
```

All Ethernet type devices will have a value in the range defined by **Ethernet**. All devices capable of containing a Pilot physical volume will be in the range defined by **PilotDisk**.

Device types provide a means of classifying the different devices attachable to the system element. Device types for Ethernet devices are:

```
DeviceTypes.anyEthernet: Device.Type = . . . ;
```

```
DeviceTypes.ethernet: Device.Type = . . . ;
```

```
DeviceTypes.ethernetOne: Device.Type = . . . ;
```

A type of **anyEthernet** indicates that the device is an Ethernet but of unspecified type. A type of **ethernet** indicates that the device is a 10 megabit Ethernet. A type of **ethernetOne** indicates that the device is a 3 megabit Ethernet.

The specific device types assigned to Pilot disks are:

```
DeviceTypes.anyPilotDisk: Device.Type = . . . ;
```

```
DeviceTypes.sa1000: Device.Type = . . . ;
```

```
DeviceTypes.sa1004: Device.Type = . . . ;
```

```
DeviceTypes.sa4000: Device.Type = . . . ;
```

```
DeviceTypes.sa4008: Device.Type = . . . ;
```

```
DeviceTypes.t300: Device.Type = ...;  
DeviceTypes.t80: Device.Type = ...;  
DeviceTypes.cdc9730: Device.Type = ...;  
DeviceTypes.q2000: Device.Type = ...;  
DeviceTypes.q2010: Device.Type = ...;  
DeviceTypes.q2020: Device.Type = ...;  
DeviceTypes.q2030: Device.Type = ...;  
DeviceTypes.q2040: Device.Type = ...;  
DeviceTypes.q2080: Device.Type = ...;
```

A type of **anyPilotDisk** indicates that the device is a Pilot disk but of unspecified type.

A type of **sa1000** indicates that the device is some unspecified disk of the Shugart Associates SA1000 family. Similarly, a type of **sa4000** indicates that the device is some unspecified disk of the Shugart Associates SA4000 family. A type of **sa1004** indicates that the device is an SA1004 disk, a type of **sa4008**, an SA4008 disk.

A type of **t300** indicates that the device is a Century Data Systems T-300 disk. A type of **t80** indicates that the device is a Century Data Systems T-80 disk. A type of **cdc9730** indicates that the device is a Control Data Corporation CDC-9730 disk.

A type of **q2000** indicates that the device is some unspecified disk of the Quantum 2000 family. Types of **q2010**, **q2020**, **q2030**, **q2040**, and **q2080** indicate Quantum disk devices of type 2010, 2020, 2030, 2040, and 2080, respectively.

Other device types included in the interface are:

```
DeviceTypes.null: Device.Type = Device.nullType;  
DeviceTypes.sa800: Device.Type = ...;
```

A type of **sa800** indicates that the device is some unspecified disk of the Shugart Associates sa800 family.

When indicating devices capable of holding a Pilot volume, Pilot will report a correct device type, although it may not be as specific as possible (i.e., a Shugart SA4008 disk might be reported as either **DeviceTypes.anyPilotDisk** or **DeviceTypes.sa4000** or **DeviceTypes.sa4008**).

The following **Extras** interfaces are interim for this release. In future releases, all **Extras** interfaces will be merged with their parent interfaces.

```
DeviceTypesExtras.anyFloppy: Device.Type = ...;
```

```

DeviceTypesExtras.Sa850: Device.Type = ...;

DeviceTypesExtras.Sa455: Device.Type = ...;

DeviceTypesExtras.Sa456: Device.Type = ...;

DeviceTypesExtraExtras.m2235: Device.Type = ...;

DeviceTypesExtraExtras.m2242: Device.Type = ...;

DeviceTypesExtraExtras.m2243: Device.Type = ...;

```

A type of **anyFloppy** indicates that the device is a floppy drive but of unspecified type.

A type of **sa850** indicates that the device is a Shugart Associates SA-850 floppy drive. Similarly, a type of **sa455** or **sa 456** indicates that the device is a Shugart SA-455 or SA-456 floppy drive, respectively.

A type of **m2235** indicates that the device is a Fujitsu 26MB rigid disk drive. Similarly, a type of **m2242** or **m2243** indicates that the device is a Fujitsu 50MB or 80 MB rigid disk drive, respectively.

2.2 Processor interface

This section presents interfaces, provided by Pilot, that permit access to features provided by the underlying Mesa processor which are not provided by the Mesa language. These interfaces define pseudo-faces—types defined by the hardware and operations directly implemented by the hardware. Pilot merely exports the definitions for the use of its clients. The types and operations are defined below.

2.2.1 BitBlt

BitBlt: DEFINITIONS ...;

The Bit Block Transfer operation in this interface is **BITBLT** which operates on rectangular arrays of bits in memory. The instruction accesses source bits and destination bits, performs a function on them, and stores the result in the destination bits.

Successive bit pairs are obtained by scanning a source bit stream and a destination bit stream. The instruction operates successively on lines of bits called *items*; it processes **width** bits from a pair of lines, and then moves down to the next item by adding **srcBpl** (bits per line) to the source address and **dstBpl** to the destination address. It continues until it has processed **height** lines.

Figure 2.1 illustrates a possible configuration of source and destination rectangles, which are always of the same size and dimensions, embedded in separate bitmaps. Approximately half of the items have been moved to the destination, and the location of the next item is highlighted in the source bitmap and shown as a dotted line in the destination bitmap.

BitBlt.BITBLT: PROCEDURE [ptr: BBptr]

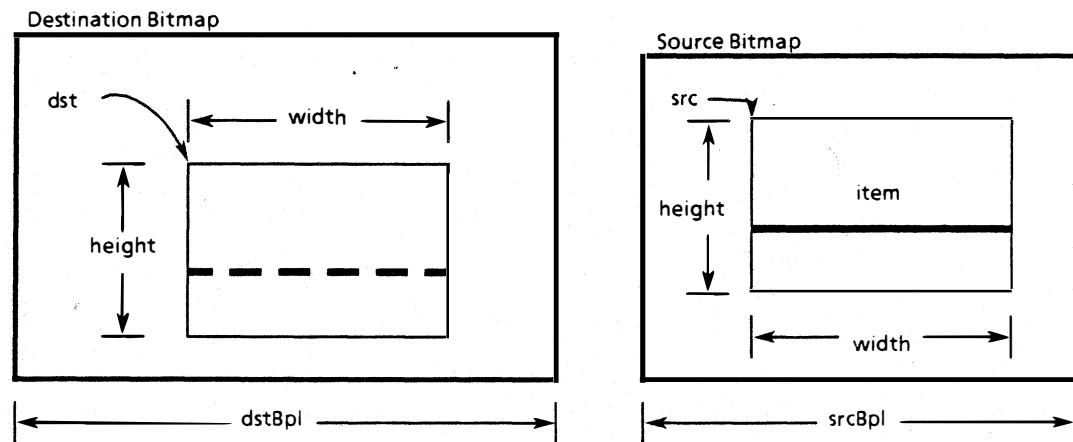


Figure 2.1 BitBlit Source and Destination

The argument to Bit Block Transfer is a short pointer to a record containing the source and destination bit addresses and bits per line, the width and height (in bits) of the rectangle to be operated on, and a word of flags that indicate the operation to be performed. The width and height of the rectangle are restricted to a maximum of 32,767. The argument record must be aligned on a sixteen word boundary.

```

BitBlit.AlignedBBTable: PROCEDURE [ip: POINTER TO BBTableSpace] RETURNS [b: BBptr];
BitBlit.BBTableSpace: TYPE = ARRAY [1..SIZE[BBTable] + BBTableAlignment) OF UNSPECIFIED;
BitBlit.BBTableAlignment: CARDINAL = 16;
AlignedBBTable ensures that the BBTable will be on a sixteen word boundary.

BitBlit.BBptr, BitBlit.BitBlitTablePtr: TYPE = POINTER TO BBTable;
BitBlit.BBTable, BitBlit.BitBlitTable: TYPE = MACHINE DEPENDENT RECORD [
  dst: BitAddress,
  dstBpl: INTEGER,
  src: BitAddress,
  srcDesc: SrcDesc,
  width: CARDINAL,
  height: CARDINAL,
  flags: BitBlitFlags,
  reserved: UNSPECIFIED ← 0];

```

This table contains all the arguments for specifying the resultant bit pattern. The following types are used to make up a **BitBlitTable (BBTable)**.

BitBlit.BitAddress: TYPE = Environment.BitAddress;

BitAddress is used to address bits.

```
BitBlt.SrcDesc: TYPE = MACHINE DEPENDENT RECORD [
  SELECT OVERLaid * FROM
  gray = > [gray: GrayParm],
  srcBpl = > [srcBpl: INTEGER],
  ENDCASE];
```

The description of the source may be a pattern to be repeated or may be particular bits. In the case of a pattern, the *gray* field would be selected. This is described in detail under *Gray Flag* following.

```
BitBlt.BitBltFlags: TYPE = MACHINE DEPENDENT RECORD[
  direction: Direction ← forward,
  disjoint: BOOLEAN ← FALSE,
  disjointItems: BOOLEAN ← FALSE,
  gray: BOOLEAN ← FALSE,
  srcFunc: SrcFunc ← null,
  dstFunc: DstFunc ← null,
  reserved: [0 511 0];
```

Direction Flag

The direction flag indicates whether the operation should take place forward (left to right, from low to high memory addresses) or backward (right to left, from high to low memory addresses). This allows an unambiguous specification of overlapping BitBlts, as in scrolling.

```
BitBlt.Direction: TYPE = {forward, backward};
```

If the direction is backward, the source and destination addresses point to the beginning of the *last* item of the blocks to be processed, and the source and destination bits per line must be *negative*. This restricts the width of the bitmaps involved to a maximum of 32,767 bits.

Disjoint Flag

If the operation's source and destination are completely disjoint, the implementation performs the operation from left to right, top to bottom.

Both the *direction* and the *disjointItems* flags in the argument record are ignored in the case that *disjoint* is set.

DisjointItems Flag

If the individual items of the source and destination are disjoint, but the rectangles otherwise overlap, the *disjointItems* flag should be set (and the *disjoint* flag should be clear); this allows the implementation to perform the operation so that, within each item, the bits are processed in the most efficient horizontal direction. The items are processed in the order indicated by *direction*.

If neither *disjoint* nor *disjointItems* is set, the implementation processes the items and the bits within items in the direction indicated by the *direction* flag.

Gray Flag

The **gray** flag allows repetitive bit patterns to be specified in a condensed format. The usual application is for generation of various shades of gray on the display, but any repetitive pattern within the limits stated below may be supplied.

If the **gray** option is specified, the **srcBpl** field of the argument record is reinterpreted as follows: Note also that the **gray** case is always **forward** and completely **disjoint** (**disjointItems** is ignored).

```
BitBlt.GrayParm: TYPE = MACHINE DEPENDENT RECORD [  
    reserved: [0..15] ← 0,  
    yOffset: [0..15],  
    widthMinusOne: [0..15],  
    heightMinusOne: [0..15]];
```

The fields **grayParm.widthMinusOne** and **grayParm.heightMinusOne** define the width (less one) in words and height (less one) in bits, respectively, of a gray brick located at **arg.src**. (see figure 2.2). Note, the term "brick" refers to a rectangular area containing the gray pattern to be copied. Conceptually, this brick is replicated horizontally and vertically to tile a plane of dimensions **arg.width** and **arg.height** that becomes the source rectangle of the operation. This brick is a maximum of sixteen words wide and sixteen lines high. Patterns, therefore, are also limited to a repetition rate of sixteen in each direction. To guarantee correct repeatability of the pattern in the horizontal direction, it is usually the case that the width of the gray brick (in bits) is a multiple of the repetition rate; the height of the gray brick is usually equal to the vertical repetition rate.

Proper alignment of the gray pattern with the destination bitmap requires the initial x and y offsets into the brick in addition to its width and height. The initial x offset is derived from **arg.src** as follows: **arg.src.word** always points to the beginning of the first line to be transferred (not to the origin of the gray brick). The x offset of the first bit to be transferred is supplied by **arg.src.bit**; this bit is always in the first word of the line. The initial y offset is the number of lines down from the origin of the brick and is specified by **grayParm.yOffset**; subtracting the y offset times the brick width from **arg.src.word** gives the origin of the gray brick.

Source and Destination Functions

```
BitBlt.SrcFunc: TYPE = {null, complement};
```

```
BitBlt.DstFunc: TYPE = {null, and, or, xor};
```

The functions available for combining the source and destination rectangles are shown in Figure 2.3.

The **src** field has two options; the **null** selection indicates using the source rectangle as is for the destination function. The **complement** selection will invert the source bits in the destination function.

The **dst** field determines the function to be used for changing bits in the destination rectangle. The **null** selection causes the destination to be "replaced" with the source bits. There is no boolean operation in this case. **Anding** the destination bits with the source bits

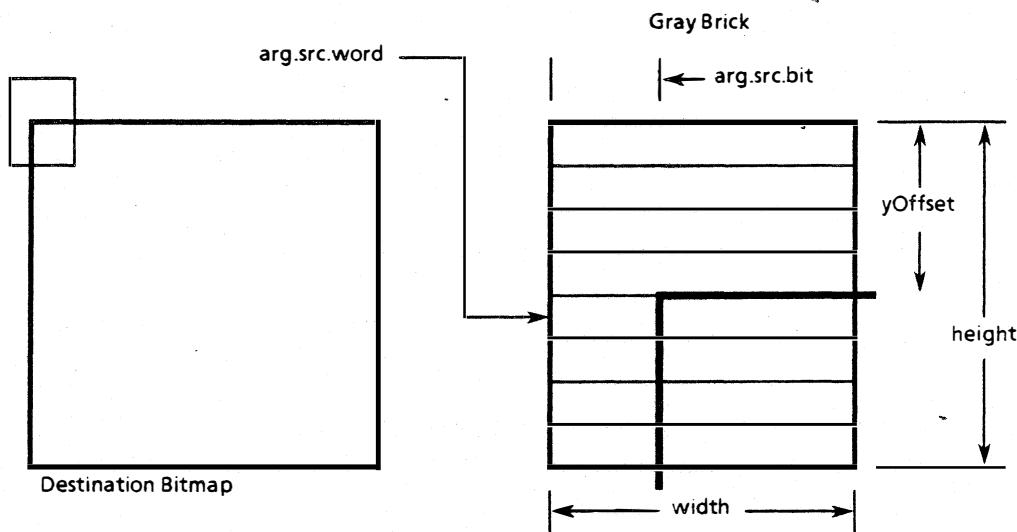


Figure 2.2 Gray Brick

		n	a	o	x
		s	s·d	s+d	s⊕d
src	n				
	c	~s	~s·d	~s+d	~s⊕d

Figure 2.3 Source and Destination Functions

leaves only those bits in common in the destination. "Painting" the destination requires **oring**. This will leave the union of the two sets of bits in the destination. The last function is the **xor**. This essentially masks out the matching bits leaving the union but not the intersection of the bits in the destination rectangle.

2.2.2 TextBlt

TextBlt:DEFINITIONS...;

The Text Block Transfer interface operates on an array of characters; it implements three functions useful in generating the font representation of the text in a bitmap. It may calculate the number of characters on a line, convert characters to their font representation, or widen or narrow select characters for justification. There is more discussion on these functions later in this section.

TextBlt.TextBlt: PROCEDURE [

index: CARDINAL, bitPos: CARDINAL, micaPos: CARDINAL, count: INTEGER,
ptr: POINTER TO TextBltArg]

```
RETURNS [
newIndex: CARDINAL, newBitPos: CARDINAL, newMicaPos: CARDINAL,
newCount: INTEGER, result: Result];
```

TextBlt proceeds through the text until either there is no more text or a stop character is encountered; it maintains the **bitPos** and the **micaPos** of the origin of each character, and increments the **count** of the number of pad characters processed. The new character positions are returned along with the **result** of what caused the completion.

```
TextBlt.TextBltArgAlignment: CARDINAL = 16;
```

```
TextBlt.TextBltArgSpace: TYPE = ARRAY [1..SIZE[TextBltArg] + TextBltArgAlignment) OF
UNSPECIFIED;
```

```
TextBlt.AlignedTextBltArg: PROCEDURE [ip: POINTER TO TextBltArgSpace]
RETURNS [p: POINTER TO TextBltArg]
```

TextBlt's static arguments are passed via a short pointer to a record; the argument record must be aligned on a sixteen word boundary.

```
TextBlt.TextBltArg: TYPE = MACHINE DEPENDENT RECORD [
reserved: [0..37777B] ← 0,
function: Function, -- display, format or resolve
last: CARDINAL, -- index of last character to process
text: LONG POINTER TO PACKED ARRAY CARDINAL OF CHARACTER,
font: FontHandle, -- Long Pointer to font information
dst: LONG POINTER, -- destination bitmap (display only)
dstBpl: CARDINAL, -- Bits per line (display only)
margin: CARDINAL, -- mica value of right margin (format only)
space: INTEGER, -- width adjustment to pad characters (display, resolve)
coord: LONG POINTER TO ARRAY CARDINAL [0..0] OF CARDINAL -- widths array for resolve
];
```

The limits of the text that TextBlt operates on are **arg.text** to **arg.last**. Depending on the function specified (explained below) specific args will be pertinent. During the **format** function, the scan is terminated before the right **arg.margin** (in micas) is passed. The **display** function Ors the character's font bits into the destination bitmap specified by **arg.dst** and **arg.dstBpl** (bits per line). The **resolve** function saves the **bitPos** of the origin of each character in the array **arg.coord**.

Justification can be accomplished using the display and resolve functions with appropriate settings of the **arg.space** and **count** values; **arg.space** is added to the width of every pad character (it may be negative), and **count** is incremented each time a pad character is encountered (it may also be initially negative). Since the amount of white space to be absorbed by (or squeezed out of) pad characters is rarely an even multiple of the number of pad characters, pad characters encountered have **arg.space + 1** added to their widths as long as **count** is negative. Thus if sixteen bits need to be added to the width of the line in order to justify it, but it contains only thirteen pad characters, **arg.space** would be set to one, and **count** would be initialized to *negative* three; this will result in widening the first three pad characters by two bits, and the remaining ten pad characters by one bit each.

```
TextBlt.Function: TYPE = {display, format, resolve};
```

The **TextBlockTransfer** implements three functions useful in generating the font representation of the text in a bitmap. The **format** function is used to calculate the number of characters that will fit on a line, given its right margin (in micas). The **display** function converts characters to their font representation in the destination bitmap, optionally widening or narrowing pad characters to perform line justification. The **resolve** function is used to record the horizontal bit position of the origin of each character in the bitmap; it also handles justification.

Caution: Because of kerning, the display function may place bits into the destination bitmap to the left of the **bitPos** of the leftmost character and to the right of the right margin. It is the programmer's responsibility to initialize the **bitPos** to allow for the left kerning of the first character, and to supply a bitmap wide enough to allow for the maximum possible right kerning. Kerning is further explained below.

```
TextBlt.FontHandle: TYPE = LONG POINTER TO Font;
```

```
TextBlt.Font: TYPE;
```

TextBlt.FontHandle points to the font information **TextBlt** needs. The interface **Fonts** describes the **TextBlt** font type. **TextBltFontFormat.FontRecord** is the concrete type of a **TextBlt.Font**. **TextBltFontFormat.FontRecord** must be aligned on a sixteen-word boundary.

```
TextBltFontFormat.FontRecordAlignment: NATURAL = 16;
```

```
TextBltFontFormat.FontRecord: TYPE = MACHINE DEPENDENT RECORD [
    fontbits(0): FontBitsPtr,
    fontwidths(2): FontWidthsPtr,
    fontchar(4): FontCharPtr,
    rgflags(6): RgFlagsPtr,
    height(8): CARDINAL];
```

The following types make up **FontRecord**:

```
TextBltFontFormat.FontBitsPtr: TYPE = LONG BASE POINTER TO ARRAY [0..0] OF UNSPECIFIED;
```

The data at **TextBltFontFormat.FontBitsPtr** is a base pointer for the character raster data. For a particular character, **TextBltFontFormat.CharEntry.offset** (defined below) is added to this base to get the address of the character's raster. The raster format includes the scan lines within the dimensions given by **fontwidths** and **fontchar**. The **height** of the raster is constant for all characters.

The memory order of the bits in the raster correspond to the memory order that **TextBlt** will paint them into the destination bitmap. Said another way, **TextBlt** paints the first scan line of the raster into the appropriate place in the first scan line of the destination bitmap, and so on. Similarly, the first bit of a raster's scan line is painted into the appropriate first bit of the scan line in the destination bitmap, and so on.

In conventional Xerox bitmap displays, the first scan line in memory corresponds to the top line on the screen, and the first bit of a scan line corresponds to the left pixel of the line.

For this case, the first scan line in the raster will be the topmost row of the character, and the first pixel (most significant bit) of a scan line will be the leftmost pixel of its row.

TextBitFontFormat.FontWidthsPtr: TYPE = LONG POINTER TO FontWidths;

TextBitFontFormat.FontWidths: TYPE = PACKED ARRAY CHARACTER OF PixelWidth;

TextBitFontFormat.PixelWidth: TYPE = CARDINAL [0..377B];

The width of the font is dependent on the width of the pixel.

TextBitFontFormat.FontCharPtr: TYPE = LONG POINTER TO FontChar;

TextBitFontFormat.FontChar: TYPE = ARRAY CHARACTER OF CharEntry;

CharEntry must be aligned on a two-word boundary.

TextBitFontFormat.CharEntryAlignment: NATURAL = 2;

TextBitFontFormat.CharEntry: TYPE = MACHINE DEPENDENT RECORD [

leftKern(0:0..0): BOOLEAN,
rightKern(0:1..1): BOOLEAN,
offset(0:2..15): RasterOffset,
mica(1): CARDINAL];

If **CharEntry.leftKern** = TRUE, the character's raster has one column preceding the char's origin, and is to be written into the destination bitmap one column preceding the current position (**bitPos**). If **CharEntry.rightKern** = TRUE, the raster extends one column past the spacing width into the space for the next char; that char's raster should begin coincident with the current char's last column (one column preceding where it would normally go).

CharEntry.offset is the offset for the address of the character's raster.

TextBitFontFormat.RasterOffset: TYPE = CARDINAL [0..37777B];

Mica indicates the "physical" width of the char (typically in micas).

TextBitFontFormat.RgFlagsPtr, RgflagsPtr: TYPE = LONG POINTER TO RgFlags;

TextBitFontFormat.RgFlags: TYPE = PACKED ARRAY CHARACTER OF Flags;

TextBitFontFormat.Flags: TYPE = MACHINE DEPENDENT RECORD [
pad(0:0..0): BOOLEAN,
stop(0:1..1): BOOLEAN];

The **pad** flag allows the character to have its width increased or decreased (in bits) for line justification. The **stop** flag will specify a stop character to terminate a **TextBit** operation.

TextBitFontFormat.maxLeftKern: CARDINAL = 1;

TextBitFontFormat.maxRightKern: CARDINAL = 1;

MaxLeftKern and **maxRightKern** support kerning up to one pixel in the respective direction.

TextBlt.Result: TYPE = {normal, margin, stop};

TextBlt returns, in place of the argument pointer on the stack, an indication of its completion condition: *normal* if the last character was processed, *margin* if the right margin was reached (format only), and *stop* if a terminating character was detected.

notInFont is returned if the printer width for the character is a distinguished value (177777B). This allows the flags to be independent of the font and yet provides a way for information in the font to cause **TextBlt** to terminate.

```
TextBlt.SoftwareTextBlt: PROCEDURE [
    index: CARDINAL, bitPos: CARDINAL, micaPos: CARDINAL, count: INTEGER,
    ptr: POINTER TO TextBltArg]
RETURNS [
    newIndex: CARDINAL, newBitPos: CARDINAL, newMicaPos: CARDINAL,
    newCount: INTEGER, result: Result];
```

SoftwareTextBlt is a software version of **TextBlt**. It is useful on processors that do not have microcode support for the **TextBlt** operation described in this section.

2.2.3 Checksum

Checksum:DEFINITIONS ...;

This interface produces a checksum for **nWords** starting at **p**. Changing the initial value **cs** is useful if forming a single checksum for discontinuous areas of memory.

```
Checksum.ComputeChecksum: PROC [cs: CARDINAL ← 0, nWords: CARDINAL, p: LONG POINTER]
RETURNS [checksum: CARDINAL];
```

Checksum.nullChecksum: CARDINAL = 177777B;

This is a ones-complement add-and-left-cycle algorithm.

2.2.4 ByteBlt

ByteBlt: DEFINITIONS ...;

The only operation in this interface is **ByteBlt** which provides a Mesa definition of a byte boundary block transfer operation. It takes descriptions of two byte blocks as arguments, transfers as many bytes as possible (the **MIN** of the two lengths), and returns a count of how many bytes were actually moved.

```
ByteBlt.ByteBlt: PROCEDURE [to, from: Environment.Block,
    overLap: ByteBlt.OverLapOption]
RETURNS [nBytes: CARDINAL];
```

ByteBlt.OverLapOption: TYPE = {ripple, move};

ByteBlt.StartIndexGreater Than StopIndexPlusOne: ERROR;

A length of zero in either **to** or **from** is acceptable, resulting in no transfer. If a negative length (**startIndex > stopIndexPlusOne**) is present in either **to** or **from**, **ByteBlt** signals **ByteBlt.StartIndexGreater Than StopIndexPlusOne**.

The **overLap** argument defines the effect of **ByteBlt** when the source and destination fields overlap. If **overLap** is **move** then the contents of the source field are preserved by the move. It acts as if the two fields did not overlap. If **overLap** is **ripple** then a low address to high address move takes place with no notice taken of overlapping fields. This mode is useful for propagating a value throughout a block of storage.

2.2.5 Other Mesa machine operations

Inline: DEFINITIONS ...;

This interface defines a set of instructions not directly accessible from Mesa. It includes some logical instructions and some extended-precision arithmetic instructions.

2.2.5.1 Accessing parts of a word or double word

The type **Environment.LONG** allows direct access to the high-order and low-order words of **LONG** values. For convenience, a copy of this type is available in the **Inline** interface.

Inline.LongNumber: TYPE = Environment.LongNumber;

Alternatively, the following operations may be used:

Inline.LowHalf: PROCEDURE [LONG UNSPECIFIED] RETURNS [UNSPECIFIED]

Inline.HighHalf: PROCEDURE [LONG UNSPECIFIED] RETURNS [UNSPECIFIED]

LowHalf and **HighHalf** return, respectively, the least and most significant words of its argument.

Note: A **LONG CARDINAL** or **LONG INTEGER** whose value is in **CARDINAL** or **INTEGER**, respectively, may be directly converted to a short value using a mesa range assertion.

The following procedures return the least and most significant bytes of a word, respectively.

Inline.LowByte: PROCEDURE [UNSPECIFIED] RETURNS [UNSPECIFIED]

Inline.HighByte: PROCEDURE [UNSPECIFIED] RETURNS [UNSPECIFIED]

2.2.5.2 Copying blocks of words

The following operations copy blocks of words.

Inline.COPY: PROCEDURE [from: POINTER, nwords: CARDINAL, to: POINTER]

**Inline.LongCOPY: PROCEDURE [from: LONG POINTER, nwords: CARDINAL,
to: LONG POINTER]**

**Inline.LongCOPYReverse: PROCEDURE [from: LONG POINTER, nwords: CARDINAL,
to: LONG POINTER]**

COPY and **LongCOPY** copy **nwords** and are equivalent to the following Mesa code fragment:

FOR i IN [0..nwords] DO (to + i) $\uparrow \leftarrow$ (from + i) \uparrow ENDLOOP;

LongCOPYReverse copies **nwords** and is equivalent to the following Mesa code fragment:

FOR i DECREASING IN [0..nwords] DO (to + i) $\uparrow \leftarrow$ (from + i) \uparrow ENDLOOP;

An upper limit of 65,535 words can be copied in any one call on **Copy**, **LongCopy**, or **LongCopyReverse**.

Caution: Many errors in **COPY**, **LongCOPY**, and **LongCOPYReverse** are the result of an incorrect order of parameters. The keyword constructor call is recommended.

2.2.5.3 Special divide instructions

All of the divide operations described in this section will raise the error **Runtime.ZeroDivisor** if the denominator is zero. All except for **UDDivMod** and **SDDivMod**, will raise **Runtime.DivideCheck** if the quotient is greater than $2^{16}-1$ (see §2.4.3 for more information on these errors).

The quotient and remainder of two cardinals or long cardinals can be obtained with the procedures

**Inline.DIVMOD: PROCEDURE [num, den: CARDINAL]
RETURNS [quotient, remainder: CARDINAL]**

**Inline.UDDivMod: PROCEDURE [num, den: LONG CARDINAL]
RETURNS [quotient, remainder: LONG CARDINAL];**

where **num** is the numerator and **den**, the denominator. The procedure

**Inline.LDIVMOD: PROCEDURE [numlow: WORD, numhigh: CARDINAL, den: CARDINAL]
RETURNS [quotient, remainder: CARDINAL]**

is the same as **DIVMOD** except that the numerator is the double length number **numhigh* 2^{16} + numlow**.

The operation

**Inline.LongDiv: PROCEDURE [num: LONG CARDINAL, den: CARDINAL]
RETURNS [CARDINAL]**

returns the single precision quotient of **num** by **den**.

If both the quotient and remainder of **num** and **den** are desired, the following operation can be used.

Inline.LongDivMod: PROCEDURE [num: LONG CARDINAL, den: CARDINAL]
RETURNS [quotient, remainder: CARDINAL]

The quotient and remainder of two long integers can be obtained with the procedure

Inline.SDDivMod: PROCEDURE [num, den: LONG INTEGER]
RETURNS [quotient, remainder: LONG INTEGER];

2.2.5.4 Special multiply instruction

The double precision product of two cardinals is obtained with

Inline.LongMult: PROCEDURE [CARDINAL, CARDINAL]
RETURNS [product: LONG CARDINAL]

2.2.5.5 Operations on bits

The following operations perform the indicated bitwise logical operations on their operand(s):

Inline.BitOp: TYPE = PROCEDURE [UNSPECIFIED, UNSPECIFIED] RETURNS [UNSPECIFIED];

Inline.BITAND, BITOR, BITXOR: **Inline.BitOp;**

Inline.DBitOp: TYPE = PROCEDURE [LONG UNSPECIFIED, LONG UNSPECIFIED]
RETURNS [LONG UNSPECIFIED];

Inline.DBITAND, DBITOR, DBITXOR: **Inline.DBitOp;**

Inline.BITNOT: PROCEDURE [UNSPECIFIED] RETURNS [UNSPECIFIED]

Inline.DBITNOT: PROCEDURE [LONG UNSPECIFIED] RETURNS [LONG UNSPECIFIED];

A word or double word can be shifted by the operations

Inline.BITSHIFT: PROCEDURE [value: UNSPECIFIED, count: INTEGER]
RETURNS [UNSPECIFIED]

Inline.DBITSHIFT: PROCEDURE [value: LONG UNSPECIFIED, count: INTEGER]
RETURNS [LONG UNSPECIFIED];

Inline.BITROTATE: PROCEDURE [value: UNSPECIFIED, count: INTEGER]
RETURNS [UNSPECIFIED];

These operations return **value** shifted by **ABS[count]** bits. The shift is left if **count > 0**, and right if **count < 0**. In both cases, zeros are supplied to vacated bit positions. In the case of **BITROTATE**, the bits are shifted circularly.

Note: A left shift is a multiply by two ignoring overflow; a right shift is an unsigned divide by two with truncation.

2.3 System timing and control facilities

System: DEFINITIONS . . . ;

NSConstants: DEFINITIONS . . . ;

This section describes some basic system and control facilities provided by Pilot. It introduces and discusses: universal identifiers, by which all network resources and other permanent objects in a network may be named; the means by which communicating processes are identified; the various forms of timekeeping provided by Pilot; the Pilot facilities for turning system power on and off; and how a client gets started.

2.3.1 Universal identifiers

A *universal identifier* may be used for naming all permanent or potentially permanent objects in the network. Every object and every resource may be assigned a separate, unique, universal identifier which is different from any other assigned for any other purpose. Thus, a particular universal identifier can be interpreted unambiguously in any context or on any processor, and it always refers to the same thing.

Universal identifiers are 5word Mesa objects of the following type.

System.UniversalID: TYPE [5];

Pilot issues a new universal identifier, distinct from all others on all other processors at all times, as a result of the operation

System.GetUniversalID: PROCEDURE RETURNS [uid: System.UniversalID];

A **UniversalID** has no internal structure perceptible by client programs, and no properties must be attributed to values of this type except the property of uniqueness. Pilot takes extreme measures to ensure with a very high probability that **UniversalIDs** are not duplicated. The supply of new universal identifiers is limited to an overall processor average of approximately one or a few per second, though the instantaneous rate of creating them can exceed this at times. If Pilot detects any danger of compromising the reliability of the uniqueness property, the process calling **GetUniversalID** is delayed until a new **UniversalID** can be safely issued.

The following are some particular uses of **UniversalIDs**:

System.PhysicalVolumeID: TYPE = RECORD [System.UniversalID];

System.VolumeID: TYPE = RECORD [System.UniversalID];

System.nullID: System.UniversalID = . . . ;

Note: **nullID** is never returned by **GetUniversalID**.

2.3.2 Network addresses

The Internet Transport Protocols are the principal means of communication among processes which reside on different machines (see §6.2, Network streams). A source or destination of such communication is identified by its **NetworkAddress**.

```
System.NetworkAddress: TYPE = MACHINE DEPENDENT RECORD{  
    net: System.NetworkNumber,  
    host: System.HostNumber,  
    socket: System.SocketNumber};
```

```
System.NetworkNumber: TYPE [2];
```

```
System.HostNumber: TYPE [3];
```

```
System.SocketNumber: TYPE [1];
```

```
System.nullNetworkAddress: System.NetworkAddress = ...;
```

```
System.nullNetworkNumber: System.NetworkNumber = ...;
```

```
System.nullHostNumber: System.HostNumber = ...;
```

```
System.broadcastHostNumber: System.HostNumber = ...;
```

```
System.nullSocketNumber: System.SocketNumber = ...;
```

```
System.localHostNumber: READONLY System.HostNumber ;
```

nullNetworkAddress is never used as a source or destination and so may be used when no valid address exists.

nullNetworkNumber is normally not used as a source or destination. However, it can be used on networks that are unable to obtain a network number.

localHostNumber is the **HostNumber** of the local machine.

Within a processor, *sockets* are used to separate and identify communication meant for different purposes or destined for different processes. Sockets are associated with network addresses and are considered to be a reusable resource which is allocated as required.

A **NetworkAddress** is normally retrieved from a Clearinghouse server. The network address of the local system element can be discovered with **NetworkStream.AssignLocalAddress** (*q.v.*). Network addresses are guaranteed to be unique between system restarts, but not across system restarts, i.e., they are reused each time the system is restarted (see chapter 6).

The case of network addresses of processors which are connected to more than one network is still to be determined.

2.3.3 Timekeeping facilities

There are three forms of timekeeping facilities in Pilot: the date and time-of-day, the "stopwatch" or interval timing function, and the "alarm clock" facility.

2.3.3.1 Time of day, and date

The time and date are maintained by Pilot and the system hardware, typically in the form of an accurate, crystal-controlled clock. The following operations are used to access the clock:

```
System.GetGreenwichMeanTime: PROCEDURE
    RETURNS [gmt: System.GreenwichMeanTime];
```

```
System.GreenwichMeanTime: TYPE = RECORD [LONG CARDINAL];
```

```
System.gmtEpoch: System.GreenwichMeanTime = [2114294400];
```

```
System.SecondsSinceEpoch: PROCEDURE [gmt: System.GreenwichMeanTime]
    RETURNS [LONG CARDINAL];
```

The **gmtEpoch** is equivalent to the following:

$$(67 \text{ years} * 365 \text{ days} + 16 \text{ leap days}) * 24 \text{ hours} * 60 \text{ minutes} * 60 \text{ seconds}$$

The **GetGreenwichMeanTime** operation returns the time as a count of seconds since a fixed, arbitrary base time. In particular,

gmt = t corresponds to the time **t-System.gmtEpoch** seconds after midnight, 1 January 1968. That is, the time **System.gmtEpoch + 1** corresponds to 00:00:01, January 1, 1968 (*i.e.*, one second after midnight, ten years prior to the first publication of the *Pilot Functional Specification*).

The "end of time" occurs 2³² seconds after 00:00:01 January 1, 1968. After the "end of time", new clock readings will not be valid. Two **GreenwichMeanTimes** can be compared directly for equality. To find which of two **GreenwichMeanTimes** comes first, apply **SecondsSinceEpoch** to each. This gives the number of seconds that each is after 00:00:00 January 1, 1968. Finally, compare the results to determine which is the later time. That is, compare **SecondsSinceEpoch [t1]** to **SecondsSinceEpoch [t2]** and not **t1** to **t2**.

```
SystemExtras.ClockFailed: SIGNAL;
```

```
PilotSwitchesExtraExtraExtraExtras.ignoreClockFailures:
```

```
PilotSwitches.PilotDomainA = '.;
```

Pilot periodically checks to see if the time-of-day clock is running correctly by **GetGreenwichMeanTime**. If it appears that it is not, the signal **SystemExtras.ClockFailed** will be raised. If the switch **PilotSwitchesExtraExtraExtraExtras.ignoreClockFailures** is down, however, the signal will not be raised.

This signal is resumable, but unless the client sets `ignoreClockFailures`, the signal will probably be raised again.

The operation

```
System.AdjustGreenwichMeanTime: PROCEDURE [
    gmt: System.GreenwichMeanTime, delta: LONG INTEGER]
    RETURNS [System.GreenwichMeanTime];
```

has the result `gmt + delta`. If `t` is a `GreenwichMeanTime` then `[t + delta]` is the `GreenwichMeanTime` that is `delta` seconds after `t`.

Within the range that they overlap, the times defined here and the Alto time standard assign identical bit patterns to a particular time. However, the Pilot standard runs an additional 67 years before overflowing.

Client programs are responsible for converting between Greenwich Mean Time and local time, taking into account Daylight Saving Time, etc., (see the next section).

The time and date is kept accurately (to within a few seconds per month) by the hardware and is adjusted as part of system maintenance. In addition, Pilot ensures that all interconnected system elements on an NS network agree about the current time within a few seconds of each other, and that they agree with an externally supplied timekeeping standard if one is available. Prior to calling the client during booting, Pilot ensures that the processor clock is set correctly. UtilityPilot clients, however, must set the processor clock prior to calling any Pilot operation. This is done by using the operations in the `OthelloOps` interface. If this is not done, the results of Pilot operations are unspecified.

2.3.3.2 Local time parameters

Client programs may obtain the parameters of the local time zone. In normal network configurations, Pilot finds the parameters from a server and remembers them in nonvolatile storage. (Currently it stores them in the root page of the system physical volume.) There is also an operation by which a client can set the parameters (typically on a stand-alone or server machine). The time zone parameters are represented as a record:

```
System.LocalTimeParameters: TYPE = MACHINE DEPENDENT RECORD [
    direction(0:0..0): System.WestEast,
    zone(0:1..4): [0..12],
    zoneMinutes(1:0..6): [0..59],
    beginDST(0:5..15): [0..366],
    endDST(1:7..15): [0..366]];
```

```
System.WestEast: TYPE = MACHINE DEPENDENT {west(0), east(1)};
```

The fields `zone`, `zoneMinutes`, and `direction` together define the time zone as so many hours and minutes west or east of Greenwich. Normally `zoneMinutes` is zero, but there are a few places in the world whose local time is not an integer number of hours from Greenwich. `beginDST` gives the last day of the year on or before which Daylight Savings Time could take effect, where 1 is January 1st and 366 is December 31st (the correspondence between numbers and days is based on a leap year). Similarly, `endDST` gives the last day of the year on or before which Daylight Savings Time could end. Note

that in any given year, Daylight Savings Time actually begins and ends at 2 A.M. on the last Sunday not following the specified date. If Daylight Savings Time is not observed locally, both **beginDST** and **endDST** are zero.

To find the local time zone parameters, a client calls

```
System.GetLocalTimeParameters: PROCEDURE [
  pVID: System.PhysicalVolumeID ← [nullID]]
RETURNS [params: System.LocalTimeParameters];

System.LocalTimeParametersUnknown: ERROR;
```

This procedure returns the local time zone parameters provided that Pilot could determine them either by consulting a network time server during initialization or because they had been previously saved on the system physical volume by a call to **SetLocalTimeParameters** (see below). If the parameters cannot be determined in either of these ways, the error **LocalTimeParametersUnknown** is raised. A normal Pilot client should always default **pVID**. A UtilityPilot client, on the other hand, must specify the ID of the physical volume of the normal system drive, if the local time parameters are to be saved on the disk..

While it is normally unnecessary for a client to do so, the time zone parameters saved in nonvolatile storage on an individual workstation can be set by calling

```
System.SetLocalTimeParameters: PROCEDURE [params: System.LocalTimeParameters,
  pVID: System.PhysicalVolumeID ← [nullID]];
```

The main use for this procedure would be in a server, where a system administrator could set the time zone parameters at system initialization time, in response to an act of Congress, etc. Pilot guarantees the local time parameters are set from the network or from the physical volume on the local disk. In UtilityPilot, however, the client must set the parameters prior to the call on **GetLocalTimeParameters**.

As with **GetLocalTimeParameters**, **pVID** should always be defaulted by a normal client.

2.3.3.3 Interval timing

It is frequently desired to measure elapsed time at the resolution of microseconds during the execution of programs. Such measurements can be used in controlling system behavior, analyzing program or system performance, and stimulating various other activities. In many cases, the processor underlying Pilot will not provide a timer with a resolution of one microsecond. As a result, Pilot would have to convert between processor dependent units and microseconds to provide a timing facility that measured in microseconds. In many cases, the overhead inherent in this conversion would be large enough to inhibit the timing of functions. For this reason, **Pulses** are provided:

```
System.Pulses: TYPE = RECORD [pulses: LONG CARDINAL];
```

A **Pulse** is a processor dependent unit of time. The actual resolution and accuracy of **Pulses** is determined by the accuracy and resolution of the internal clocks of the processor. Typically, resolution of **Pulses** will be in the range 1 - 1000 microseconds and it will be

accurate to within 10% or better. The current value of the processor interval timer may be read by

System.GetClockPulses: PROCEDURE RETURNS [System.Pulses];

A client may convert between pulses and microseconds with the operations:

**System.PulsesToMicroseconds: PROCEDURE [p:System.Pulses]
RETURNS [m: System.Microseconds];**

**System.MicrosecondsToPulses: PROCEDURE [m:System.Microseconds]
RETURNS [p:System.Pulses];**

System.Microseconds: TYPE = LONG CARDINAL;

System.Overflow: ERROR;

To perform accurate timings, the user should measure events in terms of **Pulses** and only convert to and from microseconds when it is absolutely necessary. In particular, **Pulses** should be the time units used in the inner loops of programs or in any place where time is critical.

Conversion in one direction or the other may cause an overflow. When this happens, Pilot will raise the error **Overflow**.

Caution: The error **Overflow** is not implemented in Pilot 11.0.

The processor interval timer wraps around after a processor dependent period of time, typically greater than one hour. Thus, **Pulses** cannot be used to measure events with a duration in excess of the wrap around period.

2.3.3.4 Alarm clocks

An alarm clock facility is provided by the Mesa process mechanism. A timeout value may be assigned to any condition variable by means of the operation **Process.SetTimeout** (see §2.4.1.2). A process may then "go to sleep" for that period by executing a **WAIT** operation on that condition variable. When the timeout expires (or when a **NOTIFY** operation is executed on that condition variable, whichever comes first), the process awakens and continues execution. One convenient way for a process to wait when there is no requirement for a **NOTIFY** wakeup is to call **Process.Pause** (§2.4.1.6).

The resolution of the process timer is on the order of 15-50 milliseconds. It has no accuracy whatsoever. Thus, a client process must check either the time of day, an interval timer or the processor timer if it needs to know the time accurately.

2.3.4 Control of system power

The following operations allow the processor to be turned on and off under program control.

System.PowerOff: PROCEDURE;

This operation causes the machine to be turned off. It does not return. Pilot will cause all input/output activity to be suspended, purge all of its internal caches, force out all mapped spaces to their file windows, stop all processes from further execution, and cause the display to be turned off. The only way to recover from this operation is to turn the system power on and press the restart button. If there is no power relay, the system element remains turned on but executing no programs; a unique code is displayed in the maintenance panel in this situation.

The operation

**System.SetAutomaticPowerOn: PROCEDURE [
time: System.GreenwichMeanTime, externalEvent: BOOLEAN];**

sets the internal clock of the processor to automatically turn on the system power at or after time. If externalEvent is FALSE, power is turned on at the specified time. If externalEvent is TRUE, power is turned on in response to the first external event occurring after the time specified by time. An external event is an electrical signal made available to the processor (e.g., the ringing signal of a data telephone).

If power is already on when this operation would turn it on, there is no effect. The automatic power on facility may be reset by calling

System.ResetAutomaticPowerOn: PROCEDURE;**2.3.5 Pilot's state after booting**

The device that the system was booted (loaded) from may be ascertained by referencing

System.SystemBootDevice: READONLY System.BootDevice;**System.BootDevice: TYPE = RECORD [device: Device.Type, index: CARDINAL];**

Client programs can determine if they are running upon UtilityPilot with:

System.isUtilityPilot: READONLY BOOLEAN;

Boot switches are used to transmit operational information from the booting agent (e.g., Othello) to the running boot file (see client documentation for definitions of applicable boot switches). The boot switches are made available as

System.Switches: TYPE = PACKED ARRAY CHARACTER OF System.UpDown;**System.UpDown: TYPE = MACHINE DEPENDENT {up(0), down(1)};****System.switches: READONLY System.Switches;****System.defaultSwitches: System.Switches = ALL[up];**

If a switch is down, then it is active; if a switch is up, then it is inactive. The value of switches is determined as follows. First, if the booting agent provides switches other than defaultSwitches, that value is used. Otherwise, if the boot file was constructed (by

`MakeBoot`) to contain other than `defaultSwitches`, that value is used. Otherwise, `defaultSwitches` is used.

Switch assignments are made by the Manager of Operating Systems. Ranges of switches are allocated for Pilot, for the Mesa Development Environment, and for product systems. The following list enumerates those switches currently used by to Pilot and describes their significance.

<u>Value</u>	<u>Meaning</u>
&	Hang with a maintenance panel code in lieu of going to the debugger.
0	Go to debugger as early as possible in Pilot initialization.
1	Go to debugger as soon as all code is map-logged.
2	Go to debugger just before calling <code>PilotClient.Run</code> .
3	Simulate 192K memory size for a Dandelion with no display .
4	Initialize scratch memory pages to zero.
5	Go to the Ethernet for a debugger.
6	Turn owner checking on for the system zones.
7	Disable map logging (see below).
8	Create a Pilot interrupt key watcher.
9	Simulate 256K memory size for a Dandelion with display.
:	Go to the debugger as early in initialization of the File manager as possible.
;	Go to the debugger as early in the initialization of the VM manager as possible.
<	Pretend that there is no Ethernet 1 attached to the system element.
=	Do not initialize the Communication package at system start-up.
>	Pretend that there is no Ethernet attached to the system element.
{	Set the VM backing file size to 750 pages (see below).
	Set the VM backing file size to 1400 pages (see below).
}	Set the VM backing file size to 2000 pages (see below).
↑	Turn checking on the for system zones.
?	Make loadstate resident (for debugging on UtilityPilot-based clients).
[Create a tiny heap, with tiny increment values.
%	Create a medium-size heap, with medium-size increment values (default).
]	Create a large heap, with large increment values).
/360	Display error code, global frame, and pc on boot loader errors.
/361	Transmit Ethernet packets using IEEE 802.2 Logical Link Layer protocol.
/362	Accept packets from the Ethernet in either IEEE 802.2 Logical Link Layer or Ethernet version 1.0 format.
/363	Transmit packets to hosts in the format that the receiver desires.
/366	Hold back 48 pages of reserved display memory.
/367	Hold back 64 pages of reserved display memory.
/370	Bypass the debugger substitute by going to the real debugger.
/371	Tile code with one page swap units.
/372	Give display memory to Pilot for client use.
/373	Give display memory to Pilot for client use if no bitmap display.
/374	Allows special clients to set parameters of system zones
/375	Disable map logging (see below).
/376	Delete boot loader so that the memory that it uses can be recycled.

Full map logging is the default case when Pilot is booted if there is a debugger present. Full map logging includes occasionally going to the debugger to clean up the log. A debugger is considered present if there is a debugger installed on a volume of type one

higher than that of the boot file, or if debugger pointers have been set in the boot file, or if a remote debugger is specified (boot switch "5").

If there is no debugger present, map logging proceeds until the log file fills up and then logging is disabled. This situation may be forced by setting boot switch /375. Boot switch "7" will cause Pilot to stop map logging when **PilotClient.Run** is called (at key stop 2). This key switch overrides key switch /375.

The VM backing file is the file which is used to provide the backing file for Pilot data spaces (*q.v.*). Under normal circumstances, its size is a function of the size of the volume booted from. For some logical volumes the default size may be too small. In that event, the switches "{", "|", and "}" may be used to specify the size of the backing file.

2.4 Mesa run-time support

This section describes low-level facilities used to support the execution of Mesa programs. It describes operations to support the Mesa process mechanism; facilities relating to Mesa program modules; traps, signals, and errors which may be generated by a Mesa program during execution; and finally, some miscellaneous interfaces.

2.4.1 Processes and monitors

Process: DEFINITIONS . . . ;

Most aspects of processes and monitors are made available via constructs in the Mesa language and are described in the *Mesa Language Manual*. Some operations whose frequency of use does not justify such treatment are cast as procedures.

When a process is **FORKed**, it is called a *live* process. When it has been **JOINED** or when it has been detached and its root procedure has returned, it is called a *dead* process. Programs must take care not to use or retain copies of the **PROCESS** of a dead process. Since Pilot may reuse **PROCESSES**, any operation performed on the **PROCESS** of a dead process may mistakenly operate on a different process than the one intended, with unpredictable results.

Most of the operations which take a **PROCESS** as an argument (**JOIN**, **ProcessAbort**, and **ProcessDetach**) may generate the following signal:

Process.InvalidProcess: ERROR [process: PROCESS];

This signal indicates that the argument is not a live process.

The argument of **InvalidProcess** is actually of type **UNSPECIFIED**. This is necessary since there is no generic type which includes all **PROCESS** types, independent of their result types. The same is true of all arguments and results discussed in this section that would otherwise be of type **PROCESS**.

A **PROCESS** can be checked for validity by the operation

Process.ValidateProcess: PROCEDURE [UNSPECIFIED]

If the argument does not represent a live process, **Process.InvalidProcess** will be raised. Otherwise, this operation just returns.

Caution: Since Pilot may reuse PROCESSES, **ValidateProcess** applied to the **PROCESS** of a dead process may not raise **InvalidProcess**. Such a dangling reference will appear legitimate to **ValidateProcess**, but is almost certain to cause trouble for any client program that makes use of it.

2.4.1.1 Initializing monitors and condition variables

Every monitor lock and every condition variable must be initialized before it can be used. There are three cases:

Any monitor lock or condition variable residing in a global frame will be initialized automatically when the program is **STARTED**. Any monitor lock or condition variable residing in a local frame will be initialized automatically when the procedure is entered.

Any monitor lock or condition variable allocated dynamically by the **NEW** operator (from an uncounted zone or MDS zone) will be initialized automatically upon allocation.

Any monitor lock or condition variable allocated dynamically by other than the **NEW** operator must be initialized by the programmer using the facilities described below.

Caution: Using uninitialized monitor locks or condition variables, or reinitializing monitor locks or condition variables once they are in use, will lead to totally unpredictable behavior.

The following operations are provided for initializing monitor locks and condition variables which are allocated dynamically by other than the **NEW** operator:

Process.InitializeMonitor: PROCEDURE [monitor: LONG POINTER TO MONITORLOCK];

InitializeMonitor sets the monitor unlocked and the queue of waiting processes to empty. It may be called before or after the monitor data is initialized, but *must* be called before any entry procedure is invoked. Once use of the monitor has begun, the monitor *must never be initialized again*.

Process.InitializeCondition: PROCEDURE[condition: LONG POINTER TO CONDITION,
ticks: Process.Ticks];

Process.Ticks: TYPE = CARDINAL;

InitializeCondition sets the queue of waiting processes to empty and the timeout interval of the condition variable to the specified value, in units of "ticks" of the process timer clock. It may be called before or after the other monitor data is initialized, but *must* be called before any **WAIT** or **NOTIFY** operations are performed on the condition variable. Once use of the condition variable has begun, the condition variable *must never be initialized again*.

Clients may convert process timer ticks to or from milliseconds using the following operations:

Process.Milliseconds: TYPE = CARDINAL;

Process.MsecToTicks: PROCEDURE [Process.Milliseconds] RETURNS [Process.Ticks];

**Process.TicksToMsec: PROCEDURE [ticks: Process.Ticks]
RETURNS [Process.Milliseconds];**

For setting long timeout intervals, the following operation is provided:

Process.Seconds: TYPE = CARDINAL;

**Process.SecondsToTicks: PROCEDURE [Process.Seconds]
RETURNS [Process.Ticks];**

Caution: Due to the limited range of the process timer, the maximum timeout that maybe set is about 980 seconds (16 minutes).

2.4.1.2 Timeouts

Condition variables that are initialized automatically do not time out. The time out of any condition variable may be changed by the operation:

**Process.SetTimeout: PROCEDURE
[condition: LONG POINTER TO CONDITION, ticks: Process.Ticks];**

The given timeout interval will be effective for all subsequent **WAIT** operations applied to the condition variable. This operation will not affect the timeout interval of any processes currently waiting on the condition variable.

Process.DisableTimeout: PROCEDURE [LONG POINTER TO CONDITION];

DisableTimeout sets the timeout interval for the condition variable to infinity. That is, a process waiting on the condition variable will never time out. This will be effective for all subsequent **WAIT** operations applied to that condition variable. This operation will not affect the timeout interval of any processes currently waiting on the condition variable.

SetTimeout and **DisableTimeout** are the only operations that may be used to adjust the timeout interval of a condition variable once it has been used. In particular, **InitializeCondition** must not be used for this purpose.

Caution: Since the Mesa processor reserves some distinguished values of **Ticks** for special purposes, the timeout interval of a condition variable should *not* be set via the Mesa construct:

condition.timeout ← ticks. --WRONG

2.4.1.3 Forking processes

There is a limit on the number of co-existing processes allowed by Pilot. Attempts to fork too many processes will result in the error

Process.TooManyProcesses: ERROR;

This may be caught by a catch phrase on the **FORK**, or by a catch phrase in some enclosing context.

The maximum number of coexisting processes is specified to **MakeBoot** when building a boot file. See the *Mesa User's Guide* for details.

A process which is **FORKed** but will never be **JOINED** should be detached using the operation

Process.Detach: PROCEDURE [PROCESS];

This operation conditions the process such that when it returns from its root procedure, it will be deleted immediately.

Caution: Note that a variable of type **PROCESS** does not return results. If the root procedure of a process does return results, it will be necessary to loophole the parameter to **Detach**. In those cases, care should be exercised because if the results returned take more than 12 words of storage, the storage that contains the results (a local frame) will be discarded and the space will never be recovered. If there are 12 or less words of results, the results will be discarded and the storage recovered.

A process may determine its own identity by invoking

Process.GetCurrent: PROCEDURE RETURNS [PROCESS];

2.4.1.4 Priorities of processes

When a process is created with **FORK**, it inherits the priority of the forking process. A process may change its own priority with the **SetPriority** operation.

Process.SetPriority: PROCEDURE [Process.Priority];

Process.priorityBackground: READONLY Process.Priority;

Process.priorityNormal: READONLY Process.Priority;

Process.priorityForeground: READONLY Process.Priority;

Process.Priority: TYPE = [0..7];

Larger values of **Priority** correspond to higher priorities. Implementation restrictions make it necessary to limit ordinary client processes to three priority levels, defined via exported variables, which are listed above in order of *increasing* priority. **SetPriority** should only be given one of these three constants (or a value previously obtained from **GetPriority**, which will be equal to one of these constants).

If it is desired to fork a process which runs immediately at a higher priority than the parent process, the parent can set its *own* priority to the higher level, fork the new process, and then restore its own priority.

A process may determine its own priority by calling

Process.GetPriority: PROCEDURE RETURNS [Process.Priority];

2.4.1.5 Aborting a process

A process can be aborted by calling the operation

Process.Abort: PROCEDURE [process: UNSPECIFIED];

The effect of this operation is to generate the error **ABORTED** the next time the process **WAITS** on *any* condition variable which has aborts enabled. If the process is already waiting, the error is generated immediately.

ABORTED may be caught by a catch phrase on the **WAIT**, or by a catch phrase in an enclosing context. The catch phrase is executed with the corresponding monitor locked.

The intended use of **Abort** is to provide a means whereby one process may request of another that the latter should stop what it is doing. An **ABORTED** signal may occur on *any* condition variable which has aborts enabled, and thus *every* monitor should either be protected by some catch phrase for it, or contain no condition variables which have aborts enabled.

A pending abort may be canceled by calling the operation

Process.CancelAbort: PROCEDURE [process: UNSPECIFIED];

A process may discover if there is an abort pending for it by the operation

Process.AbortPending: PROCEDURE [] RETURNS [abortPending: BOOLEAN];

When a condition variable is initialized, it has aborts disabled. A condition variable may be set to allow aborts by the operation:

Process.EnableAborts: PROCEDURE [LONG POINTER TO CONDITION];

If a process with an abort pending is currently waiting on the condition variable, **EnableAborts** will have no immediate effect on it. However, if the process times out or is **NOTIFYed**, it will be aborted at that time.

It is sometimes desirable to avoid aborts while waiting on a given condition variable. This may be effected by using

Process.DisableAborts: PROCEDURE [LONG POINTER TO CONDITION];

Condition variables are initialized to have aborts disabled. If a process with an abort pending waits or is waiting on a condition variable, the abort will be delayed until the process **WAITS** on some other condition variable which has aborts enabled.

A process can be suspended for a specified number of ticks with the operation

Process.Pause: PROCEDURE [ticks: Process.Ticks];

Pause waits with aborts enabled, and so may raise the error **ABORTED**. Note that monitor locks of the caller are not released during the pause.

The Mesa process mechanism does *not* attempt to allocate processor time fairly among processes of equal priority. A process itself will yield the processor to other processes of equal priority whenever it faults, **Pauses** or **WAITS**. If a process does these things only rarely, it may be desirable for it to occasionally yield control of the processor by calling

Process.Yield: PROCEDURE;

This places the calling process at the rear of the queue of ready-to-run processes of the same priority. Thus, all other ready processes of the same priority will run before the calling process next runs. Note however, that these other processes may make arbitrarily little progress due to page faults, etc.

The logical correctness of client programs must *not* depend on the presence or absence of calls to **Yield**. Priorities and yielding are *not* intended as a process-synchronization mechanism. They are only provided to assist in meeting performance requirements.

2.4.2 Programs and configurations

Runtime: DEFINITIONS . . . ;

Programs may be validated by

Runtime.ValidateGlobalFrame: PROCEDURE [frame: Runtime.GenericProgram];

Runtime.GenericProgram: TYPE = LONG UNSPECIFIED;

Runtime.InvalidGlobalFrame: ERROR [frame: Runtime.GenericProgram];

If **frame** is not valid, **InvalidGlobalFrame** is raised. **frame** may be either a **PROGRAM** or a **LONG POINTER TO FRAME[<program>]**. Normal usage requires a **LOOPHOLE**.

Pointers to procedure activation records (local frames) may be validated by

Runtime.ValidateFrame: PROCEDURE [frame: UNSPECIFIED];

Runtime.InvalidFrame: ERROR [frame: UNSPECIFIED];

If **frame** is definitely not valid, **InvalidFrame** is raised. **frame** should be a **POINTER TO FRAME[<procedure>]**. The checking done by **ValidateFrame** only verifies that frame looks like a valid local frame; it is not possible for it to verify that it actually is a valid local frame.

Runtime.nullProgram: PROGRAM = NIL;

For backwards compatibility, a null **PROGRAM** constant is provided. New client code should just use **NIL**.

The **PROGRAM** containing a **PROCEDURE** can be obtained using

```
Runtime.GlobalFrame: PROCEDURE [link: Runtime.ControlLink] RETURNS [PROGRAM];
```

```
Runtime.ControlLink: TYPE = LONG UNSPECIFIED;
```

ControlLink may be any **PROCEDURE**. Normal usage requires a **LOophole**. If **link** is an unbound procedure, **Runtime.UnboundProcedure** is raised. **Runtime.InvalidGlobalFrame** may also be raised.

A program which was created by **NEW <program>** may be deleted using

```
Runtime.UnNew: PROCEDURE [frame: PROGRAM];
```

UnNew deletes the program and reclaims its storage. All items which were exported by the program (procedures, variables, signals, and the program itself) become dangling references and should not be retained or used by any programs which imported them. If **frame** is not valid, **Runtime.InvalidGlobalFrame** is raised. If the program was not created by **NEW <program>**, the debugger is called.

Caution: When a program is **UnNewed**, there must be no processes executing procedures in the program or expecting to return to procedures in it. Failure to observe this rule will lead to unpredictable behavior.

Since **UnNew** may not be used while any processes are using a program, it is not possible for a process to **UnNew** the program in which it is currently executing. Since this is occasionally desirable, a special operation is provided:

```
Runtime.SelfDestruct: PROCEDURE;
```

SelfDestruct deletes the program that invokes it and then returns, with no results, to the first enclosing context which is not in the deleted program. All items which were exported by the program (procedures, variables, signals, and the program itself) become dangling references and should not be retained or used by any programs which imported them.

Caution: Since **SelfDestruct** effects a **RETURN** without results to the first enclosing context which is not in the deleted program, the procedure which was called from that context *must* be declared as having no results; otherwise a stack error will occur.

Caution: When a program is **SelfDestructed**, there must be no *other* processes executing procedures in the program or expecting to return to procedures in it. Failure to observe this rule will lead to unpredictable behavior.

The following operations are used to load configurations and programs. They are implemented by the object file **Loader.bcd**.

```
Runtime.RunConfig: PROCEDURE [  
  file: File.File, offset: File.PageCount, codeLinks: BOOLEAN ← FALSE];
```

```
Runtime.LoadConfig: PROCEDURE [
    file: File.File, offset: File.PageCount, codeLinks: BOOLEAN ← FALSE]
    RETURNS [PROGRAM];

Runtime.NewConfig: PROCEDURE [
    file: File.File, offset: File.PageCount, codeLinks: BOOLEAN ← FALSE];
    RETURNS [PROGRAM];

Runtime.ConfigError: ERROR [type: Runtime.ConfigErrorType];

Runtime.ConfigErrorType: TYPE = {
    badCode, exportedTypeClash, invalidConfig, missingCode, unknown};

Runtime.VersionMismatch: SIGNAL [module: LONG STRING];
```

These operations load a configuration or program from the object file contained in **file** starting at page **offset** of the file. **offset** enables one to skip leader pages, pack many object files into one, etc. Each program in the object file will be loaded with code links if (1) **codeLinks** = **TRUE**, and (2) the object file is a configuration, and (3) the program or a configuration containing the program specified **LINKS: CODE**, and (4) a configuration containing that configuration or program was packaged, or bound specifying code copying. If a program is loaded with code links, its links are written into the object file. The three operations differ as follows. **LoadConfig** loads the object file and returns a **PROGRAM**. The **PROGRAM** is used to start the object file. If the object file is a configuration, **PROGRAM** is one of the configuration's control programs (= **NIL** if the configuration has no control programs); if the object file is not a configuration, **PROGRAM** is the program itself. A subsequent **START <program>** will initialize the loaded programs (note that **START NIL** is a no-operation). **RunConfig** both loads and starts the object file. **NewConfig** loads the object file and throws away the **PROGRAM**, thus preventing it from being explicitly started. Using **NewConfig** is only appropriate if the configuration does not require initialization; its use is not recommended.

If an object file being loaded imports an interface item and there are several instances of that interface item being exported by already-loaded objects files, the import is bound to the most-recently loaded instance of the interface item. If an object file being loaded imports an interface item which it itself exports, the import is bound to the one it exports.

If the object file being loaded imports or exports a version of a program which differs from a version exported or imported by already-loaded files, **Runtime.VersionMismatch** is raised, passing the name of the offending program. Resuming this signal allows loading to proceed; the imported items with mismatched versions remain unbound. The signal is raised once for each mismatch encountered.

Note: If **VersionMismatch** is resumed, the system will be exporting two different versions of various programs. Object files loaded subsequently which import these programs may get **VersionMismatch** against the "bad" version; however, if the signal is resumed and the correct version is found, the desired binding will be done.

If the code for any of the programs is not contained in the object file (typically because a configuration was not bound with code copying), **Runtime.ConfigError[missingCode]** is raised. If the object file exports a **TYPE** that differs from that exported by an already loaded program, **Runtime.ConfigError[exportedTypeClash]** is raised. If any program in the object file is loaded with code links but the volume containing **file** is read-only, **Volume.ReadOnly**

is raised. If the object file contains a definitions module, is not compatible with the current version of Mesa, or is not an object file at all, **Runtime.ConfigError[invalidConfig]** is raised. If the object file is not completely contained in the file, **Space.Error[noWindow]** is raised. Any of the errors raised by **Space.Map** may also be raised. **ConfigErrorTypes** of **badCode** and **unknown** are not used at present.

Caution: If a program in the boot file imports an item which is satisfied by a configuration which is loaded at run-time, the importing program must have frame links. If this rule is not followed, the link to the imported item will be written into the boot file, and will be a dangling reference when the boot file is invoked at later times.

A object file which was loaded at run-time may be unloaded by

Runtime.UnNewConfig: PROCEDURE [link: Runtime.ControlLink];

UnNewConfig unloads the dynamically-loaded object file associated with **link**. **link** may be any **PROCEDURE** or **PROGRAM** in the object file. **UnNewConfig** frees the storage of all **PROGRAMs** of the object file, and unmaps and deallocates the virtual memory containing its code. All items that were bound to the object file are reset to unbound.

Caution: When an object file is **UnNewConfiged**, there must be no processes executing procedures in programs of the object file or expecting to return to procedures in them. Failure to observe this rule will lead to unpredictable behavior.

The time at which the currently running boot file was built by **MakeBoot** is returned by

Runtime.GetBuildTime: PROCEDURE RETURNS [System.GreenwichMeanTime];

The time at which a configuration was bound is returned by

Runtime.GetBcdTime: PROCEDURE RETURNS [System.GreenwichMeanTime];

This operation returns the bind or compile time of the outermost configuration containing the caller of **GetBcdTime**. If there are no containing configurations, **GetBcdTime** returns the compile time of the caller.

The next two operations are useful for debugging and determining what has been loaded.

Runtime.GetCaller: PROCEDURE RETURNS [PROGRAM];

GetCaller returns the **PROGRAM** that called the client's **PROGRAM**. More precisely, it returns the **PROGRAM** of the innermost enclosing context which is outside the **PROGRAM** that contains the procedure called **GetCaller**.

Runtime.IsBound: PROCEDURE [link: Runtime.ControlLink] RETURNS [BOOLEAN];

IsBound returns **TRUE** if the imported procedure **link** is bound (i.e., if **link** is being exported). Normal usage requires a **LOOPHOLE**. **link** may also be an imported variable or an imported **PROGRAM**.

Caution: Unexpected results can be experienced using code links, run-time loading and **IsBound**. In particular, if a program in the boot file is loaded with code links and imports an item which is satisfied by a configuration which is loaded at run-time, the program will

have links which appear to be bound but are actually left over from a previous boot session. Boot file importers of unbound items should be bound with frame links.

A pointer to the data portion of a program compiled with the Table Compiler is returned by

Runtime.GetTableBase: PROCEDURE [frame: PROGRAM] RETURNS [LONG POINTER];

GetTableBase may raise **Runtime.InvalidGlobalFrame**.

2.4.3 Traps and signals

Programming errors and other errors encountered by Mesa programs result in signals or errors. The first five errors described below are related to specific language features and are described in more detail in the *Mesa Language Manual*.

Runtime.StartFault: ERROR [dest: PROGRAM];

StartFault is raised if **dest** was **STARTed** but it had been started previously (perhaps by a start trap), or if **dest** was **RESTARTed** but it had not **STOPped**.

Note: If a program does **START <program>** but **program** is not valid, **Runtime.InvalidGlobalFrame** is raised. This will happen if **program** is an unbound import.

Runtime.ControlFault: ERROR [source: Runtime.ControlLink];

ControlFault is raised if a program attempts to transfer to a null control link while executing in the local frame denoted by **source**. This error passes the control link that was used. In the current version of Mesa, **ControlFault** *may* be raised on an attempt to call an unbound **PROCEDURE** (instead of **UnboundProcedure**).

Runtime.UnboundProcedure: ERROR [dest: Runtime.ControlLink];

UnboundProcedure is raised if a program attempts to call an unbound **PROCEDURE**. This error passes the **PROCEDURE** that was called.

Caution: In the current version of Mesa, **ControlFault** *may* be raised instead of **UnboundProcedure**.

Runtime.LinkageFault: ERROR;

A transfer has been attempted through a port that has not been connected to some other port or procedure (the link field of the port was **NIL**).

Runtime.PortFault: ERROR;

A transfer has been attempted to a port which is not pending (the frame field of the destination port is **NIL**). This error is used to handle the transients normally occurring while initializing coroutines.

BoundsFault: SIGNAL;

A value being assigned to a subrange variable or being used in an indexing operation was out of range. This signal may also be raised if an attempt is made to assign a signed value to an unsigned variable and vice versa. This signal is only raised by programs which have been compiled specifying bounds checking. RESUMEing this signal will allow the program to use the illegal value, with unpredictable results.

NarrowFault: ERROR;

An attempt was made to use the **NARROW** operator on a value **x** to make it of **TYPE T**, but the type of the value of **x** was some other. For example, an attempt was made to narrow a (pointer to a) variant record to a (pointer to a) specific variant, but the value of **x** was some other variant.

PointerFault: SIGNAL;

An attempt has been made to dereference a **NIL** pointer. This signal is only raised by programs which have been compiled specifying nil checking. RESUMEing this signal will use the **NIL** value, almost invariably causing an immediate address fault.

Note: Pilot leaves virtual address **NIL** ↑ and **LONG NIL** ↑ unmapped. Attempts to dereference a **NIL** pointer will usually cause an address fault.

Runtime.ZeroDivisor: SIGNAL;

An attempt was made to divide by zero. If this signal is **RESUMED**, the result of the divide operation is undefined.

Runtime.DivideCheck: SIGNAL;

An attempt was made to perform a division involving **LONG** operand(s) whose result could not be expressed in a single word. If this signal is **RESUMED**, the result of the divide operation is undefined.

2.4.4 Calling the debugger or backstop

A program can explicitly invoke the debugger or backstop by calling

Runtime.CallDebugger: PROCEDURE [LONG STRING];

Client program execution is suspended. The debugger prints the string provided and awaits user commands. A Proceed command resumes client program execution after the call to **CallDebugger**. (If continuing execution at this point is not reasonable, the call to **CallDebugger** should be placed inside a non-terminating loop.)

A program may also invoke the debugger or backstop by calling

Runtime.Interrupt: PROCEDURE;

The debugger prints "**** Interrupt ****" and awaits user commands. **Interrupt** is typically called by a user input handling process in response to some user action such as typing a special keyboard key.

2.5 Client startup

```
PilotClient: DEFINITIONS . . . ;
```

Pilot imports precisely one client interface, called **PilotClient**. The **PilotClient** interface is defined as follows:

```
PilotClient: DEFINITIONS =
BEGIN
  Run: PROCEDURE [];
END.
```

The client configuration must export a **PROCEDURE** called **PilotClient.Run**. Pilot initializes itself and without explicitly **STARTing** any client programs calls **Run**, the first client procedure, as follows:

```
...
Process.SetPriority[Process.priorityNormal];
Process.Detach[ FORK PilotClient.Run[] ];
...
```

This will cause a start trap within the program containing **Run**, and will thus start the control module(s) of the containing configuration, if any. **Run** is responsible for loading and starting all client programs, creating spaces, forking processes, etc. It may freely use the Mesa **NEW** statement, refer to any known file, and use any facility of Pilot. It may or may not have a user interface, depending upon the application it implements.

2.6 Coordinating subsystems' acquisition of resources

```
Supervisor: DEFINITIONS . . . ;
```

```
SupervisorEventIndex: DEFINITIONS . . . ;
```

This interface provides a facility for notifying interested clients of events which typically have a fairly widespread impact. The Supervisor can be used for managing the orderly acquisition and release of shared resources such as a file, a removable volume, or, in the case of restarting the machine from a restart file, the entire processor. The Supervisor facility has some similarities to the Ethernet, in that it provides a way to broadcast information (within a single processor) to an expandable collection of interested client software.

The Supervisor accommodates a model of the entire client system as a collection of subsystems which depend on some basic resource. To handle this model, the Supervisor maintains a database which describes dependency relationships, and provides a way to invoke the subsystems in a clients-first or implementors-first order.

Consider the event where a user indicates that he wants to withdraw a removable volume from a system element. The subsystems which are using the volume must release it in an orderly manner. Since the volume typically will be used by lower-level subsystems to build higher-level abstractions for its clients, the higher-level abstractions must also be released, and indeed must be released before the lower-level subsystem may release the volume. Thus, the releasing of a volume should normally proceed in a clients-first order.

Similarly, when a volume is added to a system, the subsystems which would like to use it should acquire it in an orderly manner, typically implementing subsystems first.

Events for which the Supervisor may be useful include:

- Making a restart file.
- Restarting the system element from a restart file.
- Removing or adding a physical or logical volume.
- Turning power off (possibly with Automatic Power On enabled).
- The appearance/disappearance of some service or resource on this or another system element.

The implementation module is **SupervisorImpl.bcd**.

2.6.1 Use of the Supervisor

Each subsystem should obtain a subsystem handle from the Supervisor and export it to its clients. The handles are used by clients to declare, to the Supervisor, which subsystems they depend on. Each subsystem also registers an agent procedure. When an interesting event happens, the Supervisor is invoked to notify, in proper order, the agent procedures of all subsystems, informing them of the event. Upon return from this enumeration, all subsystems will have been notified of the event.

Since several lowest-level subsystems may utilize the same basic resource, the event of releasing a resource might be organized as follows: the enumeration would have each subsystem release its use of the resource, and then the caller of the enumeration would actually release the basic resource.

On the other hand, acquisition of a new resource is slightly different. The enumeration would declare the availability of a new resource. The lowest level subsystems might implement some higher-level resource on it, and then that subsystem's clients could interrogate it for the new resources when their agent procedures were called.

For example, in the event of removing a physical volume from the system element, the agent procedure for a subsystem might perform the following actions:

1. Put the subsystem's processes to sleep, or into some quiescent state;
2. Browse through the subsystem's database and locate any objects which were built upon files residing on the physical volume to be removed; this step may well involve calls to some lower-level subsystems to determine the physical location of their objects;
3. Delete or otherwise make inactive any objects based on these files and update the database accordingly;
4. Reawaken its processes;

5. Return.

The enumeration of subsystems is typically invoked from a very high level, not from within a monitor implementing a resource which is acquired or released.

2.6.2 Supervisor facilities

An **Event** is a value that names a particular event in which some subsystems may be interested.

Supervisor.Event: TYPE = RECORD [eventIndex: Supervisor.EventIndex];

Supervisor.EventIndex: TYPE = CARDINAL;

Supervisor.nullEvent: Supervisor.Event = Supervisor.Event[LAST[Supervisor.EventIndex]];

The domain of **Event** is shared by all of the Supervisor's clients, who therefore must agree on the meaning of the values. If some software that uses events runs in several disparate systems (e.g., Star and Tajo), then those systems must agree on the values of the events which are common to both systems. In this case, there is a common definitions module, **SupervisorEventIndex**, which defines subdomains for those events common to each system, and subdomains for those events unique to each system. Also disallowed is the defining of one element of **Event** to correspond to more than one event. That is, there cannot be any catch-all **Events**.

The basic structure of the **SupervisorEventIndex** interface is a set of subrange definitions. The following ranges are defined.

SupervisorEventIndex.EventIndex: TYPE = Supervisor.EventIndex;

SupervisorEventIndex.MesaEventIndex: TYPE = CARDINAL [0..1024];

SupervisorEventIndex.CommonSoftwareEventIndex: TYPE = CARDINAL [1024..1280];

MesaEventIndex's are used by Mesa source and object files. **CommonSoftwareEventIndex**'s are used by product common software.

Note: Each client of **SupervisorEventIndex** interface should maintain an interface which defines the **Events** in its subrange.

Each software component or subsystem which is interested in events should register an **AgentProcedure**, which will be called when events occur:

Supervisor.AgentProcedure: TYPE = PROCEDURE [event: Supervisor.Event,
eventData: LONG POINTER TO UNSPECIFIED, instanceData: LONG POINTER TO UNSPECIFIED];

Supervisor.nullAgentProcedure: Supervisor.AgentProcedure = NIL;

When an agent procedure is called, it should first examine **event**, and ignore ones which it does not recognize or care about. The agent procedure may use facilities upon which it depends (see **DependsOn** below). **eventData** is supplied by the software that caused the

notification of the event, and its interpretation depends on **event.eventData** might be declared as

```
eventData: LONG POINTER TO RECORD [SELECT COMPUTED event.eventIndex FROM . . . ENDCASE;
```

instanceData is supplied when the agent procedure is declared to the Supervisor, and may be used to convey to the agent procedure any data necessary for a particular instance of its parent program. An **AgentProcedure** of **NIL** may be used for subsystems which do not wish to have an associated agent procedure. For backwards compatibility, a null **AgentProcedure** constant is provided. New client code should just use **NIL**.

The client's **AgentProcedure** must not call back into the Supervisor, either directly or indirectly, as this will cause the containing process to hang on a monitor lock.

To participate in the event mechanism, each implementing subsystem must register itself with the Supervisor. When it does, the Supervisor returns a **SubsystemHandle**, which is used to identify the subsystem to the Supervisor, and to the subsystem's clients.

```
Supervisor.SubsystemHandle: TYPE [1];
```

```
Supervisor.nullSubsystem: READONLY Supervisor.SubsystemHandle;
```

```
Supervisor.CreateSubsystem: PROCEDURE [ agent: Supervisor.AgentProcedure ← NIL,
instanceData: LONG POINTER TO UNSPECIFIED ← NIL ]
RETURNS [handle: Supervisor.SubsystemHandle];
```

This operation creates a new subsystem object and causes an agent procedure and a set of instance data to be associated with it. The returned subsystem handle typically is made available to the subsystem's clients. The agent procedure for the subsystem will be called when events happen, passing **instanceData** to it at that time.

A subsystem is deleted by

```
Supervisor.DeleteSubsystem: PROCEDURE [handle: Supervisor.SubsystemHandle];
```

```
Supervisor.InvalidSubsystem: ERROR;
```

InvalidSubsystem is raised if **handle** does not describe a valid subsystem. Clients must take care to not retain nor use the **SubsystemHandle** of a deleted subsystem.

Operations are provided for declaring the dependency relationships between subsystems, and for inquiring about current dependencies:

```
Supervisor.AddDependency: PROCEDURE [client, implementor: Supervisor.SubsystemHandle];
```

```
Supervisor.CyclicDependency: ERROR;
```

```
Supervisor.RemoveDependency: PROCEDURE [client, implementor:
Supervisor.SubsystemHandle];
```

```
Supervisor.NoSuchDependency: ERROR;
```

AddDependency declares that **client** is directly dependent on **implementor** and directly uses its services. Typically, this declaration is made because a client subsystem needs to act on some event either before or after the subsystems which he depends on act on it. Duplicate direct dependencies are ignored. If **implementor** is already registered as being directly or indirectly dependent on **client**, **CyclicDependency** is raised. If **client** or **implementor** do not describe a valid subsystem, **Supervisor.InvalidSubsystem** is raised.

RemoveDependency declares that **client** is no longer directly dependent on **implementor**. If **client** was not directly dependent on **implementor**, **NoSuchDependency** is raised. If **client** or **implementor** do not describe a valid subsystem, **Supervisor.InvalidSubsystem** is raised.

Supervisor.DependsOn: PROCEDURE [client, implementor: Supervisor.SubsystemHandle]
RETURNS [BOOLEAN];

DependsOn returns **TRUE** if and only if **client** is directly or indirectly dependent on **implementor**. If either **client** or **implementor** does not describe a valid subsystem, **Supervisor.InvalidSubsystem** is raised.

When an event happens, the client program that caused the event notifies the registered subsystems with the following operation:

Supervisor.NotifyAllSubsystems: PROCEDURE [event: Supervisor.Event,
eventData: LONG POINTER TO UNSPECIFIED, whichFirst: Supervisor.ClientsImpls];

Supervisor.ClientsImpls: TYPE = {clients, implementors};

This operation calls the agent procedures of all subsystems. If **whichFirst** is **clients**, a subsystem is notified only after all of its clients have been notified. If **whichFirst** is **implementors**, a subsystem is notified only after all of its implementors have been notified. See the definition of **AgentProcedure** for a description of **eventData**. If a subsystem handle does not describe a valid subsystem, **Supervisor.InvalidSubsystem** is raised.

Caution: No client of Tajo, CoPilot, or the Development Environment, versions 11.0, should call **NotifyAllSubsystems**. It will cause these systems to crash or hang.

For events which are only of interest to a separable set of subsystems and for which it is desired to avoid swapping in the code of all agent procedures, **NotifyRelatedSubsystems** may be used.

Supervisor.NotifyRelatedSubsystems: PROCEDURE [event: Supervisor.Event,
eventData: LONG POINTER TO UNSPECIFIED, which, whichFirst: Supervisor.ClientsImpls,
subsystem: Supervisor.SubsystemHandle];

This operation calls the agent procedures of all subsystems which are directly or indirectly clients or implementors of **subsystem**. For **which** equal to **clients**, it calls all agent procedures which are direct or indirect clients of **subsystem**. For **which** equal to **implementors**, it calls all agent procedures which are the direct or indirect implementors of **subsystem**. For **whichFirst** equal to **clients**, it visits a subsystem only after all of its clients have been visited. For **whichFirst** equal to **implementors**, it visits a subsystem only after all of its implementors have been visited. See the definition of **AgentProcedure** for a

description of **eventData**. If **subsystem** does not describe a valid subsystem, **Supervisor.InvalidSubsystem** is raised.

Caution: **NotifyRelatedSubsystems** is not implemented in Pilot 11.0.

For events which are only of interest to the immediate clients or implementors of a subsystem and for which it is desired to avoid swapping in the code of all agent procedures, **NotifyDirectSubsystems** may be used.

Supervisor.NotifyDirectSubsystems: PROCEDURE [event: Supervisor.Event,
eventData: LONG POINTER TO UNSPECIFIED ← NIL, which: Supervisor.ClientsImpls,
subsystem: Supervisor.SubsystemHandle];

This operation calls the agent procedures of all subsystems which are directly related to **subsystem**. For **which** equal to **clients**, it calls the agent procedures of all subsystems which are direct clients of **subsystem**. For **which** equal to **implementors**, it calls the agent procedures of all subsystems which are direct implementors of **subsystem**. See the definition of **AgentProcedure** for a description of **eventData**. If **subsystem** does not describe a valid subsystem, **Supervisor.InvalidSubsystem** is raised.

2.6.3 Exception handling

Handling recoverable error conditions encountered during an enumeration of subsystems requires some special consideration. Exceptions in Mesa are usually handled by signals. In the context of the Supervisor, these are not appropriate since the subsystems are enumerated sequentially, not recursively, and therefore the previously-invoked agent procedures are not in a position to catch a signal or an UNWIND.

Thus, the following procedure is suggested: The agent detecting an error condition would signal an error to the caller of **NotifyxSubsystems**. That caller would catch the signal, unwind, and then call **NotifyxSubsystems** for an inverse event to the one being aborted. Thus, each agent would then be given the chance to back out of any actions he had taken. If there is no naturally-occurring inverse event, an artificial one can be defined specifically for backing out of particular kinds of aborted events. In some cases, a two-phase protocol may be necessary to handle an event properly.

If no special information needs to be communicated while aborting an enumeration, the following signal may be used:

Supervisor.EnumerationAborted: ERROR;

The caller of the enumeration should catch it.

2.7 General object allocation

ObjAlloc: DEFINITIONS...;

This section describes the facility used to control the allocated/free state of a collection of objects. A typical application of this facility would be a storage allocator using **ObjAlloc** to manage its underlying database.

2.7.1 Basic types

```
ObjAlloc.AllocFree: TYPE = MACHINE DEPENDENT {free(0), alloc(1)};  
  
ObjAlloc.AllocationPool: TYPE = PACKED ARRAY [0..0] OF ObjAlloc.AllocFree;  
  
ObjAlloc.AllocPoolDesc: TYPE = RECORD [allocPool: LONG POINTER TO ObjAlloc.AllocationPool,  
poolSize: ObjAlloc.ItemCount];  
  
ObjAlloc.Interval: TYPE = RECORD [first: ObjAlloc.ItemIndex, count: ObjAlloc.ItemCount];  
  
ObjAlloc.ItemIndex: TYPE = LONG CARDINAL;  
  
ObjAlloc.ItemCount: TYPE = LONG CARDINAL;
```

An **ObjAlloc.AllocationPool** describes the allocated/free state of an ordered set of objects. Each object is identified by a name, called an **ObjAlloc.ItemIndex**. The location and size of an **ObjAlloc.AllocationPool** is given by an **ObjAlloc.AllocPoolDesc**.

Note: The location must be word aligned, and the size is given in terms of the number of objects in the pool.

An **ObjAlloc.Interval** describes a range of objects by giving the **ObjAlloc.ItemIndex** of the first object, and the number of objects in the range.

2.7.2 Basic procedures and errors

```
ObjAlloc.Allocate: PROCEDURE [ pool: ObjAlloc.AllocPoolDesc, count: ObjAlloc.ItemCount,  
willTakeSmaller: BOOLEAN ← FALSE] RETURNS [interval.ObjAlloc.Interval];
```

```
ObjAlloc.Error: ERROR [error: ObjAlloc.ErrorType];
```

```
ObjAlloc.ErrorType: TYPE = {insufficientSpace, invalidParameters};
```

ObjAlloc.Allocate finds, and marks as allocated, a range of **count** objects. If **willTakeSmaller** is **FALSE** and **count** contiguous objects can not be found, **ObjAlloc.Error[insufficientSpace]** is raised. If **willTakeSmaller** is **TRUE**, **ObjAlloc.Allocate** will allocate the largest range of objects whose size does not exceed **count**. In this case, **ObjAlloc.Error[insufficientSpace]** will only be raised if no free objects can be found. In either case, the returned range is guaranteed to be the range with the smallest first name that meets the needs inferred by **count** and **willTakeSmaller**.

```
ObjAlloc.ExpandAllocation: PROCEDURE [ pool: ObjAlloc.AllocPoolDesc,  
where: ObjAlloc.ItemIndex, count: ObjAlloc.ItemCount,  
willTakeSmaller: BOOLEAN ← FALSE] RETURNS [extendedBy.ObjAlloc.ItemCount];
```

An allocated range can be expanded using **ObjAlloc.ExpandAllocation**. If the objects [**where..where + count**] are all free, they are marked as allocated, and **extendedBy** is set to **count**. If only the objects [**where..where + countFree**] are free, where **0 <= countFree < count**, the result depends upon the value of **willTakeSmaller**. If **willTakeSmaller** is **FALSE**, **extendedBy** is returned as zero and no objects are marked

allocated. If **willTakeSmaller** is **TRUE**, the objects [**where..where + countFree**) are marked as allocated and **extendedBy** is returned as **countFree**.

ObjAlloc.Free: PROCEDURE [pool: ObjAlloc.AllocPoolDesc, interval: ObjAlloc.Interval,
validate:BOOLEAN ← TRUE];

ObjAlloc.AlreadyFreed: ERROR [item: ObjAlloc.ItemIndex];

A range of objects is freed by calling **ObjAlloc.Free**. If not all of the named objects are contained in **pool**, **ObjAlloc.Error[invalidParameters]** is raised and no objects are marked free. If **validate** is **TRUE** then an attempt to free an already free object results in the signal **ObjAlloc.AlreadyFreed[item]** being raised, with **item** as the smallest index of a free object in the interval. No objects are freed in this case. If **validate** is **FALSE**, the specified objects are marked as free with no checking performed.

ObjAlloc.InitializePool: PROCEDURE [pool: ObjAlloc.AllocPoolDesc, initialState:
ObjAlloc.AllocFree];

An **ObjAlloc.AllocationPool** may be initialized by calling **ObjAlloc.InitializePool**. It will set the initial state of all of the objects in the pool to the specified state.

Note: In any call to **ObjAlloc.Allocate**, **ObjAlloc.ExpandAllocation**, **ObjAlloc.Free**, or **ObjAlloc.InitializePool**, an **ADDRESS FAULT** may result if any part of the allocation pool is unmapped. Additionally, it is the clients responsibility to serialize access to the database. **ObjAlloc** provides no serialization.



Streams

Stream: DEFINITIONS . . . ;

The *Stream Facility* described in this section provides to Pilot clients a convenient, efficient, device- and format-independent interface for *sequential* access to a stream of data. In particular:

- It provides a vehicle by which processes or subsystems can communicate with each other, whether they reside on the same system element or on different system elements.
- It permits processes or subsystems to transmit arbitrary data to or from storage media in a device-independent way.
- It defines a standard way for transforming the detailed interface for a device into a uniform, high level interface which can be used by other client software.
- It provides an environment for implementing simple transformations to be performed on the data as it is being transmitted.
- It provides optional access to and control over the mapping of data onto the physical format of the storage or transmission medium being used.

The stream package provides several facilities, not all of which may be important to an individual client. First, there is the *stream interface*, the set of procedures and data types by which a client actually controls the transmission of a stream of information. Each of the operations of the stream interface takes as a parameter a **Stream.Handle** which identifies the particular stream being accessed. Second, the stream package defines the concepts of *transducer* and *filter*. A transducer is a software entity (*e.g.*, module or configuration) which implements a stream connected to a specific device or medium. A filter also implements a stream, but only for the purpose of transforming, buffering, or otherwise manipulating the data before passing it on to another stream. Transducers and filters may be provided either by Pilot or by clients. Third, the stream package provides a standard way of concatenating a sequence of filters (usually terminated with a transducer) to form a compound stream called a *pipeline*. A pipeline is accessed by means of the normal stream operations, and causes a sequence of separate transformations to be applied to data

flowing between the client program at one end and the physical storage (or transmission) medium at the other.

Pipelines permit clients to interpose new stream manipulation programs (filters and transducers) between clients (producers and consumers of data) without modifying the interfaces seen by the clients. For example, a data format conversion program can obtain its data either from a magnetic cassette or from a floppy disk, using the same stream interface, and hence the same program logic, for both. Similarly, filters performing such functions as code conversion, buffering, data conversion, and encryption, may be inserted into a pipeline without affecting the way the client sends and receives data through the stream interface.

The stream facility transmits arbitrary data, regardless of format and without prejudice to its type or characteristics. The data may comprise a sequence of bytes, words, or arbitrary Mesa data structures. The stream facility does not presume or require the encoding of information according to any particular protocol or convention. Instead, it permits clients to define their own protocols and standards according to their own needs.

In this chapter, sections 3.1, 3.2, and 3.3 will be of interest to all clients. Section 3.4 will be of interest only to those clients wishing to control the physical record characteristics of a particular stream. Section 3.5 will be of interest only to those clients wishing to implement their own filters or transducers. In addition, the clients of a particular stream type (*e.g.*, disk, tape) will normally have to consult separate documentation regarding the details of that kind of stream.

3.1 Semantics of streams

The stream facility supports transmission of a sequence of eight-bit bytes. This sequence may be divided into identifiable *subsequences*, each of which has its own *subsequence type*.

Stream.Byte: TYPE = Environment.Byte;

Environment.Byte: TYPE = [0..256];

Stream.SubSequenceType: TYPE = [0..256];

A subsequence may be null: *i.e.*, it may be of zero length and contain no bytes but still contain the **SubSequenceType** information. This information allows all subsequences to be easily identified and separated from each other while shielding clients from the bothersome problems of control-codes (*i.e.*, embedding control codes into the stream, making them transparent, and building a parser to implement such transparency).

Additionally, an *attention* flag may be inserted into a stream sequence. Attention flags are transmitted through the stream as quickly as possible, possibly bypassing bytes and changes in **SubSequenceType** which were transmitted earlier but which are still in transit. This provides a simple mechanism for implementing breaks (similar to the "attention-key" of many time-sharing systems). A byte of data is associated with an attention flag for the use of client protocols. Note that the attention flag and the data byte occupy a byte in the stream sequence.

Streams have no intrinsic notion of the bytes passing through them being grouped into physical records. The client program can completely ignore physical record structure and

is thus relieved of the burden of dealing with the associated packing and unpacking problems. If, however, it becomes necessary to control or determine the underlying physical record structure, as determined by the particular storage (or transmission) medium, the interface provides extended facilities which allow this.

All of the procedures described here are synchronous. That is, an input operation does not return until the data is actually available to the client, and an output operation does not return until the data has been accepted by the stream and client buffers may be reused. Note, however, that a stream component *may* do internal buffering and that the acceptance of data means only that the stream component itself has a correct copy and is in a position to proceed asynchronously to write or send it.

Streams in Pilot are inherently full duplex. Separate processes may be transmitting and receiving simultaneously. The stream interface does *not* guarantee mutual exclusion among different processes attempting to access the same stream. However, individual transducers or filters may restrict themselves to half duplex operation and may implement such mutual exclusion or more elaborate forms of synchronization as is appropriate. Documentation for such filters and transducers should be consulted on a case-by-case basis for details.

3.2 Operations on streams

The stream interface provides operations for sending and receiving data, for sending state information, and for dealing with stream positions. In addition, a **Delete** operation is provided to delete a stream. A create operation is not provided. Streams are only created by individual stream components, namely, pipelines, transducers and filters.

A client program identifies a particular instance of the stream interface by means of a **Stream.Handle**.

Stream.Handle: TYPE = ...;

A **Stream.Handle** identifies an object (see §3.5.1) which embodies all of the information concerning the transfer of data to or from the client program via stream operations. It is passed as a parameter to each of the data transmission operations of the following sections to specify the stream to which the operations apply.

A stream may be deleted by the operation:

Stream.Delete: PROCEDURE [sH:Stream.Handle];

The client must ensure that there are no outstanding references to the stream being deleted. Failure to observe this caution will result in unpredictable effects.

3.2.1 GetBlock and PutBlock

The principal operations for transferring blocks of data are **Stream.GetBlock** and **Stream.PutBlock**. Both are inline procedures. Each of these takes a parameter specifying the block of virtual memory to or from which bytes are to be transmitted.

Stream.Block: TYPE = Environment.Block;

```
Environment.Block: TYPE = RECORD [
  blockPointer: LONG POINTER TO PACKED ARRAY [0..0] OF Environment.Byte,
  startIndex, stopIndexPlusOne: CARDINAL];
```

A **Block** describes a section of memory which will be the source or sink of the bytes transmitted. The section of memory described is a sequence of bytes (not necessarily word aligned) which must lie entirely within a mapped space. **blockPointer** selects a word such that a **startIndex** of zero would select the left byte of that word (*i.e.*, bits 0 - 7). The selected block consists of the bytes **blockPointer**[*i*] for *i* in [**startIndex**..**stopIndexPlusOne**]. Notice that a **Block** cannot describe more than $2^{16}-1$ bytes or $2^{15}-1$ words. A **Stream.Block** can describe any part of virtual memory.

Some of the operations of this section and the next may cause signals to be generated. If such a signal is resumed, transmission continues from where it left off so that any changes made by the catch phrase to the **Block** record or to the input options (see below) are ignored. If, however, such a signal is RETRYed, the next byte of the stream sequence is transmitted to or from the byte specified by the current value of the **Block** record or input options, either of which might have been updated by the catch phrase. In no case is the stream sequence itself "backed up". Bytes previously received from input are not re-received, and bytes previously transmitted on output are not withdrawn.

The primary block input operation is **Stream.GetBlock**.

```
Stream.GetBlock: PROCEDURE [sH: Stream.Handle, block: Stream.Block]
  RETURNS [bytesTransferred: CARDINAL, why: Stream.CompletionCode,
           sst: Stream.SubSequenceType];
```

```
Stream.CompletionCode: TYPE = {normal, endRecord, sstChange, endOfStream,
                                attention, timeout};
```

The parameter **block** describes the virtual memory area into which the bytes will be placed. **GetBlock** does not return until the input is terminated. Its exact behavior, however, is controlled by a set of input options which may be set by the client using the following operation:

```
Stream.SetInputOptions: PROCEDURE [sH: Stream.Handle, options: Stream.InputOptions];
```

```
Stream.InputOptions: TYPE = RECORD [
  terminateOnEndRecord ← FALSE, signalLongBlock ← FALSE, signalShortBlock ← FALSE,
  signalSSTChange ← FALSE, signalEndOfStream ← FALSE, signalAttention ← FALSE,
  signalTimeout ← TRUE, signalEndRecord: BOOLEAN ← FALSE];
```

```
Stream.defaultInputOptions: Stream.InputOptions = [];
```

SetInputOptions controls exactly how **GetBlock** terminates and what signals it generates. Ordinarily (*i.e.*, with the parameter **options** set to **defaultInputOptions**) the transmission will not terminate until the entire block of bytes is filled unless there is a timeout. However, under the exceptional conditions described in §3.4, the transmission may terminate before the **block** is filled and may also result in a signal. In all cases the procedure will return the actual number of bytes transferred, a **CompletionCode** indicating the reason for termination, and the latest **SubSequenceType** encountered. The

input operation may conveniently be restarted where it left off by first adding the result **bytesTransferred** to **block.startIndex** to update the record describing the block of bytes.

In general, any status that may be returned from **GetBlock** may also be signalled, and the option to do so is available through **InputOptions**. A catch phrase for these signals must not attempt any other stream operations using the same **Stream.Handle**, for this will corrupt the internal state information maintained for the stream.

Three circumstances which *always* suspend the transmission of data before the **block** is filled are the detection of a change in **SubSequenceType**, the detection of an **attention**, and the detection of the end of the stream. In the first case, if the input option **signalSSTChange** is **FALSE** (the default) then the procedure **GetBlock** terminates immediately and returns the number of bytes transferred, with **why = sstChange**, and **sst** set to the new value of the **SubSequenceType**. If the input option **signalSSTChange** is **TRUE** then the signal

Stream.SSTChange: SIGNAL [sst: Stream.SubSequenceType, nextIndex: CARDINAL];

is generated. The parameter **sst** specifies the new **SubSequenceType**, and the parameter **nextIndex** specifies the byte index within the **block** where the first byte of the new subsequence will be placed. This signal may be resumed, and the effect is to continue the data transmission as though the change in **SubSequenceType** had not occurred (*i.e.*, in the same block of bytes).

If an attention is detected in the byte stream, the **GetBlock** terminates immediately and returns immediately with the number of bytes transferred and with **why = attention**. If the input option **signalAttention** is **TRUE** then the signal

Stream.Attention: SIGNAL [nextIndex: CARDINAL];

is generated. The parameter **nextIndex** specifies the byte index within the **block** where the position within the block where the next byte, the attention byte, would be placed. This signal may be resumed, and the effect is to continue the data transmission as though the **Attention** had not occurred (*i.e.*, in the same block of bytes).

A catch phrase for these signals must not attempt any other stream operations using the same **Stream.Handle**, for this will corrupt the internal state information maintained for the stream.

Implementation of the end-of-stream feature is strictly transducer and filter specific, and optional. Transducer and filter implementors may implement an end-of-stream mechanism using any protocol they desire. When putting together a pipeline from a transducer and filters, great care needs to be taken to preserve the end-of-stream feature through all the stream components. If the input option **signalEndOfStream** is **FALSE** (the default) and the stream component detects that the end-of-stream has occurred then the procedure **GetBlock** terminates immediately and returns the number of bytes transferred, with **why = endOfStream**. If the input option **signalEndOfStream** is **TRUE** and the stream component detects that the end-of-stream has occurred then the signal

Stream.EndOfStream: SIGNAL [nextIndex: CARDINAL];

is generated. The parameter **nextIndex** specifies the byte index immediately following the last byte of the stream sequence filled in a client's block.

Stream component implementors may provide special procedure calls in order to actively cause a stream to be terminated.

Semantics of the end-record feature are also transducer and filter specific. Furthermore, all transducers may not preserve the same semantics across the transmission medium. In any case, all notion of end-record processing may be suppressed by setting **terminateOnEndRecord** **FALSE** (the default). If the input option **terminateOnEndRecord** is **TRUE** and **signalEndRecord** is **FALSE** (the default) and the stream component detects that the end-record has occurred then the procedure **GetBlock** terminates immediately and returns the number of bytes transferred, with **why = endRecord**. If the input option **signalEndRecord** is **TRUE** and the stream component detects that the end-record has occurred then the signal

Stream.EndRecord: SIGNAL [nextIndex: CARDINAL];

is generated. The parameter **nextIndex** specifies the byte index immediately following the last byte of the stream sequence filled in a client's block.

The principal block output operation is **Stream.PutBlock**.

**Stream.PutBlock: PROCEDURE [sH: Stream.Handle, block: Stream.Block,
endRecord: BOOLEAN \leftarrow FALSE];**

This operation is analogous to **Stream.GetBlock**. The parameter **block** describes the area of virtual memory from which information is transmitted. This procedure returns only after the data has been accepted by the stream, at which time the client may reuse **block**. If the client is ignoring record boundaries (the default), **endRecord** should be set to **FALSE**. Otherwise, see the section on controlling physical record characteristics, §3.4.

Stream operations have the right to discard empty blocks, hence a **PutBlock** operation specifying a **block** of length zero *may* be a no-op even if **endRecord** is **TRUE**.

3.2.2 Additional data transmission operations

In addition to **GetBlock** and **PutBlock**, the following operations are provided to permit the sending and receiving of individual bytes, characters and words. All but **SendNow** are inline procedures. They are supplied so that *some* streams can provide byte or character or word operations in a more efficient manner than is possible with **GetBlock** or **PutBlock**. The documentation for individual streams should be consulted for detailed performance information.

Stream.GetByte: PROCEDURE [sH: Stream.Handle] RETURNS [byte: Stream.Byte];

Stream.GetChar: PROCEDURE [sH: Stream.Handle] RETURNS [char: CHARACTER];

Stream.GetWord: PROCEDURE [sH: Stream.Handle] RETURNS [word: Stream.Word];

Stream.Word: TYPE = Environment.Word;

GetByte and **GetChar** operations get the next **Byte** or **CHARACTER** from the stream sequence and return it just as a call upon **Stream.GetBlock** specifying a **Block** containing one byte would. The **GetWord** operation gets the next **Word** from the stream sequence and returns it just as a call upon **GetBlock** specifying a **Block** containing **Environment.bytesPerWord** bytes would. In all three cases, the effect is as if the input options to **GetBlock** had been **signalShortBlock**, **signalLongBlock**, **signalAttention**, **signalEndRecord** and **terminateOnEndRecord = FALSE**, and **signalEndOfStream**, **signalTimeout** and **signalSSTChange = TRUE**. Thus, these operations may result in the signal **SSTChange**, **EndOfStream** or **Stream.TimeOut** (see §3.2.4).

Note: When any of the signals are generated when processing a **GetWord** and **nextIndex** is an odd value, the two communicating processes are responsible for the outcome.

Stream.PutByte: PROCEDURE [sh: Stream.Handle, byte: Stream.Byte];

Stream.PutChar: PROCEDURE [sh: Stream.Handle, char: CHARACTER];

Stream.PutWord: PROCEDURE [sh: Stream.Handle, word: Stream.Word];

Stream.PutString: PROCEDURE [sh: Stream.Handle, string: LONG STRING, endRecord ← FALSE];

The **PutByte** and **PutChar** operations transmit the **Byte** or **CHARACTER** to the medium just as a call on **Stream.PutBlock** specifying a **Block** containing one byte would. The **PutWord** operation transmits the next **Word** to the medium just as a call on **PutBlock** specifying a **Block** containing **Environment.bytesPerWord** bytes would. In the first three cases, the effect is as if **endRecord** is set to **FALSE** in the call to **PutBlock**. **PutString** transmits the bytes described by **string** to the medium.

Stream.SendNow: PROCEDURE [sh: Stream.Handle, endRecord ← FALSE];

This operation flushes the stream sequence. It guarantees that all information previously output (by means of **PutBlock**, **PutByte**, **PutChar**, **PutWord**, **PutString**, or **SetSST**) will actually be transmitted to the medium (perhaps asynchronously). The default implementation of this procedure is equivalent to a call on **Stream.PutBlock** specifying a **Block** containing no bytes and **endRecord** set to **TRUE** (see §3.4). Client programs should call **SendNow** at appropriate times to ensure that the bytes and changes in **SubSequenceType** have actually been sent and are not buffered internally within the stream, awaiting additional output operations.

Through use of the **endRecord** parameter, **SendNow** may apply transducer or filter specific semantics to the transmission of the data, such as the idea of a *logical record*. A logical record may be a collection of one or more physical records. The logical record boundaries can be detected by the receiving client by proper setting of **terminateOnEndOfRecord** and perhaps **signalEndRecord** in the streams's **InputOptions**.

3.2.3 Subsequence types

The subsequence type of a stream may be changed by

Stream.SetSST: PROCEDURE [sh: Stream.Handle, sst: Stream.SubSequenceType];

All subsequent bytes transmitted on the stream have the indicated **SubSequenceType**. Even if the subsequent sequence of bytes is null (*i.e.*, a call on **SetSST** is immediately followed by another), the **SubSequenceType** change demanded by this call will still be available to the receiver of the stream sequence.

SubSequenceTypes are intended to be used to delineate different kinds of information flowing over the same stream (*e.g.*, to identify control information, indicate end-of-file, etc.). The interpretation of a **SubSequenceType** value is a function of the particular stream.

A **SetSST** operation specifying a **SubSequenceType** identical to the previous **SubSequenceType** is a no-op. Otherwise, **SetSST** always has the side effect of completing the current physical record, as explained in §3.4.

3.2.4 Attention flags

The following operation causes an attention flag and an associated byte of data to be transmitted via the stream facility.

Stream.SendAttention: PROCEDURE [sH: Stream.Handle, byte: Stream.Byte];

Note that the attention flag and the data byte occupy a byte in the stream sequence. The attention is sent as both an in-band and out-of-band signal. The out-of-band attention is not necessarily transmitted in sequence, but may bypass bytes and changes in **SubSequenceType** which were transmitted before it. **byte** is used by the client protocol to transmit other information regarding this attention.

The following operation awaits the arrival of an attention flag.

Stream.WaitForAttention: PROCEDURE [sH: Stream.Handle] RETURNS [Stream.Byte];

When the out-of-band attention is received on stream **sH**, this procedure returns the byte of data associated with the attention. It is the responsibility of the client program to determine the appropriate action to take. If more than one attention flag has been sent, these will be queued by the stream. Each return from a call on **WaitForAttention** corresponds to precisely one attention sent by **SendAttention**.

When the in-band attention is received on stream **sH**, the effect depends upon the setting of the **InputOptions**. If **signalAttention** is **FALSE**, the operation terminates with a completion code of **attention**. The next byte in the stream is the byte passed to **SendAttention**. If the input options specify **signalAttention** as **TRUE**, the signal **Attention** is raised with the index pointing in the current block to the byte passed to **SendAttention**.

WaitForAttention is usually executed by a different process from that operating upon the stream. It returns as soon as the attention is received, whether or not all of the bytes preceding it in the stream have been transferred.

3.2.5 Timeouts

Any of the operations of this section (except **SendAttention** and **WaitForAttention**) may fail to complete within a reasonable amount of time due to external conditions. In such a case the following signal is generated:

```
Stream.TimeOut: SIGNAL [nextIndex: CARDINAL];
```

The parameter of this signal indicates the position within the block of bytes where the next byte would be placed. This signal may be resumed.

If this signal is **RETRYed** all previously received data may be lost. This is because it is likely that a stream component is performing internal buffering (transferring data from its buffer into the client's block), and the action of **RETRYing** the signal may not tell the component that it must refill the client's block. Even if the component was informed of this fact, it may have discarded data already transferred into the client's block from its internal buffer.

A catch phrase for this signal must not attempt any other stream operations using the same **Stream.Handle**, for this will corrupt the internal state information maintained for the stream.

The timeout value for the stream may be read and altered by using the **getTimeout** and **setTimeout** procedures in the **Stream.Object** (section 3.5.1).

```
msecs ← sH.getTimeout[sH];
```

```
sH.setTimeout[sH, msecs];
```

3.2.6 Stream positioning

For those streams which may be accessed randomly, the position of a stream may be determined with the procedure

```
Stream.GetPosition: PROCEDURE [sH: Stream.Handle]  
RETURNS [position: Stream.Position];
```

```
Stream.Position: TYPE = LONG CARDINAL;
```

The value returned is the byte index of the next byte to be read from or written in the file.

The position of a stream may be set with the procedure

```
Stream.SetPosition: PROCEDURE [sH: Stream.Handle, position: Stream.Position];
```

3.3 Creating streams

Pilot provides no general operations for creating streams. The main reason for this is that the components of a stream (pipelines, transducers, and filters) must be able to take arbitrary parameters at the time they are created. It is not possible for Pilot to specify a general interface for their creation without either compromising the basic type-safeness of Mesa or constraining the flexibility and power of client-provided streams. Thus, the create

function is implemented on a case-by-case basis, and clients must therefore refer to documentation for individual stream components for the correct interface for this operation. Specifications for Pilot-provided transducers and filters are included in § 3.6. In this section, the general style is illustrated by means of hypothetical examples.

For example, if a utility package implements a transducer to a magnetic cassette reader, it is obligated to provide a means by which other clients can create instances of that transducer, use them, and later delete them. Suppose the name of the interface module providing this function is **CassetteStream**. Then it would provide the following operation:

```
CassetteStream.Create: PROCEDURE [ --optional parameters-- ]
RETURNS [Stream.Handle, --optional other results--];
```

A client wishing to use the stream interface to access this device would thus call **CassetteStream.Create**, then use the **Stream.Handle** returned from it as parameter to the stream operations of this chapter. When the stream was no longer needed, it would be deleted by calling **Stream.Delete**.

Similarly, a security package providing, say, an encryption facility might implement this by means of a filter for a stream. In this case, the interface module might be called **EncryptionFilter**, and it would provide the following operation:

```
EncryptionFilter.Create: PROCEDURE [Stream.Handle, --optional other parameters-- ]
RETURNS [Stream.Handle, --optional other results--];
```

The client could easily couple an instance of this filter with the transducer above. This is done by calling **EncryptionFilter.Create**, passing as a parameter the **Stream.Handle** returned from **CassetteStream.Create**. Then the **Stream.Handle** returned from **EncryptionFilter.Create** would be the one used in **GetBlock**, **PutBlock**, and the other operations of §3.2. The net effect would be stream components which, on input, read bytes from the cassette reader, decrypt them, and pass them to the client and which, on output, encrypt the bytes supplied by the client and write them on the cassette.

In general, a procedure creating a filter accepts one **Stream.Handle** as a parameter and returns another as its result. Thus, several filters, each implementing a simple transformation, may be concatenated together to implement a more interesting transformation on the stream sequence. The parameter passed to each one is the result returned from the adjacent one. Such a concatenation, called a *pipeline*, is illustrated in Figure 3.3a.

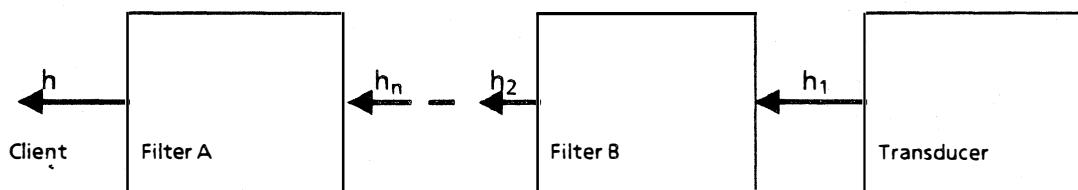


Figure 3.3a

This diagram illustrates how each **Stream.Handle** returned from a transducer or filter is passed as a parameter to the next adjacent filter, and how the last one is used directly by the client. In particular, h_1 is returned from the procedure which creates **Transducer**. It is passed to the procedure which creates **Filter B**, returning h_2 . This is passed, in turn, to the next filter, and so on, until h_n is returned and passed to **Filter A**. **Filter A** is the last one in the pipeline, and its **Stream.Handle**, h , is returned to the client.

Figure 3.3b shows the flow of data through the pipeline and the use of the various **Stream.Handles** as a result of a client call on **Stream.GetBlock** (calls on other data transmission operations are analogous).

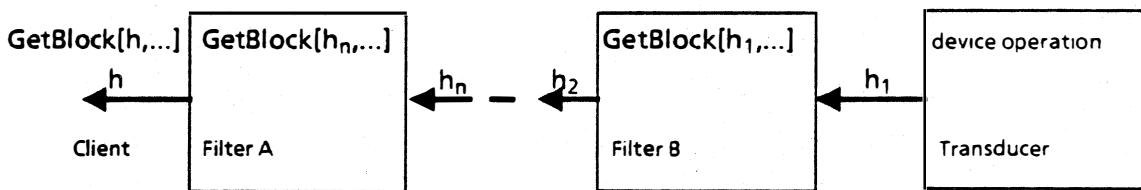


Figure 3.3b

Here, the client calls **Stream.GetBlock[h,...]**, which is transformed by the stream interface into an appropriate call on **Filter A**. **Filter A**, in turn, calls **Stream.GetBlock[h_n,...]**, which is passed to the next filter in the pipeline, and so on, until eventually a call is made on **Stream.GetBlock[h_2, ...]**. This is transformed into a call on **Filter B**, which then calls **Stream.GetBlock[h_1,...]**, to invoke **Transducer**, which actually operates the device.

Note that the only difference between a transducer and a filter is that a transducer interfaces to some device or channel, while a filter interfaces to another stream and, thus, indirectly to another filter or transducer.

Note also that the client can construct a pipeline "manually," by tediously assembling the various components, instantiating each of them, and binding them together. However, a pipeline can also be presented as an integrated package, already assembled. For example, the two components described above may have been assembled into a pipeline called **EncryptingCassetteStream**. This pipeline might then provide the following operation, which clients can call to create an instance of this pipeline:

```
EncryptingCassetteStream.Create: PROCEDURE[ --optional parameters-- ]
RETURNS [Stream.Handle, --optional other results--];
```

The client of such a stream would merely invoke this procedure to create the stream without having to bother about finding and putting together the individual components.

3.4 Control over physical record characteristics

Most of the time, the client will not wish to know about how the data in a stream sequence is divided into physical records for recording or transmission. For some applications, however, this is of vital importance. The stream facility has been designed so that the details of the physical encoding can be ignored when desired, or completely known and controlled when that is necessary. On output, complete control of the placement of bytes in

physical records can be achieved for most media. On input, complete information is available about how the bytes were arranged in physical records.

These facilities to control the placement of bytes on physical records are *not* meant to be used as a means of transmitting information. In particular, a transducer might suppress or generate empty physical records and will necessarily partition oversize "physical" records into smaller ones. Any filter may rearrange (or completely obliterate) physical record boundaries. Documentation for the individual transducer or filter and for the individual transmission or storage medium should be consulted for full details.

The output and input cases will be treated separately. On output, bytes will be placed in turn into the same physical record until one of the following events occurs:

1. The **SendNow** procedure is called. It has the side effect of causing the current record to be sent. The next byte output will begin a new physical record. This is the main mechanism for controlling physical record size on output.

A **SendNow** with **endRecord TRUE** may apply further transducer or filter dependent semantics, such as *end of logical record*.

2. A **PutBlock** procedure is called with an **endRecord** parameter of **TRUE** (this is equivalent to a **SendNow** with **endRecord TRUE**). After the transmission of this block of bytes, the current physical record is ended. If, at this point, the physical record is at its maximum size (see 5. below), an empty record will not be transferred.
3. A **SetSST** procedure has been called. The first byte of a new subsequence always begins a new record and has the new **SubSequenceType**. This may cause the previous record to be sent.
4. Enough bytes have been output to fill the physically maximal record. At this point the record will be written and a new record started. This maximum number is a function of the medium being written, hence documentation concerning the medium must be consulted to determine this value.
5. Some other device-dependent event, such as a timeout, occurs. In this case, a buffer may be flushed automatically. Details are documented with individual transducers.

On input, bytes will be placed in turn into the record until one of the following events occurs:

1. The end of the logical record is reached, and the input option **terminateOnEndRecord** is **TRUE**.

The end of the logical record is reached at the same time that the block of bytes described in the **Block** record is exhausted. In this case, neither of the signals **ShortBlock** and **LongBlock** is generated. If the input option **terminateOnEndRecord** is **TRUE**, then **why** is set to **endRecord**; otherwise, it is set to **normal**.

In any case, if the input option **signalEndRecord** is **TRUE** and the logical record has just been exhausted, then the following signal is generated.

Stream.EndRecord: SIGNAL[nextIndex: CARDINAL];

This signal indicates by **nextIndex** the position within the block of bytes where the next byte will be placed. If it is resumed, transmission continues as if it had not been generated.

A catch phrase for this signal must not attempt any other stream operations using the same **Stream.Handle**, for this will corrupt the internal state information maintained for the stream.

2. The end of a physical record is reached, the block of bytes described in the **Block** record is not exhausted, and **signalLongBlock** is **TRUE**.

If **signalLongBlock** is **TRUE**, the following signal is generated:

Stream.LongBlock: SIGNAL [nextIndex: CARDINAL];

This signal indicates by **nextIndex** the position within the block of bytes where the next byte will be placed. If it is resumed, transmission continues as if it had not been generated.

A catch phrase for this signal must not attempt any other stream operations using the same **Stream.Handle**, for this will corrupt the internal state information maintained for the stream.

3. The block of bytes described in the **Block** record is exhausted, the end of the physical record has not been reached, and the input option **signalShortBlock** has the value **TRUE**. At this time the input is terminated (without losing the subsequent bytes of the physical record, which are still available for reading by subsequent **GetBlock** operations), and the signal **Stream.ShortBlock** is generated.

Stream.ShortBlock: ERROR;

This signal may not be resumed.

The easiest approach is usually to establish a **Block** longer than the longest expected physical record and specify input options **signalLongBlock** as **FALSE**, **signalShortBlock** as **TRUE**, and **terminateOnEndRecord** as **TRUE**. At this point the transmission will cease with the entire contents of the physical record in the block of bytes, and the number of bytes transmitted will be returned as the result of the **GetBlock** procedure. In this way a signal will be generated only under unusual circumstances.

3.5 Transducers, filters, and pipelines

The stream package is designed so that clients can implement their own stream components (transducers, filters, and pipelines). The implementor of one of these has three different obligations to fulfill. First, he must design an interface (i.e., Mesa **DEFINITIONS** module) in the style described in the section about creating streams, §3.3, by which his clients create instances of that stream component. Such an interface (together with its accompanying implementation modules) is called a *stream component manager*. Second, he must provide a functional specification describing this interface and the detailed behavior of the stream component, including any specific signals, errors, parameters, etc., which it defines. Third, he must implement the actual component, if it is a filter or

transducer. (Pipelines are assumed to be composed of previously implemented components which already have their own component managers and documentation.)

This section describes the standards, data types, and operations to be used in defining a new stream component. It discusses, the precise interface which each instance of each filter or transducer must provide, and outlines a typical method for implementing a filter or transducer manager.

3.5.1 Representing filters and transducers

At run time, a filter or transducer is represented by sixteen procedures, a set of options and an instance data field so that clients may associate other data with a stream instance. The procedures execute in a common context to provide the data transmission operations of that filter or transducer. Descriptors for these procedures are stored in a record defined by the stream package, and pointed to by a **Stream.Handle**.

The procedures stored in **Object** must implement the semantics of the corresponding procedures (**GetByte**, **Put**, etc.) described in §3.2 on the stream **sH**. In particular, they must terminate according to the specifications of those sections and must generate the appropriate signals (**SSTChange**, **LongBlock**, **ShortBlock**, **EndOfStream**, **TimeOut**, **EndRecord**) as required.

Stream.Handle: TYPE = LONG POINTER TO Stream.Object;

```
Stream.Object: TYPE = RECORD [
  options: Stream.InputOptions,
  getByte: Stream.GetByteProcedure,
  putByte: Stream.PutByteProcedure,
  getWord: Stream.GetWordProcedure,
  putWord: Stream.PutWordProcedure,
  get: Stream.GetProcedure,
  put: Stream.PutProcedure,
  setSST: Stream.SetSSTProcedure,
  sendAttention: Stream.SendAttentionProcedure,
  waitAttention: Stream.WaitAttentionProcedure,
  delete: Stream.DeleteProcedure
  getPosition: StreamGetPositionProcedure
  setPosition: Stream.SetPositionProcedure
  sendNow: Stream.SendNowProcedure,
  clientData: LONG POINTER,
  getSST: Stream.GetSSTProcedure,
  getTimeout: Stream.GetTimeoutProcedure,
  setTimeout: Stream.TimeoutProcedure];
```

A client call on a Pilot stream operation is normally converted by the stream package into a call on the appropriate procedure named in the **Stream.Object** pointed to by the **Stream.Handle** parameter of that operation. Thus, it is the responsibility of the implementor of each filter and transducer to satisfy exactly the specifications of the stream package. Pilot assists in this task by utilizing the Mesa type checking machinery and by defining the uniform interface encapsulated by **Stream.Object**.

In this section, the meanings of the fields of **Stream.Object** are enumerated and a default **Stream.Object** described.

The **options** field specifies the currently valid input options for the stream.

options: **Stream.InputOptions;**

This field is set by **Stream.SetInputOptions** and its current value is passed as a parameter to the **get** procedure described below. Implementors of filters and transducers need not be concerned with maintaining or inspecting this field.

The **get** field specifies the block input procedure of the stream.

get: **Stream.GetProcedure;**

Stream.GetProcedure: **TYPE** =
PROCEDURE [SH: Stream.Handle, block: Stream.Block, options: Stream.InputOptions]
RETURNS [bytesTransferred: CARDINAL, why: Stream.CompletionCode,
sst: Stream.SubSequenceType];

In a filter, the body of a **GetProcedure** will typically contain one or more calls on **GetBlock**, **GetByte**, **GetChar**, or **GetWord** with a **Stream.Handle** parameter pointing to the next stream component in the pipeline (*i.e.*, the parameter passed at the time this filter was created). In a transducer, the body of a **GetProcedure** will typically have calls on input operations for the specific device being supported.

The **getByte** field specifies the byte input procedure of the stream.

getByte: **Stream.GetByteProcedure;**

Stream.GetByteProcedure: **TYPE** = PROCEDURE [SH: Stream.Handle]
RETURNS [byte: Stream.Byte];

The **getWord** procedure specifies the word input procedure of the stream.

getWord: **Stream.GetWordProcedure;**

Stream.GetWordProcedure: **TYPE** = PROCEDURE [SH: Stream.Handle]
RETURNS[word: Stream.Word];

The **put** field specifies the block output procedure provided by the filter or transducer.

put: **Stream.PutProcedure;**

Stream.PutProcedure: **TYPE** =
PROCEDURE [SH: Stream.Handle, block: Stream.Block, endRecord: BOOLEAN];

This procedure must regard the parameter **endRecord** = **TRUE** as an indication to flush any output buffers and actually initiate the physical transmission of information. It may suppress output requests specifying a block of no bytes provided that there is no previous output, change in **SubSequenceType**, or attention flag still waiting to be sent. This procedure may generate the signal **TimeOut** if necessary.

In a filter, the body of a **PutProcedure** will typically contain one or more calls on **PutBlock**, **PutByte**, **PutChar**, **PutWord**, or **SendNow** with a **Stream.Handle** parameter pointing to the next stream component in the pipeline (*i.e.*, the parameter passed at the time this filter was created). In a transducer, the body of a **PutProcedure** will typically have calls on output operations for the specific device being supported.

The **putByte** field specifies the byte output procedure provided by the transducer or filter.

putByte: **Stream.PutByteProcedure;**

Stream.PutByteProcedure = **PROCEDURE [sH: Stream.Handle, byte:Stream.Byte];**

This procedure may generate the signal **TimeOut** if necessary.

The **putWord** field specifies the word output procedure provided by the transducer or filter.

putWord: **Stream.PutWordProcedure;**

Stream.PutWordProcedure = **PROCEDURE [sH: Stream.Handle, word:Stream.Word];**

This procedure may generate the signal **TimeOut** if necessary.

The **setSST** field specifies the procedure to change the current **SubSequenceType** of the output side of the filter or transducer.

setSST: **stream.SetSSTProcedure;**

Stream.SetSSTProcedure: TYPE = **PROCEDURE [sH: Stream.Handle,**
sst: Stream.SubSequenceType];

This procedure should be a no-op if the new **SubSequenceType** of **sH** is the same as the old one. Otherwise, it should have the effect of completing the current physical record (as if a call on **Stream.SendNow** had been made immediately before).

A call on **setSST** may have the effect of changing the internal state of the stream component, or in the case of a filter, it may result in a call to **SetSST** to the next stream component in the pipeline, or both.

The **getSST** field specifies the procedure to find the current **SubSequenceType** of the output side of the filter or transducer (the SST set by **SetSST**). The input SST can be found by doing a get of 0 bytes.

getSST: **stream.GetSSTProcedure;**

Stream.GetSSTProcedure: TYPE = **PROCEDURE [sH: Stream.Handle]**
RETURNS [sst: Stream.SubSequenceType];

The **sendAttention** and **waitAttention** fields specify the two procedures implementing the sending of and waiting for attention flags in the transducer or filter.

sendAttention: **Stream.SendAttentionProcedure;**

```
waitAttention: Stream.WaitAttentionProcedure;  
  
Stream.SendAttentionProcedure: TYPE = PROCEDURE [sH: Stream.Handle,  
        byte: Stream.Byte];  
  
Stream.WaitAttentionProcedure: TYPE = PROCEDURE [sH: Stream.Handle]  
        RETURNS [byte: Stream.Byte];
```

These two procedures will be called by **Stream.SendAttention** and **Stream.WaitForAttention**, respectively.

The **getTimeout** field specifies the procedure to find the current timeout field of the filter or transducer.

```
getTimeout: Stream.GetTimeoutProcedure;  
  
Stream.GetTimeoutProcedure: TYPE = PROCEDURE [sH: Stream.Handle]  
        RETURNS [waitTime: LONG CARDINAL -- msecs--];
```

The **setTimeout** field specifies the procedure to set the current timeout field of the filter or transducer.

```
setTimeout: Stream.SetTimeoutProcedure;  
  
Stream.SetTimeoutProcedure: TYPE = PROCEDURE [sH: Stream.Handle,  
        waitTime: LONG CARDINAL -- msecs--];
```

The **delete** field specifies a procedure implementing the deletion of a filter or transducer.

```
delete: Stream.DeleteProcedure;  
  
Stream.DeleteProcedure: TYPE = PROCEDURE [sH: Stream.Handle];
```

This procedure is called by the **Stream.Delete** operation.

The **getPosition** and **setPosition** fields specify procedures implementing the setting and recovering of a stream position.

```
getPosition: StreamGetPositionProcedure;  
  
StreamGetPositionProcedure: TYPE = PROCEDURE [sH: Stream.Handle]  
        RETURNS [position: Stream.Position];  
  
setPosition: Stream.SetPositionProcedure;  
  
Stream.SetPositionProcedure: TYPE = PROCEDURE [sH: Stream.Handle,  
        position: Stream.Position];
```

The **sendNow** field specifies a procedure to force data to be transmitted.

```
sendNow: Stream.SendNowProcedure;
```

```
Stream.SendNowProcedure: TYPE = PROCEDURE [sH: Stream.Handle,
endRecord: BOOLEAN ← FALSE];
```

This procedure is called by the **Stream.SendNow** operation.

The following object is provided to supply default values for a **Stream.Object**. It is an exported variable. The implementor of a stream can use it to ease the burden of initializing *all* of the fields in a **Stream.Object** although the implementor must still initialize some of the fields.

```
Stream.defaultObject: READONLY Stream.Object = [
  options: Stream.defaultInputOptions,
  getByte: ..., -- requires sH.get to be defined
  putByte: ..., -- requires sH.put to be defined
  getWord: ..., -- requires either sH.getByte or sH.get to be defined
  putWord: ..., -- requires either sH.putByte or sH.put to be defined
  get: ..., -- requires sH.getByte to be defined
  put: ..., -- requires sH.putByte to be defined
  setSST, sendAttention, waitAttention, delete: ...]
```

In this description, the phrase "to be defined" means that the supplied default procedure *assumes* that the user has supplied the indicated procedure as opposed to using the default procedure. Thus, the implementor of the stream must supply either **getByte** or **get** -- both cannot be defaulted. Similarly, the implementor must supply either **putByte** or **put** -- both cannot be defaulted. The default entries for **setSST**, **getSST**, **setTimeout**, **getTimeout**, **sendAttention**, **waitAttention** and **delete** simply raise the exception **Stream.InvalidOperation**. Thus, the implementor must supply these procedures.

Stream.InvalidOperation: ERROR;

Individual default procedures may be extracted for client use by the standard Mesa extractor expression. For example, the default **get** procedure is **defaultObject.get**.

Caution: The effect of not providing at least one of **getByte/get** (**putByte/put**) is unspecified by Pilot. *Thus the stream implementor must be sure to provide at least one of each of these pairs of procedures.*

3.5.2 Stream component managers

Implementors of stream components may create instances of them by whatever means is most appropriate to their requirements. A particular filter or transducer may, for example, consist of one module, a collection of modules, a local frame used in conjunction with the Mesa **PORT** facility, or some other construct. Moreover, it may be allowed to exist on a given machine in only one or a limited number of copies which are regarded as "serially reusable" resources (for example, a transducer to a particular device, of which there is only one or a limited number on a machine), or it may be allowed to exist in as many copies as appropriate (for example, the Network stream of §6.3). It is the responsibility of the stream component manager to create (or control access to) instances of that stream component, as appropriate. When access is granted, the component manager must also provide a pointer to a **Stream.Object** containing procedure descriptors for that component.

One way of implementing a component is as a single module which is instantiated at run-time by the Mesa **NEW** statement. Declared within this module would be the procedures of the component plus a **Stream.Object** which would contain their procedure descriptors. The component manager executes **NEW** to create a new instance of one of these, followed by **START** to initialize it, pass any parameters to it, and get back a pointer to the **Stream.Object**.

The component manager deletes instances of stream components by calling **Runtime.UnNew** or **Runtime.SelfDestruct**.

Runtime.SelfDestruct sets the internal state of the process so that the module in which the calling procedure is declared will be un-**NEW**'d after the calling procedure returns to its own caller. This operation has the effect of placing a "self-destruct" mechanism in the module which will take effect after the calling process exits from it. Thus, it is a means of deleting the stream component from within that component.

The typical use of **Runtime.SelfDestruct** will be from a procedure named in the **delete** entry of the **Stream.Object**. The component manager will call **h.delete[h]** (where **h** is a **Stream.Handle**). This procedure will perform the necessary finalization, such as flushing buffers, closing files or connections, releasing storage and resources, etc. It will then call **Runtime.SelfDestruct** and finally return to the component manager. After this return, the module representing this instance of the stream component will be automatically deleted and space occupied by the component's global frame freed.

Caution: The client must ensure that there are no outstanding references to the component module being deleted -- i.e., no procedure descriptors or pointers which might be used. In addition, any process waiting for attentions (i.e., a process which has called but not returned from **WaitForAttention**) must be aborted and allowed to exit from the module. Failure to observe this caution will result in unpredictable effects. In particular, **Runtime.UnNew** must be called from outside the module being deleted.

(

(

(

File Storage and Memory

A *file* is the basic unit of long-term information storage (see §4.3). A file consists of a sequence of pages, the contents of which can be preserved across system restarts. Files are stored on *volumes* (see §4.1, 4.2) and are identified by the containing volume and a file identifier which is unique within that volume.

Pilot stores files on *logical volumes*, which are contained in *physical volumes* of storage devices (typically disks). A physical volume is the basic unit of physical availability for random access file storage. It represents the notion of a storage medium whose availability is intrinsically independent of that of other instances of such media (e.g., one physical disk pack). A logical volume is either a physical volume or a subset of a physical volume or a collection of subsets of physical volumes. A logical volume is the unit of storage for client files and the system data structures for manipulating them. It becomes logically available or unavailable as a unit and contains only complete files (i.e., files cannot span logical volumes). Volumes which have been damaged may be restored by *scavenging* (see §4.4).

Client programs access data in files by mapping them into *spaces* in virtual memory (see §4.5). Pilot provides client programs with facilities for associating areas of virtual memory with portions of files, for allocating sections of virtual memory independent of mapping, and for influencing swapping between virtual and real memory.

Pilot provides free storage management through *zones* and *heaps* (see §4.6). Zones are segments of storage in client-designated areas of virtual memory. Heaps are available for managing arbitrarily sized nodes; they support the Mesa language facilities for dynamic storage allocation.

A general purpose log file facility (§4.7) allows recording of information in a client-supplied log file.

4.1 Physical volumes

PhysicalVolume: DEFINITIONS ...;

This section describes those interfaces provided by Pilot which permit clients to initialize and manage physical volumes. Pilot brings the system physical volume online during Pilot initialization, repairing it if necessary. Thus most clients will not need to use the

facilities in this section. However, UtilityPilot-based clients do not have a system physical volume; these clients must manage physical volumes themselves. Clients which might use the **PhysicalVolume** facilities include volume management utility programs, system elements with several physical volumes, and UtilityPilot-based systems. Sections 4.1.1 through 4.1.4, 4.1.7, and 4.1.8 deal with general physical volume management, section 4.1.5 with initializing a physical volume, and section 4.1.6 with scavenging. See also Chapter 8 for facilities to format physical volumes and install boot files on them.

4.1.1 Physical volume name and size

The fundamental name for a physical volume is its **ID**.

PhysicalVolume.ID: TYPE = System.PhysicalVolumeID;

System.PhysicalVolumeID: TYPE = RECORD [System.UniversalID];

PhysicalVolume.nullID: PhysicalVolume.ID = [System.nullID]; -- "null ID"

Pilot ensures with a very high probability that each distinct physical volume is assigned a distinct **ID**. No **ID** is reused for any purpose by any copy of Pilot on any machine at any time. Thus, a physical volume may be unambiguously identified by its **ID**, even if it is moved to another machine or environment, or if it is stored off-line for a long time. **nullID** is never assigned as an **ID** and is used to indicate the absence of a physical volume.

The error **PhysicalVolume.Error[physicalVolumeUnknown]** may be raised by any of the operations that take an **ID** as an argument.

A physical volume is organized as a sequence of up to 2^{32} pages, each containing **Environment.wordsPerPage** words. Pages are numbered starting from zero. The actual volume size is accounted for by Pilot and does not result in the redefinition of the maximum page number.

PhysicalVolume.PageCount: TYPE = LONG CARDINAL;

PhysicalVolume.firstPageCount: PhysicalVolume.PageCount = 0;

PhysicalVolume.lastPageCount: PhysicalVolume.PageCount = LAST[LONG CARDINAL];

PhysicalVolume.PageNumber: TYPE = LONG CARDINAL;

PhysicalVolume.firstPageNumber: PhysicalVolume.PageNumber = 0;

PhysicalVolume.lastPageNumber: PhysicalVolume.PageNumber = LAST[LONG CARDINAL] - 1;

Pilot's maximum values for **PageCount** and **PageNumber** do not, for all practical purposes, limit the size of a physical volume.

4.1.2 Physical volume errors

PhysicalVolume operations may raise the following signals:

PhysicalVolume.Error: ERROR [error: PhysicalVolume.ErrorType];

PhysicalVolume.ErrorType: TYPE = {badDisk, badSpotTableFull, containsOpenVolumes, diskReadError, hardwareError, hasPilotVolume, alreadyAsserted, insufficientSpace, invalidHandle, nameRequired, notReady, noSuchDrive, noSuchLogicalVolume, physicalVolumeUnknown, writeProtected, wrongFormat, needsConversion};

PhysicalVolume.NeedsScavenging: ERROR;

The conditions causing each error are described as the error occurs in the text. The errors raised by each operation are indicated with the operation's description.

4.1.3 Drives and disks

A *drive* is an I/O device capable of containing a Pilot physical volume. Such devices have a **Device.Type** which is in the range defined by **Device.PilotDisk**. The storage medium on a drive is the physical object which holds the stored information, typically a fixed disk or a removable disk pack. It will be called a *disk* in the description which follows. A drive is uniquely named by its device index. A drive may be in two *states*: if a drive is *ready* then it contains a storage device, e.g., a disk pack, that may be accessed by Pilot; if the drive is *not ready* then it does not contain an accessible storage device.

PhysicalVolume.ErrorType: TYPE = {..., noSuchDrive, ...};

All operations which take a device index will raise **PhysicalVolume.Error[noSuchDrive]** if provided a device index which does not denote a drive.

The set of drives on a machine may be enumerated with the operation

PhysicalVolume.GetNextDrive: PROCEDURE [index: CARDINAL] RETURNS [nextIndex: CARDINAL];

PhysicalVolume.nullDeviceIndex: CARDINAL = LAST[CARDINAL];

GetNextDrive is a stateless enumerator. Enumeration begins and ends with the value **nullDeviceIndex**. **GetNextDrive** may raise **Error[noSuchDrive]**.

For every drive, Pilot maintains a monotonically increasing *change count* of the number of times that the drive has changed state between ready and not ready. If a drive changes state, the change count for that drive will increase by at least one. Thus while the change count remains the same the client can be sure that the same disk is mounted on the drive.

The client may wait for one or more drives to change state by invoking

PhysicalVolume.AwaitStateChange: PROCEDURE [changeCount: CARDINAL, index: CARDINAL ← PhysicalVolume.nullDeviceIndex] RETURNS [currentChangeCount: CARDINAL];

The **AwaitStateChange** operation waits until the change count of the drive equals or exceeds **changeCount**, then returns the new change count. If **index = nullDeviceIndex**, the operations waits until the sum of the change counts of all drives equals or exceeds **changeCount**, then returns the sum. **AwaitStateChange** may raise **Error[noSuchDrive]**.

A unique instance of a disk mounted on a drive is represented by a **PhysicalVolume.Handle**. A **Handle** denotes both a drive and the change count at the time at which the **Handle** was obtained. A **Handle** is valid until the drive that it denotes changes state. After that time, the error **Error[invalidHandle]** is raised by any operation that takes a **Handle** as an argument.

PhysicalVolume.Handle: TYPE [3];

PhysicalVolume.ErrorType: TYPE = {..., invalidHandle, ...};

PhysicalVolume.GetHandle: PROCEDURE [index: CARDINAL] RETURNS [PhysicalVolume.Handle];

**PhysicalVolume.InterpretHandle: PROCEDURE [instance: PhysicalVolume.Handle]
RETURNS [type: Device.Type, index: CARDINAL];**

A **Handle** is obtained for a drive using **GetHandle**. The change count of the drive at the time **GetHandle** is invoked defines the valid change count for the disk mounted on the drive represented by the returned **Handle**. **GetHandle** may raise **Error[noSuchDrive]**. **InterpretHandle** returns the drive denoted by a given **Handle**. The returned **type** may be general rather than precise, i.e., a type naming a device family rather than a specific member of the family. **InterpretHandle** may raise **Error[invalidHandle]**.

Information about the ready state of a drive can be obtained with

**PhysicalVolume.IsReady: PROCEDURE [instance: PhysicalVolume.Handle]
RETURNS [ready: BOOLEAN];**

IsReady may raise **Error[invalidHandle]**.

4.1.4 Disk access, Pilot volumes, and non-Pilot volumes

The disk on a ready drive may be in one of three states: inactive, Pilot access, and non-Pilot access. An *inactive* disk may be accessed only in stylized ways that permit clients to determine in which of the other two states to place the device. A disk with *Pilot access* contains a Pilot physical volume and may be accessed only through the Pilot File, **PhysicalVolume**, Space and Volume interfaces. *Non-Pilot access* indicates that the the disk may be accessed only through special interfaces which permit *direct access* (that is, unembellished with Pilot space, mapping and file structures) to the storage device. Whenever a drive becomes ready, Pilot places its disk in the inactive state. Once a client has obtained a **Handle** for a drive and ascertained that the disk is ready, the client must inform Pilot what type of access to the disk is desired. The following operations allow clients to determine and change the state of a drive.

To aid the client in determining how to access a disk, Pilot provides two facilities. The first is an operation which examines the disk and determines whether or not it contains a Pilot volume.

**PhysicalVolume.GetHints: PROCEDURE [
instance: PhysicalVolume.Handle, label: LONG STRING ← NIL]
RETURNS [pvID: PhysicalVolume.ID, volumeType: PhysicalVolume.VolumeType];**

PhysicalVolume.VolumeType: TYPE = {notPilot, probablyNotPilot, probablyPilot, isPilot};

The returned **volumeType** gives Pilot's best guess as to the nature of the disk on **instance**. In **volumeType**: **notPilot** indicates that the disk is definitely not a Pilot physical volume; **probablyNotPilot** indicates that the disk may or may not be a Pilot volume but attempting to use the disk as a Pilot physical volume is likely to fail; **probablyPilot** indicates that the disk may not actually contain a Pilot volume, but that an attempt to use it as a Pilot physical volume is very likely to succeed; **isPilot** indicates that the disk almost certainly is a Pilot physical volume. In all four cases, **pvid** is the identifier that the disk appears to have and **label** is the apparent label of the disk. (See **PhysicalVolume.CreatePhysicalVolume** below for more information about physical volume labels.) It does not matter whether or not the access state of the disk has already been asserted. **GetHints** does not change the access state of the disk. **GetHints** may raise **Error[notReady]** or **Error[invalidHandle]**.

As a second facility to aid the client in determining how to access a disk, Pilot permits the client *read-only*, direct access to the device. This allows the client to examine the disk safely to determine if it contains a known, but non-Pilot, volume. Such access is provided by special Pilot interfaces.

Given the result of the **GetHints** operation and of reading the disk, the client can declare the access desired to the disk. Upon return from these operations, that the client has the indicated access to the disk.

PhysicalVolume.AssertPilotVolume: PROCEDURE [instance: PhysicalVolume.Handle]
RETURNS [PhysicalVolume.ID];

PhysicalVolume.ErrorType: TYPE = {..., alreadyAsserted, ...};

AssertPilotVolume asserts to Pilot that the disk contains a Pilot volume. If **instance** is not in the inactive state, **Error[alreadyAsserted]** is raised. If Pilot's data structures are not in order, **NeedsScavenging** is raised (see Section 4.1.6 on scavenging). **Error[notReady]** and **Error[invalidHandle]** may also be raised. On return, the disk is in the Pilot-access state and the physical volume named by the returned value may be accessed. The returned physical volume is said to be *online*.

PhysicalVolume.Offline: PROCEDURE [pvid: PhysicalVolume.ID];

PhysicalVolume.ErrorType: TYPE =
{..., containsOpenVolumes, physicalVolumeUnknown, ...};

Offline terminates access to an online Pilot physical volume, returning the drive containing that volume to the inactive state. All logical volumes contained on the physical volume must be closed. This operation may raise **Error[physicalVolumeUnknown]** or **Error[containsOpenVolumes]**.

Caution: In the current version of Pilot, if a disk goes not ready while in the Pilot access state, the results are unspecified.

Non-Pilot access to a disk is initiated and terminated with the following operations.

```
PhysicalVolume.AssertNotAPilotVolume: PROCEDURE [instance: PhysicalVolume.Handle];  
  
PhysicalVolume.FinishWithNonPilotVolume: PROCEDURE [instance: PhysicalVolume.Handle];  
  
PhysicalVolume.ErrorType: TYPE = {..., hasPilotVolume, ...};
```

AssertNotAPilotVolume initiates direct access to a storage device. If the drive is not currently in the inactive state, **Error[alreadyAsserted]** is raised. **Error[invalidHandle]** may also be raised. On return, unlimited access to the device is permitted by Pilot through special direct access facilities.

FinishWithNonPilotVolume returns a disk being accessed with non-Pilot access to the inactive state. It raises **Error[hasPilotVolume]** if **instance** currently is in Pilot-access mode. It may also raise **Error[invalidHandle]**.

4.1.5 Physical volume creation

Pilot disks are created by first creating a physical volume and then creating logical volumes upon that physical volume. All storage devices require formatting before their first use. (See §8.3.1 for formatting, §4.2.4 for logical volume creation.) A physical volume is created by invoking

```
PhysicalVolume.CreatePhysicalVolume: PROCEDURE [  
    instance: PhysicalVolume.Handle, name: LONG STRING]  
    RETURNS [PhysicalVolume.ID];  
  
PhysicalVolume.maxNameLength: CARDINAL = 40;  
  
PhysicalVolume.ErrorType: TYPE = {..., badDisk, diskReadError, nameRequired, ...};
```

This creates a physical volume upon **instance**. If **instance** is in the Pilot access state, Pilot first calls **Offline** to place it in the inactive state. This may raise **Error[physicalVolumeUnknown]** or **Error[containsOpenVolumes]**. The label of the newly created physical volume is **name**. The name must contain at least one character or **Error[nameRequired]** is raised. If the name contains more than **maxNameLength** characters, only the first **maxNameLength** characters will be used as the label. The newly created volume is placed online (i.e., just as if **AssertPilotVolume** had been called) and its **ID** is returned. If the specified drive is in either the Pilot access state (i.e., online) or in the non-Pilot access state, **Error[alreadyAsserted]** is raised. If Pilot cannot do the necessary disk access required to create a physical volume on the disk, **Error[badDisk]** or **Error[diskReadError]** will be raised. This operation may also raise **Error[notReady]** and **Error[invalidHandle]**.

4.1.6 Scavenging

Scavenging is the process of returning a physical or logical volume to a consistent state. This is necessary if the volume was damaged by software errors, pages on the disk went

bad, the volume is not of the current version, or the like. Section 4.4 covers scavenging logical volumes. A physical volume can be scavenged by invoking

```
PhysicalVolume.Scavenge: PROCEDURE [instance: PhysicalVolume.Handle,
    repair: PhysicalVolume.RepairType, okayToConvert: BOOLEAN]
    RETURNS [status: PhysicalVolume.ScavengerStatus];

PhysicalVolume.RepairType: TYPE = {checkOnly, safeRepair, riskyRepair};

PhysicalVolume.ScavengerStatus: TYPE = RECORD [
    badPageList, bootFile, germ, softMicrocode, hardMicrocode:
        PhysicalVolume.DamageStatus,
    internalStructures: PhysicalVolume.RepairStatus];

PhysicalVolume.DamageStatus: TYPE = {okay, damaged, lost};

PhysicalVolume.RepairStatus: TYPE = {okay, damaged, repaired};

PhysicalVolume.NOProblems: READONLY PhysicalVolume.ScavengerStatus = ...;
```

The purpose of **Scavenge** is two-fold. First, it allows Pilot to place its internal physical volume data structures in order so that client access to the physical volume may be permitted. Second, it returns a **ScavengerStatus** describing any damage found for which the client has repair responsibility. **PhysicalVolume.Scavenge** is responsible for the integrity of the physical volume only. To repair any logical volume damage, the client must call **Scavenger.Scavenge**.

If the volume is not of the current version, i.e., not compatible with the Pilot boot file which is running, it must be made so before any access is allowed. Invoking **Scavenge** with **okayToConvert** = **TRUE** will cause the volume's version to be increased to the current version. This is the only way to cause volume conversion. Scavenging to a previous version is not supported, nor is scavenging a volume forward more than one version.

The physical volume to-be scavenged must be offline. **Error[alreadyAsserted]** is raised if the specified disk drive is online. If the volume version is incorrect and **okayToConvert** is **FALSE**, **Error[needsConversion]** is raised. **Error[badDisk]** is raised if the damage to the physical volume data structures is so great that the physical volume cannot be reconstructed. **Error[invalidHandle]** may also be raised.

If **repair** is set to **safeRepair** or **riskyRepair**, the scavenger will attempt to repair the damage that it finds on the physical volume. The **safeRepair** mode is limited to repairs that are expected to be low risk. The **riskyRepair** mode imposes no such limits and should be used only as a last resort. In particular, it should be used only when the hardware is known to be functioning correctly. If **repair** is set to **checkOnly**, no repair is attempted but a **ScavengerStatus** indicating any damage is returned.

The individual status fields have the following meanings:

badPageList: okay is returned if the bad page list is intact. A status of **damaged** is returned if damage is found and the parameter **repair** was set to **checkOnly**. A status of **lost** indicates that damage was found and **repair** was set to **safeRepair** or **riskyRepair**. If **badPageList** = **lost**, the physical volume scavenger resets the bad page

list to empty and marks all logical volumes on this physical volume to be scavenged. Bad pages must be marked bad again using a disk utility such as Othello.

bootFile, germ, softMicrocode, hardMicrocode: **okay** is returned if the indicated file, and the reference to it in the physical volume's data structures, are intact. If the status returned is **damaged**, the indicated file has been found to be damaged. That is, there are unreadable pages, missing pages, or the file is otherwise not in valid boot file format. The physical volume scavenger will mark the containing logical volume to be scavenged. The client should either delete the boot file and reinstall it, or scavenge that logical volume to discover and repair any unreadable or missing pages before replacing its contents. If the status returned is **lost**, the reference to the indicated file contained in the physical volume's data structures appears to be damaged, either because the data structures have been damaged or the boot file has been deleted. If the file has a unique file type and has not been deleted, the client should be able to find it and restore it via **OthelloOps.SetPhysicalVolumeBootFile** as the appropriate physical volume boot, germ, or microcode file.

internalStructures: the status returned is **okay** if no damage is discovered in the internal data structures of the physical volume. The status returned is **damaged** if damage was found and the parameter **repair** was set to **checkOnly**, or if **repair** was set to **safeRepair** and damage was found that can be repaired only in **riskyRepair** mode. The status is **repaired** if **repair** was set to **riskyRepair**, or if **repair** was set to **safeRepair** and damage was found which could be repaired safely.

The constant **noProblems** is provided to allow the client to determine with a single comparison whether it has any work to do after the physical volume scavenger finishes.

Caution: In Pilot 11.0 the local time parameters may be lost any time the physical volume scavenger repairs internal volume structures. This will be the case when **internalStructures** is not reported as **okay** and **repair** is set to **safeRepair** or **riskyRepair**. It is the client's responsibility to reset local time parameters correctly if they have been lost.

Caution: In Pilot 11.0 the only significant fields of **status** are **badPageList** and **internalStructures**. The other fields are always returned as **okay**, and for them none of the validity checking implied is performed.

4.1.7 Logical volume operations on physical volumes

The logical volumes on an online physical volume may be enumerated by invoking

```
PhysicalVolume.GetNextLogicalVolume: PROCEDURE [  
    pVid: PhysicalVolume.ID, lVid: System.VolumeID]  
    RETURNS [System.VolumeID];
```

This operation is a stateless enumerator. The enumeration begins and ends with **Volume.nullID**. **GetNextLogicalVolume** may raise **Error[physicalVolumeUnknown]** and **Error[noSuchLogicalVolume]**.

The physical volume that contains a given logical volume is returned by

```
PhysicalVolume.GetContainingPhysicalVolume: PROCEDURE [lvID: System.VolumID]
RETURNS [pvID: PhysicalVolume.ID];
```

If lvID is unknown to Pilot, Volume.Unknown is returned. Note that lvID need not be open to invoke this operation. However, it must be in an online physical volume.

4.1.8 Miscellaneous operations on physical volumes

The set of online physical volumes is enumerated by

```
PhysicalVolume.GetNext: PROCEDURE [pvID: PhysicalVolume.ID]
RETURNS [PhysicalVolume.ID];
```

This operation is a stateless enumerator. The enumeration begins and ends with PhysicalVolume.nullID. If pvID is not known to Pilot, Error[physicalVolumeUnknown] is raised.

The attributes of an online physical volume may be ascertained by invoking

```
PhysicalVolume.GetAttributes: PROCEDURE [pvID: PhysicalVolume.ID, label: LONG STRING ← NIL]
RETURNS [instance: PhysicalVolume.Handle, layout: PhysicalVolume.Layout];
```

```
PhysicalVolume.Layout: TYPE =
{partialLogicalVolume, singleLogicalVolume, multipleLogicalVolumes, empty};
```

A handle to the drive containing the physical volume is returned in instance, the label name string is returned in label, and the nature of the logical volumes that exist upon pvID is returned in layout. If the volume label is longer than the string label, only the characters which will fit into the string are returned. A layout value of singleLogicalVolume indicates that there is one entire logical volume on pvID; multipleLogicalVolumes indicates that there is more than one logical volume upon pvID. A value of empty indicates that no logical volumes have been created upon pvID. GetAttributes may raise Error[physicalVolumeUnknown].

The physical volume name (label) may be changed by invoking

```
PhysicalVolume.ChangeName: PROCEDURE [pvID: PhysicalVolume.ID, newName: LONG STRING];
```

If the length of newName exceeds PhysicalVolume.maxNameLength, only the first maxNameLength characters are used. If newName does not contain at least one character, Error[nameRequired] is raised. ChangeName may also raise Error[physicalVolumeUnknown].

A physical volume may have pages upon it that are unusable (e.g., some sector of the disk has failed). Such pages are called *bad pages*. A page is marked as bad by the operation

```
PhysicalVolume.MarkPageBad: PROCEDURE
[pvID:PhysicalVolume.ID, badPage: PhysicalVolume.PageNumber];
```

After a page has been marked bad, Pilot no longer attempts to access it. If a page is to be marked as bad, the logical volume containing that page should be closed before invoking **MarkPageBad**. This is not checked by Pilot. Moreover, after this operation returns, that logical volume should be scavenged before being opened. Pilot will remember only a limited number of bad pages for a given physical volume. If Pilot's table of bad pages is full, **Error[badSpotTableFull]** is raised and **badPage** is not remembered as being bad. See §8.3 for a description of Pilot facilities for identifying bad pages. **MarkPageBad** may also raise **Error[physicalVolumeUnknown]**.

The set of bad pages on a physical volume may be enumerated by invoking

```
PhysicalVolume.GetNextBadPage: PROCEDURE [
    pvid: PhysicalVolume.ID, thisBadPageNumber: PhysicalVolume.PageNumber]
    RETURNS [nextBadPageNumber: PhysicalVolume.PageNumber];

PhysicalVolume.nullBadPage: PageNumber = LAST[PageNumber];
```

This operation is a stateless enumerator. Enumeration begins and ends with **nullBadPage**. **GetNextBadPage** may raise **Error[physicalVolumeUnknown]**.

4.2 Logical volumes

Volume: DEFINITIONS ...;

In this section the term *volume*, where not specified as logical or physical, will refer to a logical volume.

Before being presented to Pilot for the first time, a volume must be initialized, and it may require scavenging or re-initialization after system crashes. Such operations are performed using Othello (see the *Mesa User's Guide*), or by a user-written volume initializer (See Chapter 8).

The current version of Pilot supports a maximum of ten logical volumes on a physical volume.

4.2.1 Volume name and size

The fundamental name for a volume is its **ID**:

```
Volume.ID: TYPE = System.VolumeID;
System.VolumeID: TYPE = RECORD [System.UniversalID];
Volume.nullID: Volume.ID = [System.nullID];
```

Pilot ensures with a very high probability that each distinct volume is assigned a distinct **ID**. No **ID** is reused for any purpose by any copy of Pilot on any machine at any time. Thus a volume may be unambiguously identified by its **ID**, even if it is moved to another machine, or if it is stored offline for a long time. **Volume.nullID** is never the name of a volume and is used to denote the absence of a volume.

The maximum size of a logical volume is 2^{32} bytes, or 223 pages.

Volume.maxPagesPerVolume: LONG CARDINAL = 8388608; -- 2^{23}

Volume.PageCount: TYPE = LONG CARDINAL; -- simulates [0..Volume.maxPagesPerVolume]

Volume.firstPageCount: Volume.PageCount = 0;

Volume.lastPageCount: Volume.PageCount = Volume.maxPagesPerVolume;

Volume.minPagesPerVolume: READONLY Volume.PageCount;

Note: Because LONG subrange types are not implemented in the current version of Mesa, the current version of Pilot defines **Volume.PageCount** as a LONG CARDINAL, and defines constants **firstPageCount** and **lastPageCount** to specify FIRST[PageCount] and LAST[PageCount]. These constants should be used rather than the FIRST and LAST operators, which cannot supply the correct value in the case of a simulated subrange. Minimum and maximum values are similarly defined for **Volume.PageNumber** below.

Volume.PageNumber: TYPE = LONG CARDINAL; -- simulates [0..Volume.maxPagesPerVolume)

Volume.firstPageNumber: volume.PageNumber = 0;

Volume.lastPageNumber: volume.PageNumber = Volume.maxPagesPerVolume - 1;

4.2.2 Logical and physical volumes

The correspondence between logical and physical volumes is not dynamic but is established at volume initialization time. When a logical volume exists on several physical volumes, all of the physical volumes must be available before the logical volume is available. Logical volumes permit the simulation of volume sizes not present in hardware. For example, several smaller disks can be combined to look like a larger disk.

Clients should contemplate combining physical volumes into logical volumes only if file sizes are likely to exceed the size of an individual physical volume. Pilot offers no recovery if one of the physical volumes comprising a logical volume is lost or destroyed. The contents of the remaining physical volumes are, in general, irretrievable.

Note: There is no mechanism to create a logical volume which spans multiple physical volumes.

There is one volume known as the *system volume*, intended to be used as the default volume by Pilot and its clients. The system volume is the logical volume which contains the boot file of the system being executed. The ID of this volume is contained in

Volume.SystemID: READONLY Volume.ID;

Note: In UtilityPilot-based systems there is no system volume. **Volume.systemID** will have the value **Volume.nullID**.

4.2.3 Volume error conditions

The following errors may be raised during many **Volume** operations. The description of each operation indicates which errors it can raise.

Volume.Unknown: **ERROR [volume: Volume.ID];**

Unknown is raised when a volume is not known to Pilot. No part of the volume is online. **Unknown** will be raised if **volume.nullID** is used for any operation except those which start an enumeration.

Volume.NotOnline: **ERROR [volume: Volume.ID];**

NotOnline indicates that a volume is only partially online, i.e., not all of the physical volumes comprising the volume are online.

Volume.NotOpen: **ERROR [volume: Volume.ID];**

Operations which require the volume to be open raise **NotOpen** if the volume is partially online or online but closed.

Volume.ReadOnly: **ERROR [volume: Volume.ID];**

Attempting to change the contents of a volume which is open for reading but not writing, will cause **ReadOnly** to be raised.

Volume.NeedsScavenging: **ERROR [volume: Volume.ID];**

NeedsScavenging indicates that Pilot data structures on the volume are inconsistent or incorrect. This can occur as a result of a system crash, or the volume may have the format of an incompatible version of Pilot, or the volume may not in fact be a Pilot volume.

Volume.InsufficientSpace: **ERROR [**
currentFreeSpace: Volume.PageCount, volume: Volume.ID];

The error **InsufficientSpace** is raised when there is not enough space left in the volume for the requested operation to complete. The number of pages actually available is returned in **currentFreeSpace**.

Volume.Error: **ERROR [error: Volume.ErrorType];**

Volume.ErrorType: **TYPE = {...};**

The specific values for **Error** are defined below as they occur in the text.

4.2.4 Creating and erasing logical volumes

A logical volume can be created on a physical volume by invoking

Volume.Create: **PROCEDURE [**
pvID: System.PhysicalVolumeID, size: Volume.PageCount, name: LONG STRING,

```

type: Volume.Type, minPVPageNumber: PhysicalVolume.PageNumber ← 1]
RETURNS [volume: Volume.ID];

PhysicalVolume.maxSubvolumesOnPhysicalVolume: READONLY CARDINAL;

Volume.maxNameLength: CARDINAL = 40;

Volume.Type: TYPE = MACHINE DEPENDENT
  {normal(0), debugger(1), debuggerDebugger(2), nonPilot(3)};

Volume.ErrorType: TYPE = {nameRequired, pageCountTooSmallForVolume,
  subvolumeHasTooManyBadPages, tooManySubvolumes};

```

This creates a new logical volume on **pVID** of type **type** and containing **size** pages. (See §4.2.6, Opening and closing volumes, for a discussion of the significance of volume types.) The volume label, which can be used to identify the logical volume, is **name**. The label is not used by Pilot. Only the first **Volume.maxNameLength** characters of **name** are used. The newly created volume will not overlap any other logical volumes upon **pVID**. Logical volumes occupy one or more contiguous, disjoint regions of physical volumes. The volume will start at a page number at least as large as page **minPVPageNumber** of **pVID**; it may start later.

If this new volume will cause the number of subvolumes to exceed **maxSubvolumesOnPhysicalVolume**, **Error[tooManySubvolumes]** will be raised. If **pVID** is not a valid physical volume, **PhysicalVolume.Error[physicalVolumeUnknown]** is raised. If **size** is not enough pages to make a volume, **Error[pageCountTooSmallForVolume]** is raised. If there is insufficient unused space on **pVID** to create the logical volume, **PhysicalVolume.Error[insufficientSpace]** will be raised. If **name** is **NIL** or its length is zero, **Error[nameRequired]** is raised. If there are too many bad pages on the area of the disk to be used for the proposed logical volume, **Error[subvolumeHasTooManyBadPages]** to be raised. Hardware errors encountered in creating the volume will cause **PhysicalVolume.Error[hardwareError]** to be raised.

A logical volume may be erased, destroying its previous contents, by invoking

Volume.Erase: PROCEDURE [volume: Volume.ID];

Volume **volume** may be online or open when this operation is invoked, and **Erase** does not affect this status. **Erase** may raise the errors **Unknown**, **NotOnline**, **ReadOnly**, or **PhysicalVolume.Error[hardwareError]**.

4.2.5 Volume status and enumeration

The logical volumes of an online physical volume may be enumerated by **PhysicalVolume.GetNextLogicalVolume** (see §4.1.7).

A client may determine the status of a logical volume by calling

Volume.GetStatus: PROCEDURE [volume: Volume.ID] RETURNS [Volume.Status];

```
volume.Status: TYPE = {unknown, partiallyOnLine, closedAndInconsistent,
                      closedAndConsistent, openRead, openReadWrite};
```

The meaning of each **Status** is as follows. **unknown** indicates that no part of **volume** is contained in an online physical volume. **partiallyOnLine** indicates that the volume spans multiple physical volumes and at least one of those physical volumes is offline. **closedAndInconsistent** means that all parts of **volume** are online but it needs scavenging before it can be opened. **closedAndConsistent** means all parts of **volume** are online, and it is closed and does not need scavenging. **openReadWrite** indicates that **volume** is open and accessible for both reading and writing. **openRead** indicates that the volume is open only for reading.

Clients can discover the identities of online or open logical volumes by calling

```
volume.GetNext: PROCEDURE [volume: Volume.ID,
                           includeWhichVolumes: volume.TypeSet ← onlyEnumerateCurrentType]
                           RETURNS [nextVolume: Volume.ID];
```

```
volume.TypeSet: TYPE = PACKED ARRAY Volume.Type OF Volume.BooleanDefaultFalse;
```

```
volume.BooleanDefaultFalse: TYPE = BOOLEAN ← FALSE;
```

```
volume.OnlyEnumerateCurrentType: Volume.TypeSet = [];
```

GetNext is a stateless enumerator with a starting and ending value of **Volume.randomUUID**. It enumerates the logical volumes of the type(s) specified by **includeWhichVolumes** which are currently online or open. **GetNext** may raise the error **Unknown**.

4.2.6 Opening and closing volumes

When a Pilot boot file is invoked, the *system physical volume* and *system logical volume* are the physical and logical volumes containing the boot file. During its initialization, Pilot brings the system physical volume online and opens the system logical volume, scavenging it if necessary. If the logical volume version is not current (compatible with the Pilot boot file which is running), initialization scavenging will cause it to be converted to the current version.

Note: For UtilityPilot-based systems there is no system physical or logical volume, and no physical or logical volumes are brought online.

A client may open an online volume, making its files accessible, by calling

```
volume.Open: PROCEDURE [volume: Volume.ID];
```

Once a volume is open, the client may create, read, write, and delete files on the volume. Opening an already open volume is a no-op. A volume will be opened read-only if the volume being opened is of a higher **volume.Type** than the system volume. This will be the case if, (a) the system volume is of type **normal**, and **volume** is of type **debugger** or **debuggerDebugger**, or (b) the system volume is of type **debugger** and **volume** is of type **debuggerDebugger**.

Note: For UtilityPilot-based systems volumes are always opened read-write.

An attempt to write on or otherwise change the state of a read-only volume will cause **ReadOnly** to be raised. Other errors which may be raised by **Open** are **Unknown**, **NotOnline**, and **NeedsScavenging**.

Caution: If a debugger opens (for read-write) any volume which its debuggee currently has open, that debuggee should not be allowed to continue execution. Opening the volume changes its state, and the debuggee's Pilot will have out-of-date information about the volume. Continuing its execution in this case will have unpredictable (and undesirable) results.

The client may close an open volume by calling

Volume.Close: PROCEDURE [volume: Volume.ID];

This operation assures that the volume is in a physically consistent state. The data on the volume will no longer be accessible. Closing a closed volume is a no-op. **Close** may raise errors **Unknown** and **NotOnline**.

4.2.7 Volume attributes

Volumes have attributes which can be examined and some of which can be set.

Volume.GetAttributes: PROCEDURE [volume: Volume.ID]
RETURNS [volumeSize, freePageCount: Volume.PageCount, readOnly: BOOLEAN];

This operation may be applied to any online or open volume. The attributes **volumeSize** and **freePageCount** indicate the number of pages and free pages, respectively, of the volume. **freePageCount** is the maximum length file that can be created, or the maximum by which the size of a file may be grown, at that time. Because the space reflected by **freePageCount** must also be used for Pilot internal data structures, it may not be possible to create or extend a file by precisely this much. In general, the amount of free space left after creating or extending a file cannot be predicted exactly. **readOnly** is **TRUE** if the volume is open for reading but not writing, i.e., if it is of a higher **Volume.Type** than the system volume. **GetAttributes** may raise **Unknown**, **NotOnline**, and **NeedsScavenging**.

The ID of the volume that contains the debugger is kept in

Volume.debuggerVolumeID: READONLY Volume.ID;

If it is equal to **Volume.nullID**, there is no debugger present on a local volume. In UtilityPilot-based systems, **debuggerVolumeID** is always **nullID**.

The type of an online or open volume may be ascertained with the procedure

Volume.GetType: PROCEDURE [volume: Volume.ID] RETURNS [type: Volume.Type];

GetType may raise errors **Unknown**, **NotOnline**, and **NeedsScavenging**.

The volume label is set when a volume is created. The label can be used by the client to identify the logical volume, but it is not significant to Pilot. The label of an online or open volume may be changed by the following operation.

Volume.ChangeLabelString: PROCEDURE [volume: Volume.ID, newLabel: LONG STRING];

Only the first **Volume.maxNameLength** characters of **newLabel** are used. If **newLabel** is **NIL** or its length is zero, **Error[nameRequired]** is raised. **ChangeLabelString** may raise **Unknown**, **NotOnline**, **ReadOnly**, and **NeedsScavenging**.

The label of an online or open volume may be retrieved by the following operation.

Volume.GetLabelString: PROCEDURE [volume: Volume.ID, s: LONG STRING];

If the length of the volume label exceeds that of **s**, the returned label will contain only as many characters as will fit. The length will not exceed **maxNameLength**. **GetLabelString** may raise **Unknown**, **NotOnline**, and **NeedsScavenging**.

4.2.8 Volume root directory

The volume root directory provides a mechanism for client file systems to retain a **File.File** for the root of their file system. It provides a mapping from a **File.Type** into a **File.File**. For any given **File.Type** there can be at most one root file. A **File.Type** of **FileTypes.tUntypedFile** functions as a null value for the root directory operations. The operations in this section allow manipulation of an open volume's root directory.

Volume.RootDirectoryError: ERROR [type: Volume.RootDirectoryErrorType];

Volume.RootDirectoryErrorType: TYPE =
{directoryFull, duplicateRootFile, invalidRootFileType, rootFileUnknown};

Root directory operations may raise the error **RootDirectoryError**. Individual errors are described with the operations that raise them. All of the root directory operations may also raise **Unknown**, **NotOnline**, and **NotOpen**, and **NeedsScavenging**.

Inserting a file into the volume root directory is accomplished by

Volume.InsertRootFile: PROCEDURE [type: File.Type, file: File.File];

Volume.maxEntriesInRootDirectory: READONLY CARDINAL;

If the root directory already has an entry for **type**, **RootDirectoryError[duplicateRootFile]** is raised. The root directory is of fixed size. If the insertion would result in more than **maxEntriesInRootDirectory** entries, **RootDirectoryError[directoryFull]** is raised. An attempt to insert a file with type **FileTypes.tUntypedFile** into the root directory results in the error **RootDirectoryError[invalidRootFileType]**. **ReadOnly** may also be raised.

```
Volume.RemoveRootFile: PROCEDURE [
    type: File.Type, volume: Volume.ID ← volume.SystemID];
```

The entry for a given **File.Type** may be removed from the root directory by **RemoveRootFile**. It may raise **RootDirectoryError[rootFileUnknown]** and **ReadOnly**.

```
Volume.LookUpRootFile: PROCEDURE [type: File.Type] RETURNS [file: File.File];
```

The file previously stored for a given file type may be retrieved by calling **LookUpRootFile**. If there is no entry in the root directory for that type, **RootDirectoryError[rootFileUnknown]** is raised.

```
Volume.GetNextRootFile: PROCEDURE [
    lastType: File.Type, volume: Volume.ID ← volume.SystemID]
RETURNS [file: File.File, type: File.Type];
```

The set of root files in the root directory may be enumerated by calling the stateless enumerator **GetNextRootFile**. The enumeration begins and ends with **FileTypes.tUntypedFile**. It may raise **RootDirectoryError[rootFileUnknown]**.

4.3 Files

File: DEFINITIONS . . . ;

FileTypes: DEFINITIONS . . . ;

CommonSoftwareFileTypes: DEFINITIONS . . . ;

A *file* is the basic unit of long-term information storage. A file consists of a sequence of pages, the contents of which can be preserved across system restarts. Files are named by specifying the containing volume, and by a *file identifier* which is unique within that volume. The operations described in this section enable clients to create and destroy files, and to examine and set their attributes.

4.3.1 File naming

A file is named by giving the identifier of the volume on which it resides and the **ID** of the file:

File.ID: TYPE [2];

File.File: TYPE = RECORD [fileID: File.ID, volumeID: System.VolumeID];

File.nullID: File.ID = . . . ; -- "null ID"

File.nullFile: File.File = [File.nullID, [System.nullID]];

File.IDs are unique within any single volume. Since Pilot ensures with a very high probability that each distinct volume is assigned a distinct volume identifier, the combination of a volume identifier and a **File.ID** in a **File.File** is similarly unique. Pilot will normally create files with **File.IDs** which have never appeared on the containing volume. However, Pilot may reuse the **File.IDs** of deleted files under some circumstances. **File.nullID**

is never allocated as the **ID** of a file, and will cause the error **File.Unknown** to be raised if used for any operation except those that start an enumeration. **File.nullFile** may be used to denote the absence of a file.

All **File** operations require the volume containing the file to be open.

4.3.2 Addressing within files

Pilot files may hold up to 2³² bytes (2²³ pages) and may be randomly accessed on a page-by-page basis. All addresses within a file are in terms of page numbers, representing offsets (in pages) from the beginning of the file. The first page of a file is page number zero.

File.PageNumber: TYPE = LONG CARDINAL; -- simulates [0..File.maxPagesPerFile]

File.maxPagesPerFile: LONG CARDINAL = 8388607; -- 2²³-1

File.firstPageNumber: File.PageNumber = 0;

File.lastPageNumber: File.PageNumber = File.maxPagesPerFile - 1;

Note: Because **LONG** subrange types are not implemented in the current version of Mesa, the current version of Pilot defines **PageNumber** as a **LONG CARDINAL** and defines constants **firstPageNumber** and **lastPageNumber** to specify **FIRST[PageNumber]** and **LAST[PageNumber]**. These constants should be used rather than the **FIRST** and **LAST** operators, which cannot supply the correct value in the case of a simulated subrange. Minimum and maximum values are similarly defined below for **File.PageCount**.

File.PageCount: TYPE = LONG CARDINAL; -- simulates [0..File.maxPagesPerFile]

File.firstPageCount: File.PageCount = 0;

File.lastPageCount: File.PageCount = File.maxPagesPerFile;

4.3.3 File types

In Pilot, every file must be assigned a *file type* at the time it is created. A file type is of type **File.Type** and is constant for the life of the file. It provides a means for Pilot, various scavenging programs, and clients to recognize the purpose for which each file was intended. This is especially important because files on Pilot disks do not inherently have meaningful strings for names, making it difficult for a human user or programmer to recognize which file is which. To make this principle work effectively, each different kind of file should be assigned its own unique type. See Appendix B for an explanation of how file types are assigned and managed.

File types are intended to be used by Pilot clients in distinguishing the types of objects represented by Pilot files. Each specific application may assign its own type to its own files, either for redundancy or for control of the processing of those files.

File types are allocated by the Manager of System Development and are defined as follows:

File.Type: TYPE = RECORD [CARDINAL];

The center of this scheme is the **FileTypes** interface, maintained by the Pilot group. In this file are defined all subranges of **File.Type** assigned to individual client and application groups. This module is designed so that it can be recompiled whenever a new type is assigned without invalidating any old version. Thus, within certain limits, a program may include any version of **FileTypes** which contains the file types of interest to it without building in an unnecessary or awkward compilation dependency.

The basic structure of **FileTypes** is a set of subrange and constant definitions. The following ranges are defined. (The reader should consult the documentation of the appropriate system to see how the specific file types have been defined):

FileTypes.MesaFileType: TYPE = CARDINAL[...];

FileTypes.DCSFileType: TYPE = CARDINAL[...];

FileTypes.TestFileType: TYPE = CARDINAL [...];

FileTypes.SBSOFileType: TYPE = CARDINAL [...];

FileTypes.CommonSoftwareFileType: TYPE = CARDINAL [...];

FileTypes.DocProcFileType: TYPE = CARDINAL [...];

FileTypes.FileServiceFileType: TYPE = CARDINAL [...];

FileTypes.ServicesFileType: TYPE = CARDINAL [...];

FileTypes.MesaDEFFileType: TYPE = CARDINAL [...];

FileTypes.PerformanceToolFileType: TYPE = CARDINAL [...];

FileTypes.DiagnosticsFileType: TYPE = CARDINAL [...];

FileTypes.CADFileType: TYPE = CARDINAL [...];

FileTypes.CedarFileType: TYPE = CARDINAL [...];

FileTypes.VersatecFileType: TYPE = CARDINAL [...];

Mesa file types are used by Mesa source and object files. DCS file types are used by development common software. Test file types are used by the test tools. SBSO file types are used by OPD Small Business Systems Operation. Common Software file types are used by product common software. File service file types are used by the file server. Printing service file types are used by the print server. MesaDE file types are used in the Mesa development environment. Performance tool file types are used to store binary data typically generated by performance tools. Diagnostics file types are used by diagnostics software. CAD file types are used by computer aided design software. Cedar file types are used by the PARC Cedar project. Versatec file types are provided for the use of Versatec.

The type

```
FileTypes.tUntypedFile: File.Type = [LAST[CARDINAL]];
```

may be used as a null value, denoting the absence of a type. This is not enforced by Pilot however.

The following common software file types are defined in the range **CommonSoftwareFileType**:

```
CommonSoftwareFileTypes.tUnassigned: File.Type = [...];
```

```
CommonSoftwareFileTypes.tDirectory: File.Type = [...];
```

```
CommonSoftwareFileTypes.tBackstopLog: File.Type = [...];
```

```
CommonSoftwareFileTypes.tCarryVolumeDirectory: File.Type = [...];
```

```
CommonSoftwareFileTypes.tClearingHouseBackupFile: File.Type = [...];
```

```
CommonSoftwareFileTypes.t fileList: File.Type = [...];
```

```
CommonSoftwareFileTypes.tBackstopDebugger: File.Type = [...];
```

```
CommonSoftwareFileTypes.tBackstopDebuggee: File.Type = [...];
```

These are mostly self-explanatory. **tDirectory** is obsolete. **tfileList** is the file type of the file list used by the Floppy file system (see §5.5).

4.3.4 File error conditions

The following errors may arise during file operations:

```
File.Error: ERROR [type: File.ErrorType];
```

```
File.ErrorType: TYPE = {invalidParameters, reservedType};
```

Most file operations raise **Error**. **Error[invalidParameters]** is raised by operations when the parameters specify an illegal condition. **Error[reservedType]** is raised when one of Pilot's reserved file types is used improperly.

```
File.Unknown: ERROR [file: File.File];
```

Unknown indicates that the file does not exist on the given volume. It is also raised if **File.nullFile** is supplied to any operation except a stateless enumerator.

```
File.MissingPages: ERROR [  
    file: File.File, firstMissing: File.PageCount, countMissing: File.PageCount] ;
```

MissingPages indicates that the specified pages are missing from the file due to an exceptional condition, usually a disk hardware error. This error is not raised by any **File** operation, but is raised by other Pilot operations.

File operations may raise the errors **Volume.Unknown**, **Volume.NotOnline**, **Volume.NotOpen**, **Volume.InsufficientSpace**, and **Volume.ReadOnly**.

4.3.5 File creation and deletion

To create a new file on a volume, call the procedure:

```
File.Create: PROCEDURE[
    volume: System.VolumeID, initialSize: File.PageCount, type: File.Type]
    RETURNS [file: File.File]
```

A **File.File** for the new file is returned. Files are created as temporary files. The file initially contains the number of pages specified by **initialSize** (filled with zeros). Pilot attempts to allocate contiguous space on the volume, if such is available. There are significant performance penalties associated with increasing the size of a file. Programmers should make every attempt to create the file with the size it will eventually be. If **initialSize** is zero or greater than **File.maxPagesPerFile**, **Error[invalidParameters]** is raised. If there is not enough space on the volume to contain the file, **Volume.InsufficientSpace** is raised. **Volume.ReadOnly** is raised if the volume is open for reading only.

The **type** attribute of the file is a tag provided by Pilot for the use of higher level software. If **type** is one of a set of values reserved by Pilot, **Error[reservedType]** is raised.

By creating a file on an empty volume, creating a second file, and so on, a client program can construct a set of files all of whose space is guaranteed to be contiguous.

A file is deleted by the operation

```
File.Delete: PROCEDURE [file: File.File];
```

The file is deleted permanently; no "undelete" operation exists. **File.Unknown** is raised if there is no such file on the volume. **Volume.ReadOnly** is raised if the volume is open for reading only.

Caution: The file being deleted must not contain any file windows for mapped spaces (see §4.6.2); the behavior of Pilot in such circumstances is undefined.

4.3.6 File attributes

Aside from its name and contents, a file has three other attributes: size, type, and temporary/permanent status. These can be examined using the operations defined below. All of these operations may raise **File.Unknown**.

The size of a file may be ascertained by calling

```
File.GetSize: PROCEDURE [file: File.File] RETURNS [size: File.PageCount];
```

The size of a file may be altered by calling

```
File.SetSize: PROCEDURE [file: File.File, size: File.PageCount];
```

If the size is increased, Pilot attempts to allocate disk space physically adjacent to the end of the file, and it also attempts to allocate a contiguous sequence of pages, if such is available. Any new pages of the file are filled with zeros. Attempting to set the size to zero or greater than `File.maxPagesPerFile` will cause `Error[invalidParameters]` to be raised. `Volume.ReadOnly` will result if the volume is readonly, and `Volume.InsufficientSpace` will be raised if there are not enough free pages on the volume for the new file size.

Extending a file is a fairly expensive operation. It is better for a client to determine the ultimate amount by which a file is to be extended, and do it all at once rather than to increase its size a page or two at a time. This both reduces the amount of disk traffic and increases the likelihood that Pilot will be able to allocate a contiguous sequence of pages for the extension. There are also continuing performance penalties for accessing a fragmented file, which may result from growing the file one or more times.

Caution: For a file which is being shrunk, the pages being deleted must not be mapped into virtual memory. The behavior of Pilot in such circumstances is undefined.

The rest of the attributes of a file can be inspected collectively by calling

```
File.GetAttributes: PROCEDURE [file: File.File]
    RETURNS [type: File.Type, temporary: BOOLEAN];
```

The `temporary` attribute indicates whether the file is temporary or permanent. Pilot deletes temporary files when the volume is next booted, scavenged, or opened for writing. Permanent files are preserved across system restarts. A file is always created as temporary. A file may be made permanent by calling the operation

```
File.MakePermanent: PROCEDURE [file: File.File];
```

A file should not be made permanent before the client has safely stored the `File.File` for that file in some client-level directory or other permanent data structure. The scavenger (§ 4.4) provides means for recovering a permanent file for which the `File.File` has been lost.

The intended sequence for making a permanent file is as follows: When a client creates a file, it is temporary. The client then stores the `File.File` for that file in a safe place, doing `Space.ForceOut` on the safe place to guarantee that it is written into the backing file. The client then makes the file permanent using `File.MakePermanent`.

4.4 Scavenging

```
Scavenger: DEFINITIONS ...;
```

The act of repairing an inconsistent or damaged Pilot logical volume is known as scavenging. A Pilot logical volume may become damaged for any number of reasons. A machine that is using the volume may stop abnormally due to hardware or software failure. The drive containing the volume may fail and damage the volume, or the physical medium containing (part of) the volume might fail. A damaged volume may not be accessed until it has been repaired. This is enforced at the time that `Volume.Open` is called. If the volume is detected as damaged by Pilot, `Volume.NeedsScavenging` is raised. A volume is repaired using the `Scavenger` interface.

4.4.1 Scavenging a volume

A damaged volume is repaired by the operation

```
Scavenger.Scavenge: PROCEDURE [volume, logDestination: Volume.ID,  
                                repair: Scavenger.RepairType, okayToConvert: BOOLEAN]  
                                RETURNS [logFile: File.File];
```

```
Scavenger.RepairType: TYPE = MACHINE DEPENDENT {checkOnly(0),  
                                              safeRepair(1), riskyRepair(2)};
```

```
Scavenger.Error: ERROR [error: Scavenger.ErrorType];
```

```
Scavenger.ErrorType: TYPE = {..., volumeOpen, cannotWriteLog,  
                            needsRiskyRepair, needsConversion, ...};
```

The purpose of the **Scavenge** operation is two-fold. First, it allows Pilot to place its own data structures in order so that client access to the volume may be permitted. Second, it produces a log file (described below) describing the state of the volume. The log file is intended to be used by client-level scavengers to reconstruct client data structures.

The volume to be scavenged is given by **volume**. If **volume** is open, the error **Error[volumeOpen]** is raised. The log file is created on the volume **logDestination**. If **logDestination** equals **volume**, the created log file is **permanent**; otherwise, the log file is **temporary**. Volume **logDestination** must be open if it is not the same as the volume to be scavenged. **Scavenge** may also raise **Volume.NotOnline** and **Volume.Unknown**.

The level of repair attempted by the scavenger is governed by the value of **repair**. A value of **checkOnly** causes a log file to be produced but no repair is done. In this case, it is advisable to specify **logDestination** to be a volume different from the scavenger since it may not be possible to build a log file on a damaged volume. If **repair** is **safeRepair**, the scavenger will attempt to repair the damage that it finds upon the volume. This is the normal usage. If Pilot is unable to repair the volume satisfactorily in this mode, **Error[needsRiskyRepair]** is returned. Certain forms of repair are performed only if **repair** is equal to **riskyRepair**. Scavenging in **riskyRepair** mode should be attempted only after the hardware has been verified to be working correctly.

Caution: In the current version of Pilot, **repair** equal to **checkOnly** is not implemented.

okayToConvert determines whether conversion of a volume of an incompatible version will occur. A volume is of an incompatible version if its format is not compatible with the Pilot boot file which is running. If **okayToConvert** is **TRUE** scavenging will convert a volume from the previous version to the current one. If the volume version is incompatible but **okayToConvert** is **FALSE**, **Error[needsConversion]** is raised. Scavenging to a previous version is not supported, nor is scavenging a volume forward more than one version. **okayToConvert** is set to **FALSE** during pilot initialization, causing the system logical volume to not be converted forward.

If a previous log file for this volume exists, Pilot attempts to delete it after Pilot data structures have been repaired, but before a new log is written. This delete is comparable

to a call on the **DeleteLog** operation (see below). If Pilot is unable to write the log for any reason, **Error[cannotWriteLog]** is returned and no scavenging is done.

Caution: In the current version of Pilot, the volume is repaired even if **cannotWriteLog** is raised.

During Pilot initialization, the system logical volume is scavenged as necessary with **repair = safeRepair** and **okayToConvert = TRUE**. The resulting log file is placed on the system volume.

4.4.2 Scavenger log file

A log file describes the state of a volume after the **Scavenge** operation has been invoked. It contains information about the volume and the outcome of the **Scavenge** as well as a list of all files on the volume and the problems, if any, with each file. A log file contains a data structure of type **LogFormat**.

```

Scavenger.LogFormat: TYPE = MACHINE DEPENDENT RECORD [
    header: Scavenger.Header,
    files: ARRAY [0..0] OF FileEntry];

Scavenger.Header: TYPE = MACHINE DEPENDENT RECORD [
    seal: CARDINAL ← Scavenger.LogSeal,
    version: CARDINAL ← Scavenger.currentLogVersion,
    volume: Volume.ID,
    date: System.GreenwichMeanTime,
    repairMode: Scavenger.RepairType,
    incomplete: BOOLEAN,
    repaired: BOOLEAN,
    bootFilesDeleted: Scavenger.BootFileArray,
    pad: [0..0] ← 0,
    numberOffiles: LONG CARDINAL];

Scavenger.LogSeal: CARDINAL = 130725B;

Scavenger.currentLogVersion: CARDINAL = 1;

Scavenger.BootFileArray: TYPE =
    PACKED ARRAY Scavenger.BootFileType OF BOOLEAN;

Scavenger.BootFileType: TYPE = MACHINE DEPENDENT {
    hardMicrocode(0), softMicrocode(1), germ(2), pilot(3), debugger(4), debuggee(5)};

Scavenger.noneDeleted: Scavenger.BootFileArray = ALL[FALSE];

Scavenger.FileEntry: TYPE = MACHINE DEPENDENT RECORD [
    file: File.ID,
    sortKey: LONG CARDINAL,
    numberofproblems: CARDINAL,
    problems: ARRAY [0..0] OF Scavenger.Problem];

```

```
Scavenger.Problem: TYPE = MACHINE DEPENDENT RECORD [
    trouble: SELECT entryType:Scavenger.EntryType FROM
        unreadable, missing => [first: File.PageNumber, count: File.PageCount],
        duplicate, orphan => [id: Scavenger.OrphanHandle]
    ENDCASE];

Scavenger.EntryType: TYPE = MACHINE DEPENDENT {
    unreadable(0), missing(1), duplicate(2), orphan(3)};

Scavenger.OrphanHandle: TYPE [2];

Scavenger.tScavengerLog: READONLY File.Type;

Scavenger.tScavengerLogOtherVolume: READONLY File.Type;
```

The log consists of a **Header** followed by zero or more **FileEntries**. The **Header** describes the scavenged volume and the outcome of scavenging. The **seal** field is used to verify that a file is in fact a scavenger log; its value should be **LogSeal**. The **version** is the log file format version; its value should be **currentLogVersion**. The scavenge occurred on volume **volume** at time **date** with the value of the **repair** argument which was passed to the **Scavenge** operation equal to **repairMode**. If **incomplete** is **TRUE**, the file list may not include all files or problems due to insufficient space on the log destination volume or overflow of the internal tables used when scavenging. The header is always complete. A value of **TRUE** for **repaired** indicates that all volume structures are in order and the volume may be accessed. If it was necessary to delete one or more boot files in order to complete the scavenge, the elements of **bootFilesDeleted** corresponding to the deleted boot files will be **TRUE**. Boot files are deleted only in very unusual situations. The count of files on the scavenged volume is given by **numberOfFiles**.

Following the header are **Header.numberOfFiles** contiguous entries of type **FileEntry**. In each entry, **file** identifies the file, **sortKey** is a sort accelerator for client scavengers, and **numberOfProblems** is the number of problems associated with the file. If **numberOfProblems** is not zero, **problems** contains one **Problem** entry for each problem encountered. Note that some files will be absent from the list if **header.incomplete** is **TRUE**.

There are four categories of problem: **unreadable** pages, **missing** pages, **duplicate** pages, and **orphan** pages. If the data portion of a sequence of file pages is unreadable or the label can be read correctly, but is either self-inconsistent or is inconsistent with the rest of the file, an **unreadable Problem** entry is entered in the log. If a sequence of file pages is missing, a **missing Problem** entry is created. If a page has an unreadable label, it cannot be associated with any file and is reported as an **orphan Problem** of a **FileEntry** which has **file** equal to **File.nullID**. Finally, if there are two or more pages claiming to be the same page of a file, one is arbitrarily chosen as the actual file page. The rest are reported as **duplicate Problem** entries. A page identified as **orphan** or **duplicate** is provided a **Scavenger.OrphanHandle** in the problem entry so that the page may be accessed. The size of a **Problem** entry in the log is always **SIZE[Problem]**.

The scavenger cannot detect the absence of one or more pages from the very end of a file. It is the client's responsibility to deal with failures of this nature. If only the first page of a file is missing, Pilot assumes that the file is **permanent**. Missing or unreadable pages

should be accessed only via operations provided by the **Scavenger** interface for dealing with such pages and not by, e.g., **Space.Map**

A scavenger log file built upon the volume being scavenged will be of file type **tScavengerLog**. A log file written to a different volume will have type **tScavengerLogOtherVolume**.

A log file may also be generated by the following operation:

```
Scavenger.MakeFileList: PROCEDURE [volume, logDestination: Volume.ID]
RETURNS [logFile: File.File];
```

This procedure will generate a **Log** for the volume **volume** without the overhead of actually scavenging the volume. If either of the specified volumes is not open, **Volume.NotOpen** is raised. **Volume.Unknown** is raised if either volume is unknown. The resulting log will be the same form as a log generated by **Scavenge** except that no problems are reported. The log file is not an "official" log file, i.e., it is not affected by **Scavenge**, **GetLog**, or **DeleteLog**. The returned file is a temporary file; it is the client's responsibility to make it permanent if that is appropriate.

Caution: The client should not create or delete files from **volume** while **MakeFileList** is in process or the log may be incomplete or incorrect.

4.4.3 Operations on log files

The current log file for an open volume, as produced by the most recent invocation of **Scavenger.Scavenge**[**volume**, . . .], is returned by

```
Scavenger.GetLog: PROCEDURE [volume: Volume.ID]
RETURNS [logFile: File.File];
```

If there is no log file, **File.nullFile** is returned. Even if the returned **logFile** is not **File.nullFile** the log file will not exist if it has been deleted by some means other than a **Scavenger.DeleteLog**. Thus, the client must be prepared to catch the signal **File.Unknown** while accessing **logFile**. **GetLog** may also raise **Volume.NotOpen**, **Volume.NotOnline**, or **Volume.Unknown**.

The current log file for an open volume may be deleted by

```
Scavenger.DeleteLog: PROCEDURE [volume: Volume.ID];
```

volume is the volume which was scavenged to produce the log file. The log file may be on **volume** or it may be on another volume, depending on the log destination chosen for the **Scavenge**. If the volume containing the log file is not open for writing, the file is not deleted. Subsequent **GetLog** operations on **volume** return **File.nullFile** until **Scavenge**[**volume**, . . .] is called again. **DeleteLog** does not affect log files generated by **MakeFileList**. **DeleteLog** may also raise **Volume.NotOpen**, **Volume.NotOnline**, **Volume.Unknown**, or **Volume.ReadOnly**.

4.4.4 Investigating and repairing damaged pages

The damage reported in the log file may be investigated and repaired through the use of the following operations. All of these operations require the volume to be open. All of the operations raise **File.Unknown** if the specified file cannot be found, and **Volume.NotOpen**, **Volume.NotOnline**, or **Volume.Unknown** for the specified problem with the volume. Those which change volume contents may raise **Volume.ReadOnly**.

An unreadable page, as described by an **unreadable Problem** entry, may be read by

```
Scavenger.ReadBadPage: PROCEDURE [
    file: File.File, page: File.PageNumber, destination: Space.PageNumber]
RETURNS [readErrors: BOOLEAN];

Scavenger.ErrorType: TYPE = {..., diskHardwareError, diskNotReady,
    noSuchPage, ...};
```

The contents of page **page** of **file** are read into virtual memory page **destination** which must be mapped and writable. (An address fault or write protect fault is indicated if it is not.) The effect is to overwrite the previous contents of **destination** with the contents of the specified file page. The returned value **readErrors** indicates whether or not any error was encountered while accessing the specified file page. Read errors that occur while reading **page** affect only the value of **readErrors** and are otherwise ignored. If the read encountered errors, the data is *not* guaranteed to be reliable. If **page** does not exist or lies beyond the end of **file**, **Error[noSuchPage]** is raised. If the target disk is not ready, **Error[diskNotReady]** is raised. If the target disk reports a drive-level failure (as opposed to a page-level failure such as a read error), **Error[diskHardwareError]** is raised.

An **unreadable** page may be rewritten or a **missing** page may be replaced by

```
Scavenger.RewritePage: PROCEDURE [
    file: File.File, page: File.PageNumber, source: Space.PageNumber]
RETURNS [writeErrors: BOOLEAN];
```

The current contents of page **page** of **file** are overwritten by virtual memory page **source**, which must be mapped. The original disk page is reused if it is present (to *replace* a file page, use **ReplaceBadPage** below); if the original page is missing, Pilot will allocate a new page for that file page. The return value **writeErrors** indicates whether or not errors were encountered while trying to rewrite the specified page. If **writeErrors** returns **FALSE**, the page should be considered to be rehabilitated. Clients should first attempt to rewrite bad file pages using **RewritePage**. If this fails repeatedly, the client should use **ReplaceBadPage** to rewrite the file page in a different backing page.

If **page** is beyond the end of **file**, **Error[noSuchPage]** is raised. If no page can be allocated to replace a **missing** page, **Volume.InsufficientSpace** is raised. If the target disk is not ready, **Error[diskNotReady]** is raised. If the target disk reports a drive-level failure, **Error[diskHardwareError]** is raised. An address fault will result if **source** is not mapped.

The following procedure also rewrites a bad page in a file, but in addition it discards the disk page that the file currently occupies and allocates a new one:

```
Scavenger.ReplaceBadPage: PROCEDURE [
    file: File.File, page: File.PageNumber, source: Space.PageNumber]
RETURNS [writeErrors: BOOLEAN];
```

ReplaceBadPage will allocate a new page for the specified file page and mark the old page as bad in the physical volume's bad page list. The returned value **writeErrors** indicates whether or not errors were encountered while replacing the file page. This operation always allocates a single new page even if **writeErrors** is returned as **TRUE**. **ReplaceBadPage** is subject to the same error conditions as **RewritePage**.

An orphan page may be read by the operation

```
Scavenger.ReadOrphanPage: PROCEDURE [
    volume: Volume.ID, id: Scavenger.OrphanHandle, destination: Space.PageNumber]
RETURNS [file: File.File, type: File.Type, pageNumber: File.PageNumber,
readErrors: BOOLEAN];
```

```
Scavenger.ErrorType: TYPE = {..., orphanNotFound, ...};
```

The contents of virtual memory page **destination** are overwritten by the contents of the orphan page designated by **id**. The **destination** page must be mapped and writable or an address fault or write protect fault will occur. This operation returns the information that Pilot knows about **id**. The file to which it appears to belong is given by **file**, the apparent page number within that file by **pageNumber**, and the type of file by **type**. If errors were encountered in reading the orphan page, **readErrors** is returned **TRUE** and the returned data is not guaranteed to be accurate.

Caution: There is no validity checking to ensure that the page referred to by **id** is actually an orphan. It is the client's responsibility to pass only a currently valid **OrphanHandle**.

If **id** does not refer to a valid page on **volume**, **Error[orphanNotFound]** is returned. If the target disk is not ready, **Error[diskNotReady]** is raised. If the target disk reports a drive-level hardware failure, **Error[diskHardwareError]** is raised.

Once the client is through with an orphan page, it should be deleted by the operation

```
Scavenger.DeleteOrphanPage: PROCEDURE [volume: Volume.ID, id: Scavenger.OrphanHandle];
```

The specified orphan page is deleted, making invalid all outstanding references to it. If the page is usable, it will be returned to **volume**'s free page pool. If the page is incorrigible, it will be added to the bad page list for the physical volume containing **volume**. If **id** does not refer to a valid page on **volume**, **Error[orphanNotFound]** is raised.

Caution: There is no validity checking to ensure that the page referred to by **id** is actually an orphan. It is the client's responsibility to pass only currently valid **OrphanHandles**. In particular, it is possible for a client to delete a random page from a random file by supplying a random, but valid, value for **id**.

4.5 Virtual memory management

Space: DEFINITIONS . . .

SpaceUsage: DEFINITIONS . . .

The Mesa Processor provides a large, linearly addressed, word-organized virtual memory common to all **PROCESSES** and devices. All software, including Pilot, common software, and applications, resides in this single, uniformly-addressable resource. Pilot both manages and implements it using the system element's physical resources. In particular, client programs can associate areas of virtual memory with portions of files and manage system performance and reliability by controlling swapping between virtual and real memory.

4.5.1 Fundamental concepts of virtual memory

The Mesa Processor virtual memory is organized as a sequence of 2^{24} pages, each containing **Environment.wordsPerPage** words. Pages are numbered starting from zero. Clients can use one fewer page than provided by the Mesa Processor because the last page is reserved for system use. A specific implementation of the processor may provide a smaller virtual address space, which does not require redefining the maximum page number but is accounted for in Pilot's internal data structures. A client program can determine the size of its virtual address space, as described in §4.5.6.1 below.

Environment.wordsPerPage: CARDINAL = 256;

Environment.PageNumber: TYPE = LONG CARDINAL; --[0.. 2^{24} -1)

Environment.firstPageNumber: Environment.PageNumber = 0;

Environment.lastPageNumber: Environment.PageNumber = 16777214; -- 2^{24} -2

Note: Because **LONG** subrange types are not implemented in the current version of Mesa, the current version of Pilot defines **PageNumber** as a **LONG CARDINAL** and defines the constants **firstPageNumber** and **lastPageNumber** to specify **FIRST[PageNumber]** and **LAST[PageNumber]**. Similarly for **PageCount** and **PageOffset** below.

Environment.PageCount: TYPE = LONG CARDINAL; --[0.. 2^{24} -1]

Environment.firstPageCount: Environment.PageCount = 0;

Environment.lastPageCount: Environment.PageCount = lastPageNumber + 1; -- 2^{24} -1

Environment.PageOffset: TYPE = Environment.PageNumber;

Environment.firstPageOffset: Environment.PageOffset = 0;

Environment.lastPageOffset: Environment.PageOffset = lastPageNumber;

The following operation returns a **LONG POINTER** to the first word of a page.

**Environment.LongPointerFromPage: PROCEDURE [page: Environment.PageNumber]
RETURNS [LONG POINTER] = INLINE . . . ;**

The following operation returns the number of the page containing **pointer**. If **pointer** is **NIL**, the value returned is undefined—no signal is raised.

```
Environment.PageFromLongPointer: PROCEDURE [pointer: LONG POINTER]
  RETURNS [Environment.PageNumber] = INLINE . . .;
```

For convenience, copies of the types **wordsPerPage**, **PageNumber**, **PageCount**, and **PageOffset**, and the procedures **LongPointerFromPage** and **PageFromLongPointer** are available in the **Space** interface.

```
Space.WORDsPerPage: CARDINAL = Environment.WORDsPerPage;
```

```
Space.PageNumber: TYPE = Environment.PageNumber;
```

```
Space.PageCount: TYPE = Environment.PageCount;
```

```
Space.PageOffset: TYPE = Environment.PageOffset;
```

```
Space.LongPointerFromPage: PROCEDURE [page: Environment.PageNumber]
  RETURNS [LONG POINTER] = INLINE . . .;
```

```
Space.PageFromLongPointer: PROCEDURE [pointer: LONG POINTER]
  RETURNS [Environment.PageNumber] = INLINE . . .;
```

A basic concept used to describe parts of virtual memory is the **Interval**.

```
Space.Interval: TYPE = RECORD [pointer: LONG POINTER, count: Environment.PageCount];
```

```
Space.nullInterval: Space.Interval = [pointer: NIL, count: 0];
```

An **Interval** is a sequence of pages in the virtual address space, and is described by a pointer to the first page and a count of the number of pages. When Pilot returns an **Interval** to the client, the pointer points to the first word of the first page of the **Interval**. When **Intervals** are passed to Pilot, the pointer may point to any word in the first page. Clients should be careful not to misconstrue the pointer passed to Pilot as defining the first address affected by an operation; **Space** operations always start at page boundaries. **nullInterval** may be used to denote the absence of an interval. It is returned by a few **Space** operations.

Pilot implements virtual memory using the resources of real memory and files. In particular, any part of virtual memory which contains information must be associated with *backing storage* consisting of a sequence of pages from some file. This sequence of file pages is called a *window*. The act of associating an area of virtual memory with a window is known as *mapping*; the resulting interval is called a *map unit*. Any attempt by a program to reference or store into a virtual memory location which is not contained in a mapped interval causes an *address fault*. Any attempt by a program to store into a virtual memory location which has read-only access causes a *write protect fault*. Both faults cause the debugger to be called with an appropriate message.

When an interval is mapped, it is typically subdivided into modest-sized *swap units* to allow more efficient management of swapping. When a **PROCESS** references a page not present in real memory, Pilot reads in the page and any adjacent swapped-out pages of the containing swap unit. Thus the size of a swap unit limits how many pages will be swapped

in when one of its pages is referenced. When inactive pages are moved from real memory to backing storage, Pilot ignores swap unit boundaries. That is, it will swap out a run of consecutive inactive pages even if the run crosses one or more swap unit boundaries. As described below, some attributes of mapped intervals are maintained as properties of the individual swap units.

Note: In unusual circumstances (described below), Pilot may break a client-specified swap unit into smaller swap units.

When an interval is mapped, its swap units are given initial access permissions.

Space.Access: TYPE = {readWrite, readOnly};

Each swap unit has its own **Access** status. **readWrite** specifies that clients are allowed to read and write in the swap units. **readOnly** specifies that only reading is allowed. Any attempt to write into a page of a swap unit which is **readOnly** results in a write protect fault. Operations are also provided for changing the access of existing swap units.

When an interval is mapped, its swap units are given an initial *life*. This specifies whether or not the initial contents of the backing file are useful.

Space.Life: TYPE = {alive, dead};

Each swap unit has its own **Life** status. **alive** specifies that a swap unit initially contains useful data; **dead** specifies that it does not. Pilot uses this information to avoid reading pages of the interval from backing storage and writing pages containing no useful data. When a swap unit is marked **dead**, the contents of each page will be unpredictable until that page is *written* into by the client. Until that time, the client can make no assumption about the contents of the pages or their consistency with the corresponding pages of the window. Pilot insists that **readOnly** swap units be **alive**; any attempt to make a **readOnly** swap unit be **dead** will be ignored—it will remain **alive**. A swap unit becomes **alive** when (1) one of its pages has been written into, or (2) it is made **readOnly**. A page can be swapped out either explicitly by the client or implicitly by Pilot in managing memory. The operation **Space.Kill** is provided to make existing swap units **dead**.

Any Space operation may raise the signal:

Space.Error: ERROR[type: space.ErrorType];

Space.ErrorType: TYPE = { ... };

Specific values of **ErrorType** are defined below. In addition, some operations may raise other signals as defined below.

If any Space operation is given an **Interval** whose pages are not completely contained within the implemented virtual memory of the system element, **Space.Error[pointerPastEndOfVirtualMemory]** is raised.

Space.ErrorType: TYPE = { ..., pointerPastEndOfVirtualMemory, ... };

Any Space operation that transfers data to backing storage may encounter an unrecoverable error in reading or writing the data. If so, it will raise the signal

```
Space.IOError: ERROR [page: Environment.PageNumber];
```

page is the first page of the data being transferred which is in error.

4.5.2 Mapping files to virtual memory intervals

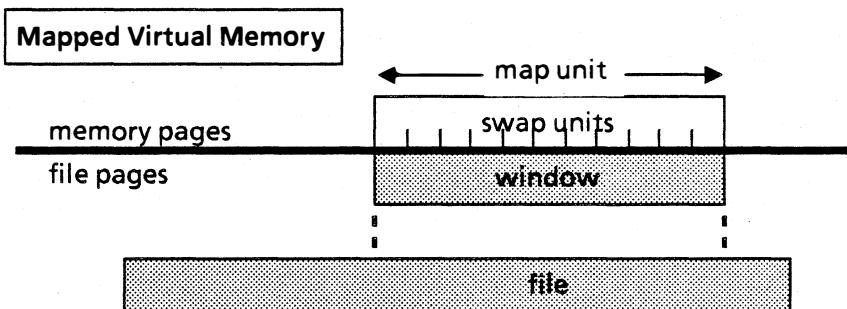
As described above, Pilot implements virtual memory by associating intervals of memory with *backing storage* consisting of a sequence of pages from some file. This sequence of file pages is called a *window*. Associating an area of virtual memory with a window is known as *mapping*; the resulting interval is called a *map unit*. Virtual memory is normally allocated when an interval is mapped.

A **Window** is a contiguous group of pages in a file starting at a specified base.

```
Space.Window: TYPE = RECORD [
  file: File.File,
  base: File.PageNumber,
  count: Environment.PageCount];
```

The window within the file starts at **base**, the first page relative to the beginning of the file, and extends for **count** pages or to the end of the file, whichever comes first. The *actual window length* is the lesser of **count** and the file size minus **base**. If **count** is set to **Environment.lastPageCount**, the window will extend to the end of the file.

When an interval is mapped, it is typically subdivided into modest-sized *swap units* to allow more efficient management of swapping. If there is no known grouping of the references to the pages of a map unit, uniform-sized swap units should be specified; this is the default. If there is no knowledge of the proper size for the uniform swap unit size, the client may request a default swap unit size. If there is some known grouping of the references to the pages of a map unit, the map unit may be subdivided into swap units with specific sizes and locations. In some circumstances, Pilot may break a client-specified swap unit into smaller swap units.



System performance can be severely degraded if a swap unit is a substantial fraction of the size of real memory. Clients should ensure that map units are divided into swap units of manageable size. As a general rule, a swap unit should not exceed one-tenth the size of real memory.

The operations for controlling the allocation of intervals and mapping them to windows are **Map**, **ScratchMap**, and **Unmap**.

```
Space.Map: PROCEDURE [
    window: Space.Window,
    usage: Space.Usage ← Space.unknownUsage,
    class: Space.Class ← file,
    access: Space.Access ← readWrite,
    life: Space.Life ← alive,
    swapUnits: Space.SwapUnitOption ← Space.defaultSwapUnitOption]
RETURNS [mapUnit: Space.Interval];

Space.Usage: TYPE = [0..2048];

Space.unknownUsage: Space.Usage = 0;

Space.Class: TYPE = MACHINE DEPENDENT {
    unknown(0), code(1), globalFrame(2), localFrame(3),
    zone(4), file(5), data(6), spareA(7), spareB(8), pilotResident(31)};

Space.SwapUnitOption: TYPE = RECORD [
    body: SELECT swapUnitType: Space.SwapUnitType FROM
        unitary = > NULL,
        uniform = > [size: Space.SwapUnitSize ← Space.defaultSwapUnitSize],
        irregular = > [
            sizes: LONG DESCRIPTOR FOR ARRAY [0..0] OF Space.SwapUnitSize]
        ENDCASE];

Space.SwapUnitType: TYPE = {unitary, uniform, irregular};

Space.defaultSwapUnitOption: Space.SwapUnitOption =
    [uniform[Space.defaultSwapUnitSize]];

Space.SwapUnitSize: TYPE = CARDINAL;

Space.defaultSwapUnitSize: Space.SwapUnitSize = 0;

Space.ErrorType: TYPE = {
    ... incompleteSwapUnits, invalidSwapUnitSize, invalidWindow, noWindow, ... };

Space.InsufficientSpace: ERROR [available: Environment.PageCount];
```

Map allocates an interval of virtual memory and associates it with a window of a file. The allocated interval is called a map unit. The window is then the backing store for the map unit. The length of the map unit is the actual window length, which is the lesser of **window.count** and the size of the file minus **window.base**. The allocated map unit is returned.

Caution: Clients must not delete the backing storage for any mapped interval or close the volume containing it. The behavior of Pilot in such circumstances is undefined.

Caution: Clients should ensure that different map units are not mapped to overlapping windows of a file if any of them is writable. The contents of the windows and the map units in such circumstances are unpredictable.

If **window.file** is **File.nullFile**, then **window.volume**, **window.base**, **life**, and **access** are ignored and Pilot supplies anonymous backing file storage for the interval. Such a window is called a *data window* (a window mapped to a file is called a *file window*). The length of the allocated window and map unit is **window.count**. The interval is mapped with **access = readWrite** and **life = dead**. Backing storage for data windows is allocated on the system volume. Information in data windows is discarded when the client **Unmaps** the interval or, if the system crashes, when the system volume is next opened for writing. For UtilityPilot-based systems, data windows are backed only by resident memory.

Map may encounter various conditions which will cause errors to be raised:

<u>Condition</u>	<u>ERROR</u>
Actual window length is 0	Space.Error[noWindow]
Not enough contiguous free virtual memory	Space.InsufficientSpace
window.base > File.lastPageNumber	Space.Error[invalidWindow]
Volume can't be located	Volume.Unknown
Volume partially online	Volume.NotOnline
Volume online and closed	Volume.NotOpen
File does not exist on the volume	File.Unknown
Any of the pages of window do not exist	File.MissingPages
Cannot supply backing file for a data window	Volume.InsufficientSpace
Volume is read-only, but access = readWrite	Volume.ReadOnly

Note that **Space.InsufficientSpace** passes back the maximum amount that could have been allocated.

The interval is mapped with the **access** given. If **access = readOnly**, **life** is ignored and the interval is mapped with **life = alive**. If **access = readWrite** but **window.volume** is read-only, **Volume.ReadOnly** is raised.

usage identifies the data in the map unit. The **usage** of map units will be available to the debugger and performance monitoring tools. The interface **SpaceUsage** defines subranges of **Space.Usage** for various clients and applications. Clients are encouraged to have their own private definitions file which further suballocates the **Space.Usages** assigned to them by the **SpaceUsage** interface.

class indicates the class of the data in the map unit. Pilot uses this data in its swapping decisions. Clients will normally specify only **file** for file windows and **data** for data windows.

If **swapUnits.swapUnitType = uniform**, the map unit is subdivided into equal-sized swap units of the indicated size. If **size** equals **defaultSwapUnitSize** or 0, Pilot will choose an appropriate size. If **size** equals or exceeds the size of the map unit, the swap unit serves no purpose; in this case specifying unitary swap units is more efficient.

If **swapUnits.swapUnitType = irregular**, the map unit is subdivided into irregular-sized swap units of the sizes given in **swapUnits.sizes**. Each element of **swapUnits.sizes** is the size of the corresponding swap unit. If the size of any irregular swap unit is greater than an implementation-dependent upper limit, it will be subdivided into smaller swap units.

Excess elements of `swapUnits.sizes` are ignored. If the window does not completely cover the last swap unit, this swap unit will be shorter than requested. If any required element of `swapUnits.sizes` is 0, `Space.Error[invalidSwapUnitSize]` is raised. If any required element of `swapUnits.sizes` is unmapped storage, an address fault will result. If the sum of the elements of `swapUnits.sizes` is less than the size of the map unit, `Space.Error[incompleteSwapUnits]` is raised.

If `swapUnits.swapUnitType = unitary`, the map unit is not subdivided into smaller swap units. This indicates the client's desire to have the map unit swap as a single entity.

`ScratchMap` is a more convenient way than `Map` to allocate temporary storage.

```
Space.ScratchMap: PROCEDURE [  
    count: PageCount, usage: Space.Usage ← space.unknownUsage]  
RETURNS [pointer: LONG POINTER];
```

The operation

```
Space.Unmap: PROCEDURE [  
    pointer: LONG POINTER, returnWait: Space.ReturnWait ← wait]  
RETURNS [nil: LONG POINTER];  
  
Space.ReturnWait: TYPE = {return, wait};  
  
Space.ErrorType: TYPE = {..., notMapped, ...};
```

removes the association between the map unit containing `pointer` and the map unit's window. This frees the map unit's virtual memory for other uses. If `returnWait = wait`, the operation does not return until the contents of the window reflect the contents of the interval. If `returnWait = return`, the operation returns immediately without waiting for any required output to complete. Pilot ensures, however, that client actions on the backing window have the same effect as if `returnWait = wait` had been specified. If the interval is mapped to a data window, the information in the window is discarded. If `pointer` is not contained in a map unit, `Space.Error[notMapped]` is raised. If the data in the interval cannot be written to the window, `Space.IOError` is raised.

Note: For the current release `returnWait = return` is equivalent to `returnWait = wait`.

Of course, pointers into a map unit should not be retained after unmapping. To encourage this, `Unmap` returns a `NIL` pointer. The intended usage is

```
myPointer ← Space.Unmap[myPointer];
```

References to an interval formerly occupied by the map unit can result in an address fault, or worse, may access or overwrite other data if the virtual memory is reused.

4.5.3 Explicitly reading and writing virtual memory

`CopyIn` and `CopyOut` are similar to read and write operations in a conventional file system. However, since the interval involved must already be mapped to a backing file, each can also be thought of as a file-to-file copy. Neither operation returns until the data has been transferred and neither changes the mapping of the interval.

The operation

Space.CopyIn: PROCEDURE [pointer: LONG POINTER, window: Space.Window]
RETURNS [countRead: Environment.PageCount];

Space.ErrorType: TYPE = { . . . , readOnly, . . . };

reads the contents of **window** into virtual memory starting at the page that contains **pointer**. **countRead** is the amount read, which is the lesser of **window.count** and the size of the file minus **window.base**. All virtual memory pages into which data will be read must be mapped. The contents of **window** are not changed by this operation.

Note: The virtual memory modified may start before **pointer** ↑ since reading starts at the first word of the page containing **pointer**.

Caution: Clients should not **CopyIn** from any part of a window currently mapped in virtual memory with write access. The data read in such circumstances is unpredictable.

If any portion of the virtual memory involved is read-only, **Space.Error[readOnly]** is raised. If any portion of the virtual memory involved is unmapped, **Space.Error[notMapped]** is raised. If the data cannot be read from the window, **Space.IOError** is raised. In all of these cases, the pages preceding the offending page may have been overwritten by the corresponding portion of **window**. See also the list of errors raised by both **CopyIn** and **CopyOut**, below.

The operation

Space.CopyOut: PROCEDURE [pointer: LONG POINTER, window: Space.Window]
RETURNS [countWritten: Environment.PageCount];

writes the current contents of virtual memory, starting at the page that contains **pointer**, out to **window**. **countWritten** is the amount written, which is the lesser of **window.count** and the size of the file minus **window.base**. All of the virtual memory pages from which data will be read must be mapped. The contents of virtual memory are not changed by this operation.

Note: The virtual memory being read may start before **pointer** ↑ since reading starts at a page boundary.

Caution: Clients should not **CopyOut** to any part of a window which is currently mapped in virtual memory. The contents of those map units in such circumstances is unpredictable.

If any portion of the virtual memory involved is unmapped, **Space.Error[notMapped]** is raised. If the data in the interval cannot be read from backing storage or if it can not be written to the given window, **Space.IOError** is raised. In both of these cases, the pages of the window corresponding to those preceding the offending virtual memory page may have been overwritten by the corresponding portion of virtual memory. If **window.volume** is read-only, **Volume.ReadOnly** is raised.

CopyIn and **CopyOut** both raise the following exceptions: If **window.base > File.lastPageNumber**, **Space.Error[invalidWindow]** is raised. If the volume cannot be located, **Volume.Unknown** is raised. **Volume.NotOnline** is raised if any part of the volume is

not online. If the volume is closed, **Volume.NotOpen** is raised. If the file does not exist on the volume, **File.Unknown** is raised. If any of the required pages of **window** do not exist, **File.MissingPages** is raised.

4.5.4 Swapping

Before a virtual memory location can be accessed, the page containing that location must be in real memory. If it is not, Pilot must read the contents of that page from its window into a real-memory page. If there is no available real memory page, Pilot makes room by writing pages to their backing window(s). Since Pilot keeps track of which pages match the contents of their window, it need not write unchanged pages.

There are two ways in Pilot to cause swapping: demand swapping, and controlled swapping.

4.5.4.1 Demand swapping

When a **PROCESS** attempts to reference a virtual page not currently in real memory, it causes a *page fault*. When a page fault occurs, execution of that **PROCESS** is suspended. Pilot reads in the page referenced and any adjoining swapped-out pages of the containing swap unit. This is known as *demand swapping*. The suspended **PROCESS** is blocked until the read operation is complete. Of course, any other ready **PROCESSES** are allowed to proceed concurrently with the handling of the page fault.

4.5.4.2 Controlled swapping

Pilot also swaps in response to *advice* given by the client indicating its intentions with respect to particular intervals. The operations provided allow the client to advise Pilot about:

- an interval that will be referenced soon;
- a recently referenced interval that will not be referenced for a while;
- an interval whose current contents are not wanted anymore (i.e. will be written before being read);

This advice enables Pilot to manage memory better than with simple demand swapping.

An operation is also provided to assure that the current contents of an interval are accurately reflected in its backing window. This is useful for transactional systems.

The operations **Activate**, **Deactivate**, and **Kill** allow the client to advise Pilot so it can manage swapping better. **ForceOut** allows the client to assure that the information in an interval will survive a system crash. Each of these operations can be applied to any interval of virtual memory, independent of map unit boundaries. The operations apply only to mapped portions of the specified interval, ignoring unmapped regions.

Space.Activate: PROCEDURE [interval: Space.Interval];

Space.Deactivate: PROCEDURE [interval: Space.Interval];

Activate indicates to Pilot that the interval is expected to be referenced in the near future and that Pilot should begin reading it in. This operation returns without waiting for any input to complete. **Deactivate** indicates to Pilot that the interval is not likely to be referenced soon, and that Pilot should write it out and release the real memory allocated to it. This operation also returns without waiting for any output to complete.

The following procedures allow the activation and deactivation of swap units containing Mesa code.

Space.ActivateProc: PROCEDURE [proc: --GENERIC-- PROCEDURE];

Space.DeactivateProc: PROCEDURE [proc: --GENERIC-- PROCEDURE];

Space.ErrorType: TYPE = { . . . , invalidProcedure, . . . };

ActivateProc causes the swap unit (code pack) containing the code for the procedure **proc** to be activated, and **DeactivateProc** deactivates it. If **proc** has arguments or results, normal usage is **ActivateProc[LOOPHOLE[proc, PROCEDURE]]**. If **proc** is not a valid procedure, **Space.Error[invalidProcedure]** is raised.

A common technique for using **ActivateProc** and **DeactivateProc** is to package a vacuous procedure with the code of interest. This procedure serves as a "handle" on a code pack, decoupling the function implemented by the code pack and the explicit procedures which compose it.

The operation

Space.Kill: PROCEDURE [interval: Space.Interval];

asserts to Pilot that the current contents of the interval are of no further value. **Kill** is intended to be used two ways: to avoid reading a page about to be overwritten, and to avoid writing a page which is no longer useful.

Pilot uses this information to avoid input/output activity on the interval. When **Kill** is applied to an interval, any real memory in the interval is immediately reclaimed; furthermore, any writable swap units wholly contained in the interval are marked **dead**. Pilot may supply arbitrary values for the contents of any page of a **dead** swap unit until the page is next *written* into by the client. The client should not make any assumptions about the contents of these pages or their consistency with the corresponding pages of the window (see also the previous discussion of the **Life** attribute).

The operation

Space.ForceOut: PROCEDURE [interval: Space.Interval];

causes the window(s) of the interval to agree with the current contents of virtual memory. It does not return until all required writing is complete. Any pages of the interval in real memory will remain there. Since Pilot keeps track of which pages match the contents of their window, **ForceOut** can bypass writing unchanged pages. If the data in the interval can not be written to the given window, **Space.IOError** is raised. If **ForceOut** causes any pages to be written to backing storage, the swap units containing those pages will be marked **alive**.

Any temporary disagreement between an interval and its window should be invisible during normal operation of the system. The intended use of **ForceOut** is to guarantee that the information in an interval will survive a system crash, by forcing it out to a non-volatile backing storage.

Calls on **Activate** and **Deactivate** may be added or deleted anywhere in a program without affecting its correctness. Calls on **Kill** may be deleted from, but not necessarily added to, a program without affecting its correctness. Calls on **ForceOut** may be added to, but not necessarily deleted from, a program without affecting its correctness.

4.5.5 Access control

The following operations allow portions of virtual memory to be made read-only or read-write.

Space.SetAccess: PROCEDURE [interval: Space.Interval, access: Space.Access];

This operation makes all swap units which include any portion of interval to be **readOnly** or **readWrite**. If the swap units were made **readOnly**, subsequent attempts to store into a page of any of these swap units will cause a write protect fault. If **access** = **readWrite** but the volume to which the interval is mapped is read-only, **Volume.ReadOnly** is raised.

When an interval is made **readOnly**, Pilot also does a **ForceOut** on the swap units and marks them **alive**. While doing this, if the data in the interval cannot be written to its window, **Space.IOError** is raised; in this case, the swap units preceding the offending page may have been made **readOnly** and **alive**.

If an arbitrary interval within a map unit is given, this operation may affect less virtual memory than that implied by the client-specified swap unit structure; this is because Pilot may occasionally break a client-specified swap unit into smaller swap units. A client can precisely specify which swap units are affected by having **interval.begin** and **end** on the boundaries of the client-specified swap units.

Note: The virtual memory affected may start before **interval.pointer**↑ since this operation starts at the first page of the swap unit containing **interval.pointer**↑. Similarly, the virtual memory affected may extend past **(interval.pointer + count * wordsPerPage)↑**.

Two convenience operations are also provided.

Space.MakeReadOnly: PROCEDURE [interval: Space.Interval] =
 INLINE { Space.SetAccess[interval, readOnly] };

Space.MakeWritable: PROCEDURE [interval: Space.Interval] =
 INLINE { Space.SetAccess[interval, readWrite] };

4.5.6 Explicit allocation of virtual memory and special intervals

Virtual memory is normally allocated when a window is mapped. However, facilities are also provided to allocate virtual memory explicitly, independent of the act of mapping.

4.5.6.1 Special intervals of virtual memory, main data spaces, and pointers

When virtual memory is being explicitly allocated, some intervals are of special interest:

Space.virtualMemory: READONLY Space.Interval;

virtualMemory describes the entirety of the virtual memory address space as actually implemented on the system element on which Pilot is running. The actual size of the virtual memory of a particular system element is given by **virtualMemory.count**.

A special kind of interval which is recognized by the Mesa processor and by Pilot is the *Main Data Space* (**MDS**). This interval consists of 256 pages (2^{16} words) and holds the Mesa run-time data structures needed to support the execution of a collection of **PROCESSES**. Every **PROCESS** is associated with some **MDS**. The procedure

Space.MDS: PROCEDURE RETURNS [Space.Interval];

returns the interval of the **MDS** of the **PROCESS** calling it. One **MDS** may be shared by many **PROCESSES**. A **PROCESS** may allocate virtual memory either inside or outside of its own **MDS**. Information within the **MDS** can be accessed by a **POINTER**, which is interpreted relative to the beginning of the **MDS**. Information outside of the **MDS** is accessed by a **LONG POINTER** or a **POINTER RELATIVE** to a **LONG BASE POINTER**. Since space in the **MDS** is typically in short supply, clients should normally allocate virtual memory outside the **MDS**. Executable code is not contained within any **MDS** and is shared by all **PROCESSES** in all **MDS**'s.

Note: Although the Mesa Processor allows multiple **MDS**'s, only a single **MDS** is implemented by the current version of Pilot.

4.5.6.2 Explicit allocation of virtual memory

Operations are provided for the explicit allocation and deallocation of an interval of virtual memory independent of the act of mapping.

Space.Allocate: PROCEDURE [
 count: Environment.PageCount, within: Space.Interval \leftarrow **Space.virtualMemory**,
 base: Environment.PageOffset \leftarrow **Space.defaultBase**]
RETURNS [interval: Space.Interval];

Space.defaultBase: Environment.PageOffset = ...;

Space.ErrorType: TYPE = { ..., alreadyAllocated, invalidParameters, ...};

This operation allocates an interval of unmapped virtual memory within an arbitrary containing interval. If **count** is zero, **Space.Error[invalidParameters]** is raised.

Managing an allocated interval is the responsibility of the client. Part or all of the interval may be used for mapping windows using **Space.MapAt**.

The client may either specify exactly the location of the interval to be allocated or have Pilot choose a suitable interval. To have Pilot choose a suitable starting location within the containing interval, the client passes **defaultBase**. If there are not enough contiguous unallocated pages in **within**, **Space.InsufficientSpace** is raised; this signal passes the

maximum amount that could have been allocated. To specify the location of the interval exactly, the client gives a base other than **defaultBase**. The interval to be allocated will start at the specified offset **base** from the start of the containing interval. If the requested interval would overlap an already allocated interval, **Space.Error[alreadyAllocated]** is raised. If the end of the interval would exceed the end of the containing interval, **Space.Error[invalidParameters]** is raised.

Note: When Pilot chooses the location of the interval, any special properly-contained subintervals of **within** (e.g., the MDS) may be skipped over. Thus Pilot may raise **Space.InsufficientSpace** when **within = Space.virtualMemory** even though there is still space available in the MDS.

The operation

```
Space.Deallocate: PROCEDURE [interval: Space.Interval];
```

```
Space.ErrorType: TYPE = { . . . , notAllocated, stillMapped, . . . };
```

deallocates the interval, making it available for other uses. **interval** should only contain virtual memory obtained from **Space.Allocate** or **Space.UnmapAt**. If any portion of the interval is mapped, **Space.Error[stillMapped]** is raised. If any portion of the interval is already deallocated, **Space.Error[alreadyDeallocated]** is raised. If **interval** exceeds the limits of implemented virtual memory, **Space.Error[invalidParameters]** is raised.

4.5.6.3 Mapping explicitly allocated virtual memory to files

The operations for controlling the mapping of explicitly allocated intervals are **MapAt** and **UnmapAt**.

```
Space.MapAt: PROCEDURE [
    at: Space.Interval,
    window: Space.Window,
    usage: Space.Usage ← Space.unknownUsage,
    class: Space.Class ← file,
    access: Space.Access ← readWrite,
    life: space.Life ← alive,
    swapUnits: Space.SwapUnitOption ← Space.defaultSwapUnitOption]
RETURNS [mapUnit: Space.Interval];
```

This operation maps a window of a file to virtual memory starting at **at.pointer**. The interval **at** must have been previously obtained from **Allocate** or **UnmapAt** or be a subinterval of one. The resulting interval is a map unit. The length of the map unit is the actual window length. If **at** contains any unallocated pages, **Space.Error[notAllocated]** is raised. If the end of the map unit would exceed the end of **at**, **Space.Error[invalidParameters]** is raised. This operation is otherwise analogous to **Space.Map** (q.v.).

The operation

```
Space.UnmapAt: PROCEDURE [
    pointer: LONG POINTER, returnWait: Space.ReturnWait ← wait]
RETURNS [interval: Space.Interval];
```

removes the association between the map unit which contains **pointer** and its window. **interval** describes the map unit being unmapped. If the virtual memory of the map unit was originally obtained from **Allocate**, the associated interval *remains* the property of the client. If the virtual memory of the map unit was originally obtained from **Map**, the client *acquires* the associated interval. The client retains this interval until it is **Deallocated**. This operation is otherwise identical to **Space.Unmap** (*q.v.*). Note that a client can **Unmap** an interval originally obtained from **Allocate** and subsequently mapped with **MapAt**; the associated interval becomes the property of Pilot.

4.5.7 Map unit and swap unit attributes, utility operations**The operation**

```
Space.GetMapUnitAttributes: PROCEDURE [pointer: LONG POINTER]
RETURNS [mapUnit: Space.Interval, window: Space.Window,
usage: Space.Usage, class: Space.Class, swapUnits: Space.SwapUnitOption];
```

returns the location and length of the map unit which contains **pointer**, the window to which it is mapped, the usage of the map unit, its swapping class, and the swap unit structure. If the map unit is mapped to a data window, the returned **window** will be **[File.nullID, Volume.nullID, 0, count]**. **window.count** (which equals the returned **interval.count**) reflects the actual size of the map unit. It may be less than the **window.count** given to **Map** or **MapAt** if the file was not long enough to supply the requested count. If **swapUnits.swapUnitType = uniform**, the returned **swapUnits.size** is the actual size of the swap units; **defaultSwapUnitSize** is never returned. If **swapUnits.swapUnitType = irregular**, the returned **swapUnits.sizes** is **NIL**; **GetSwapUnitAttributes** may be used to discover the sizes of irregular swap units. If **pointer** is not in any map unit, this operation returns **mapUnit = Space.nullInterval** and **window.count = 0**. Thus a pointer **p** points to unmapped storage if **GetMapUnitAttributes[p].mapUnit.count = 0**. If the map unit containing **pointer** was mapped by some facility other than **Space**, **Space.Error[invalidParameters]** is raised.

The operation

```
Space.GetSwapUnitAttributes: PROCEDURE [pointer: LONG POINTER]
RETURNS [swapUnit: Space.Interval, access: Space.Access, life: Space.Life];
```

returns the location, length, current access, and current life of the swap unit which contains **pointer**. The returned count reflects the actual size of the swap unit. In the case of uniform or irregular swap units, the size will differ from the size given to **Map** or **MapAt** if the requested size was zero or larger than Pilot implements. Also, Pilot may occasionally break a client-specified swap unit into smaller swap units. If **pointer** is not in any swap unit, this operation returns **interval = Space.nullInterval**.

The following operation returns the number of pages required to contain a specified number of words.

```
Space.PagesFromWords: PROCEDURE [wordCount: LONG CARDINAL]
    RETURNS [pageCount: Environment.PageCount] = ...;
```

The operation

```
Space.Pointer: PROCEDURE [pointer: LONG POINTER] RETURNS [POINTER];
```

converts a **LONG POINTER** to an equivalent **POINTER**. If the argument is not in the MDS of the calling **PROCESS**, **Space.Error[invalidParameters]** is raised.

The operation

```
Space.PointerFromPage: PROCEDURE [page: Environment.PageNumber] RETURNS [POINTER];
```

returns a **POINTER** which points to the first word of the argument page. If the argument is not in the MDS of the calling **PROCESS**, **Space.Error[invalidParameters]** is raised.

4.6 Pilot memory management

Four different facilities are available for acquiring and managing storage areas. Global frame space is considered a precious resource, but may be used for small (a few dozen words) storage that needs to be shared by multiple procedures and processes. Local frames, existing only as long as it's procedure instance, may be used for storage items that are less than a few hundred words in length and are not shared among procedures and processes. The Space machinery, described in detail in §4.5, provides contiguous groups of pages (256 word blocks) in the virtual memory and is most suitable for obtaining large blocks of storage. There is also a Pilot free storage package for managing arbitrarily sized nodes within client-designated areas of virtual memory called zones.

All state information pertaining to a zone is recorded within the zone itself, and, as a consequence, each zone can be managed independently of all others through the same interface, **zone**. The **Heap** facility provides further assistance in managing arbitrary sized nodes. The following properties distinguish a heap from a zone:

1. Heaps are more automatic, occupying system-designated (rather than client-designated) virtual memory, and expanding automatically (rather than requiring a client call).
2. Heaps are designed to support the Mesa language facilities for dynamic storage allocation (**UNCOUNTED ZONES**, **NEW**, **FREE**).
3. Some care is taken to treat large nodes (e.g., larger than 128 words) efficiently.
4. There is no mechanism to file away a heap and recreate it later.

It is expected that most Pilot clients will want to use the heap facilities. The zone facilities provide extra fine-grain control which may be useful for certain critical applications. Like

the zone facility, the heap performs best when the sizes of nodes are small compared to the size of the entire heap.

4.6.1 Zones

Zone: DEFINITIONS . . . ;

The Pilot zone management facility is based upon a suggestion by Don Knuth (*The Art of Computer Programming*, Volume 1, p. 453, #19). Within a zone, free nodes are kept as a linked list. One hidden word containing bookkeeping information is stored with each allocated node, and additional bookkeeping information is kept in the header of each zone. Allocation and release of nodes are usually very fast. Adjacent free nodes are always able to be coalesced. It is also possible to add new areas of virtual memory to enlarge a zone. These new areas, called *segments*, are linked together so that they may be deleted if all the nodes in a segment become free. In addition, an entire zone may be deleted. A zone may be saved in a file, and later *recreated* in memory at a different address.

The zone facility performs best when the sizes of nodes are small compared to the sizes of the block(s) making up the zone. A typical use for a zone is, for example, for small, transient data structures, such as the nodes of a temporary list structure or the bodies of (short) strings when the maximum length must be computed dynamically or the structure must outlive the frame that creates it. Use of a zone for large (i.e., multi-page) nodes decreases flexibility in storage management and is not recommended.

The allocator in the Pilot free storage package returns 16 bit pointers relative to a **LONG BASE POINTER** supplied at the time the zone is created. Note that these values are free pointers (type **RELATIVE POINTER TO UNSPECIFIED**) which must be cast appropriately (usually by assignment) before being used. Allocated nodes are not relocatable within the zone, and there is no garbage collection or automatic deallocation.

Because of its use for managing private, internal zones of Pilot, the zone facility raises no signals or errors. Instead, the various operations return a status from the enumerated type:

Zone.Status: TYPE = { . . . };

4.6.1.1 Zone management

A zone can be created from a block of client supplied virtual memory by calling the procedure **Create**.

```
Zone.Create: PROCEDURE [
  storage: LONG POINTER, length: Zone.BlockSize, zoneBase: Zone.Base,
  threshold: Zone.BlockSize ← Zone.minimumNodeSize, checking: BOOLEAN ← FALSE]
  RETURNS [zH: Zone.Handle, s: Zone.Status];
```

Zone.BlockSize: TYPE = CARDINAL;

Zone.Base: TYPE = Environment.Base;

Zone.minimumNodeSize: READONLY Zone.BlockSize;

Zone.Handle: TYPE [2] ;

Zone.nullHandle: Zone.Handle = ... ;

Zone.Status: TYPE = { ..., okay, storageOutOfRange, zoneTooSmall, ... };

A zone is created to occupy the number words of virtual memory specified by **length** and beginning at the word pointed to by **storage**. The argument **zoneBase** is a **LONG BASE POINTER** which supplies the base address for all relative pointer calculations in this zone. The argument **threshold** indicates the minimum size node that will be maintained by this zone. All allocation requests will be rounded up to this size and no unallocated fragments smaller than this will be left in the zone.

The argument **checking** indicates whether or not some internal checking of the consistency of the zone is turned on. The **checking** option is useful for helping debug client programs which are improperly using or freeing nodes in the zone. Because it causes each node to be checked on each zone operation, checking degrades performance somewhat.

The virtual memory must be mapped and have write permission. If it does not, an address fault or write protection fault will be generated as if the client program had attempted to write directly into that area of virtual memory. If **length** is too small to support a zone with at least one node of size **threshold**, a status of **zoneTooSmall** is returned. All segments of a zone must lie entirely within a single 64K word address space, i.e., all of the zone must be addressable by 16 bit relative pointers based on **base**. If that is not the case, or if the zone size is not in the range [0..2¹⁶), a status of **storageOutOfRange** is returned.

Caution: In this version of Pilot, zone sizes are restricted to the range [0..2¹⁵).

If a zone is successfully created, the **Create** operation returns a status of **okay** and a **Zone.Handle** which is used to identify the zone for all other zone operations.

nullHandle is never the **Handle** to an actual zone and is provided as a reference to the null zone.

A client may save a zone in a file for later use. Since the implementation of a zone may change from release to release, client code using *filed zones* must be prepared to cooperate in recovering from a "wrong version" condition detected by Pilot, as explained below. A client may request Pilot to resurrect an old zone, presumably one previously saved in a permanent file, with the procedure:

Zone.Recreate: PROCEDURE [storage: LONG POINTER, zoneBase: Zone.Base]

RETURNS [zH: Zone.Handle, rootNode: Zone.Base RELATIVE POINTER, S: Zone.Status];

Zone.Status: TYPE = { . . . , wrongSeal, wrongVersion};

The **storage** parameter to **Recreate** should point to a place in virtual memory which is mapped to a file window containing the contents of a zone created (or recreated) earlier in the same or an earlier run. While the **storage** and corresponding **zoneBase** need not remain fixed each time a zone is recreated, the arithmetic difference between them must be kept invariant. Note also that the relative positions of any segments added to the zone must stay invariant.

Normally **Recreate** returns a status of **okay**, together with an ordinary zone handle for the zone and the value of the *root node* of the zone. However it is possible that an incompatible implementation change in Pilot has been made since the zone was created, in which case **Recreate** returns a status of **wrongVersion**, an invalid zone handle, and the correct value of the root node of the old zone. *In this case it is the client's responsibility to rebuild a new version of the zone, perhaps by enumerating the nodes reachable from the root node via fields defined within the client node format(s).* Finally, a status of **wrongSeal** indicates a client programming error: the storage passed to Pilot does not begin with a fixed "seal" value, and probably never contained a valid zone. In this case, the returned handle and root node are both undefined.

Zone.GetRootNode: PROCEDURE [zH: Zone.Handle]
RETURNS [node: Zone.Base RELATIVE POINTER];

Zone.SetRootNode: PROCEDURE [zH: Handle, node: Zone.Base RELATIVE POINTER];

Zone.nil: READONLY Zone.Base RELATIVE POINTER;

To support the notion of a filed zone, Pilot allows a *root node* to be associated with every zone. This value, initially set to **Zone.nil**, is just a short relative pointer which the client may use to point to a distinguished node within the zone, thus providing a "point of purchase" on the data structures contained within the zone. As discussed above, the entire set of nodes in a filed zone should be enumerable from the root (unless the entire data structure can be reconstructed from some other source).

The Mesa construct **NIL** does not apply to **RELATIVE POINTERS** such as those used to reference nodes. For this reason, the constant **Zone.nil** is provided for representing the nil **RELATIVE POINTER**.

There is no explicit operation for destroying a zone. The client program merely recovers the storage it had provided and ceases to use the zone.

The following procedure returns the attributes of a zone.

Zone.GetAttributes: PROCEDURE [zH: Zone.Handle]
RETURNS [zoneBase: Zone.Base, threshold: Zone.BlockSize,
checking: BOOLEAN, storage: LONGPOINTER, length: Zone.BlockSize,
next: Zone.SegmentHandle];

Zone.SegmentHandle: TYPE [1];

Zone.nullSegment: READONLY Zone.SegmentHandle;

- The results **zoneBase**, **threshold**, **storage**, and **length** are exactly as specified when the zone was created. The result **checking** indicates whether or not consistency checking is currently enabled for this zone (see below). The result **next** is a handle for an additional segment of this zone (see §4.6.1.2); **zone.nullSegment** is returned if there are no additional segments in this zone. No validity check is made of **zH**, the **Zone.Handle**, prior to returning these results.

The following operation is used to enable or disable consistency checking of the zone. If **checking** is **TRUE**, a consistency check is made that all of the nodes in the zone, and the data structures of the zone, are well-formed.

```
Zone.SetChecking: PROCEDURE [zH: Zone.Handle, checking: BOOLEAN]
RETURNS [s: Zone.Status];
```

```
Zone.Status: TYPE = { . . . , invalidZone, invalidSegment, invalidNode, nodeLoop, . . . };
```

A status of **invalidZone** indicates the the basic data structures of the zone identified by **zH** are malformed. A status of **invalidSegment** indicates that although the primary block of virtual memory in the zone is okay, one of its segments (see §4.6.1.2) is malformed. A status of **invalidNode** indicates that within the zone, some node is malformed or invalid. This could mean that the overhead word of the node has been overwritten, that a 'node' has been freed which does not lie within the virtual memory constituting the zone, or that a 'free' node is not properly linked on the free list in the zone. A status of **nodeLoop** indicates that the free list has a loop within it. Except as otherwise indicated, any of these status results can be returned if consistency checking is enabled and the corresponding condition is detected during the execution of any of the operations in the **Zone** interface.

4.6.1.2 Segment management

The virtual memory provided to the zone at the time it is created is the *primary storage* of the zone. It is of fixed size and cannot be reclaimed by the client so long as the zone is of any value. Additional blocks of storage can be added to the zone by the procedure:

```
Zone.AddSegment: PROCEDURE [zH: Zone.Handle, storage: LONG POINTER,
length: Zone.BlockSize]
RETURNS [sH: Zone.SegmentHandle, s: Zone.Status];
```

```
Zone.Status: TYPE = { . . . , segmentTooSmall, . . . };
```

This operation creates a new segment of the zone containing the number of words indicated by **length** and beginning at the virtual memory word pointed to by **storage**. The virtual memory of the segment must be mapped and have write permission. If it does not, an address fault or write-protect condition will be generated as if the client had written or referenced that part of virtual memory directly. This area of virtual memory must also be addressable by 16 bit pointers relative to the **zoneBase** of the zone, and **length** must be in the range [0..2¹⁶). If it is not, a status of **storageOutOfRange** is returned. If **length** does not specify enough virtual memory to implement a segment and to contain at least one node of size **threshold**, a status of **segmentTooSmall** is returned.

Caution: In this version of Pilot, segment sizes are restricted to the range [0..2¹⁵).

All segments of a zone are linked together in a list pointed to by the **nextSegment** attribute of the zone. The attributes of any segment, including the next member of the list are returned by:

```
Zone.GetSegmentAttributes: PROCEDURE [zH: Zone.Handle, sH: Zone.SegmentHandle]
RETURNS [storage: LONG POINTER, length: Zone.BlockSize, next: Zone.SegmentHandle];
```

A segment may be removed from a zone if it contains no allocated nodes. This is accomplished by the procedure:

```
Zone.RemoveSegment: PROCEDURE [zH: Zone.Handle, sH: Zone.SegmentHandle]
    RETURNS [storage: LONG POINTER, S: Zone.Status];
```

```
Zone.Status: TYPE = { ..., nonEmptySegment, ... };
```

A status of **okay** indicates that the segment was successfully removed. A status of **nonEmptySegment** indicates that the segment still contains allocated nodes and that therefore it could not be removed. A status of **invalidZone** or **invalidSegment** is returned if the data structures of the zone are not well-formed enough to permit removal of the segment.

4.6.1.3 Node allocation and deallocation

The operations of this section provide the facilities for allocating and deallocating nodes in a zone.

```
Zone.MakeNode: PROCEDURE [zH: Zone.Handle, n: Zone.BlockSize,
    alignment: Zone.Alignment ← a1]
    RETURNS [node: Zone.Base RELATIVE POINTER, S: Zone.Status];
```

```
Zone.Alignment: TYPE = { a1, a2, a4, a8, a16 };
```

```
Zone.Status: TYPE = { ..., noRoomInZone, ... };
```

MakeNode allocates a node of **n** words in the zone identified by **zH**. An optional **alignment** may be specified for this node, in which case the node is aligned in virtual memory as follows:

if **alignment** is set to **a1** then the node is word aligned

if **alignment** is set to **a2** then the node is double word aligned

if **alignment** is set to **a4** then the node is quad word aligned

if **alignment** is set to **a8** then the node is eight word aligned

if **alignment** is set to **a16** then the node is sixteen word aligned

If a node of at least **n** words of the desired alignment can be allocated, a 16 bit pointer relative to the **zoneBase** of the zone is returned pointing to the node, along with a status of **okay**. More than the requested number of words will be allocated to avoid fragmentation of the free space remaining in the zone into pieces of size less than the **threshold** of the zone. If a contiguous block of space is not available in the zone, a status of **noRoomInZone** is returned. The value **Zone.nil** is returned by **MakeNode** if it is unable to allocate a node.

If **B** is the **zoneBase** of the zone and **node** is the relative pointer returned by **MakeNode** then a Mesa **LONG POINTER** to the node is represented by the expression **@B[node]**. If **B** =

Space.MDS[].pointer then the expression LOOPHOLE[node, POINTER] is a Mesa short pointer to the node.

Zone.FreeNode: PROCEDURE [zH: Zone.Handle, p: LONG POINTER]
RETURNS [s: Zone.Status];

This operation deallocates the node pointed to by **p** in the zone indicated by **zH**. If the node does not lie within that part of virtual memory addressable by 16 bit relative pointers based on the **zoneBase** of the zone, or the node is not marked in use, a status of **invalidNode** is returned. Otherwise, a status of **okay** is returned. More detailed checking, including that the node actually lies within the zone (or one of its segments) is only done if consistency checking is enabled.

Zone.SplitNode: PROCEDURE [zH: Zone.Handle, p: LONG POINTER, n: Zone.BlockSize]
RETURNS [s: Zone.Status];

This operation splits the node pointed to by **p**, retaining the first **n** words and freeing the remainder. No split occurs if the remainder would be smaller than the **threshold** of the zone.

Zone.NodeSize: PROCEDURE [p: LONG POINTER] RETURNS [n: Zone.BlockSize];

This operations returns the actual size of the node pointed to by **p** (this may exceed the allocated size to avoid fragmentation). No check is made to determine the validity of the node.

4.6.2 Heaps

Heap: DEFINITIONS . . . ;

The heap facility consists of the Pilot interface **Heap** together with some language features built into Mesa. The operations in **Heap** are primarily concerned with creating and deleting heaps. Almost all node allocation and deallocation may be performed using Mesa **NEW** and **FREE** constructs, which also allow initialization and pointer management. The reader is assumed to be familiar with these Mesa features.

4.6.2.1 Heap management

There are three types of heaps: *normal*, *uniform*, and *MDS*. Normal heaps allow allocation of arbitrary sized objects. Uniform heaps allow allocation of objects whose size is equal to or less than a fixed size. The *MDS* heaps allow allocation of arbitrary sized objects from within the *MDS*.

Normal and uniform heaps are identified by a value of type **UNCOUNTED ZONE**, MDS heaps by a value of type **MDSZone**. Pilot provides a standard normal heap and a standard *MDS* heap:

Heap.systemZone: READONLY UNCOUNTED ZONE;

Heap.systemMDSZone: READONLY MDSZone;

Note that the **READONLY** attribute applies not to the contents but to the reference to the particular heap.

The system provided heaps can be used to share information between subsystems. If a subsystem requires a lot of private storage it is often more efficient to create a private heap than to use the system provided heaps. If objects being allocated are all the same size, uniform heaps are more efficient since less overhead is required for each node. To create additional heaps, call either **Create** to create a normal heap, **CreateUniform** to create a uniform heap, or **CreateMDS** to create an MDS heap.

```
Heap.Create: PROC [
    initial: Environment.PageCount,
    maxSize: Environment.PageCount ← Heap.unlimitedSize,
    increment: Environment.PageCount ← 4,
    swapUnitSize: Space.SwapUnitSize ← Space.defaultSwapUnit,
    threshold: Heap.NWords ← Heap.minimumNodeSize,
    largeNodeThreshold: Heap.NWords ← Environment.wordsPerPage/2,
    ownerChecking: BOOLEAN ← FALSE,
    checking: BOOLEAN ← FALSE]
    RETURNS [UNCOUNTED ZONE];

Heap.CreateUniform: PROC [
    initial: Environment.PageCount,
    maxSize: Environment.PageCount ← Heap.unlimitedSize,
    increment: Environment.PageCount ← 4,
    swapUnitSize: Space.SwapUnitSize ← Space.defaultSwapUnit,
    objectSize: Heap.NWords,
    ownerChecking: BOOLEAN ← FALSE,
    checking: BOOLEAN ← FALSE]
    RETURNS [UNCOUNTED ZONE];

Heap.CreateMDS: PROC [
    initial: Environment.PageCount,
    maxSize: Environment.PageCount ← Heap.unlimitedSize,
    increment: Environment.PageCount ← 4,
    swapUnitSize: Space.SwapUnitSize ← Space.defaultSwapUnit,
    threshold: Heap.NWords ← Heap.minimumNodeSize,
    largeNodeThreshold: Heap.NWords ← Environment.wordsPerPage/2,
    ownerChecking: BOOLEAN ← FALSE,
    checking: BOOLEAN ← FALSE]
    RETURNS [MDSZone];

Heap.NWords: TYPE = [0..32766];

Heap.unlimitedSize: Environment.PageCount = ...;

Heap.minimumNodeSize: READONLY Heap.NWords;

Heap.Error: ERROR [type: Heap.ErrorType];
```

```
Heap.ErrorType: TYPE = { . . . , maxSizeExceeded, invalidParameters, invalidSize,
insufficientSpace, otherError, . . . };
```

When an allocation request would exceed the current size of a heap, the heap is automatically expanded by **increment** pages. It is still a good idea to specify a reasonable value for **initial** to minimize fragmentation. (The expansions to a heap are not, in general, contiguous in virtual memory.)

If a nondefault **maxSize** is specified, the signal **Heap.Error[maxSizeExceeded]** is raised when a heap is being created or expanded, or a large node is being allocated, and the total number of pages allocated for the heap exceeds **maxSize**. The signal **Heap.Error[insufficientSpace]** is raised when the underlying zone implementation returns a status to the **Heap** package that is either unexpected or not understood.

If a nondefault **swapUnitSize** is specified, the spaces created to hold the heap and its extensions will have uniform swap units of size **swapUnitSize**. If it is defaulted, no swap units will be created.

For normal or MDS heaps, the argument **threshold** indicates the minimum size node that will be maintained by this heap. All allocation requests will be rounded up to this size and no unallocated fragments smaller than this will be left in the heap. The argument **largeNodeThreshold** indicates the size of node which will not be allocated in the normal fashion. Allocation requests of this size or larger will be handled by creating a separate space for each, which is deleted when the node is deallocated.

For uniform heaps, the argument **objectSize** indicates the size node that will be maintained by this heap. All allocation requests greater than this size will result in the signal **Heap.Error[invalidSize]** being raised.

If **ownerChecking** is **TRUE**, owner checking is enabled (see the description in §4.7.2.3 below of the operation **CheckOwner**). The argument **checking** indicates whether or not some internal checking of the consistency of the heap is turned on.

The checking option is useful for helping debug client programs which are improperly using or freeing nodes in the heap. However, because it checks each node on each heap operation, it does degrade performance noticeably.

A heap may be deleted with one of the operations, as appropriate:

```
Heap.Delete: PROCEDURE [z: UNCOUNTED ZONE, checkEmpty: BOOLEAN ← FALSE];
```

```
Heap.DeleteMDS: PROCEDURE [z: MDSZone, checkEmpty: BOOLEAN ← FALSE];
```

If **checkEmpty** is **TRUE**, then **Heap.Error[invalidHeap]** will be raised if there are still nodes in the heap which have not been deallocated.

4.6.2.2 Node allocation and deallocation

Nodes are allocated from a heap using the Mesa **NEW** operator and are deallocated using the Mesa **FREE** statement. For the remainder of this section, assume that **z** and **mz** have

been declared as a **UNCOUNTED ZONE** and an **MDSZone**, respectively, and have been initialized. For example:

z: UNCOUNTED ZONE = Heap.SystemZone;

mz: MDSZone = Heap.systemMDSZone;

or

z: UNCOUNTED ZONE = Heap.Create[initial: . . .];

mz: MDSZone = Heap.CreateMDS[initial: . . .];

(It is also possible to initialize **z** and **mz** by assignment subsequent to their declaration.)

If **T** is a type and **t** is an expression of type **T** then

z.NEW[T ← t]

allocates a node of size at least **SIZE[T]**, sets its contents to **t**, and returns a long pointer to the node. Similarly for

mz.NEW[T ← t]

except a short pointer is returned. If **p** is a **LONG POINTER TO T** pointing to a node previously allocated from **z** then

z.FREE[@p];

sets **p** to **NIL** and frees the node **p** had pointed to (in that order). Similarly, if **mp** is a **POINTER TO T** pointing to a node previously allocated from **mz** then

mz.FREE[@mp];

sets **mp** to **NIL** and frees the node **mp** had pointed to (in that order). In both cases of **FREE**, if **p** is **NIL** then the operation is a no-op.

A special construct is provided for allocating a string body from a heap:

z.NEW[StringBody[n]]

allocates a node large enough to hold a string body of **n** characters, initializes its **length** field to 0 and its **maxlength** field to **n** (but leaves its **text** field uninitialized), and returns a **LONG STRING** pointing to the node. Similarly for

mz.NEW[StringBody[n]]

except a short **STRING** is returned.

4.6.2.3 Miscellaneous operations

It is possible to determine the initial parameters and current statistics of a heap by calling the appropriate one of:

```
Heap.GetAttributes: PROC [z: UNCOUNTED ZONE]
  RETURNS [
    heapPages, maxSize, increment: Environment.PageCount,
    swapUnitSize: Space.SwapUnitSize,
    ownerChecking, checking: BOOLEAN, attributes: Heap.Attributes];

Heap.Attributes: TYPE = RECORD [
  SELECT tag: Type FROM
    normal = > [
      largeNodePages: Environment.PageCount,
      threshold, largeNodeThreshold: Heap.NWords],
    uniform = > [objectSize: Heap.NWords],
  ENDCASE];

Heap.GetAttributesMDS: PROC [z: MDSZone]
  RETURNS [
    heapPages, largeNodePages, maxSize, increment: Environment.PageCount,
    swapUnitSize: Space.SwapUnitSize,
    threshold, largeNodeThreshold: Heap.NWords,
    ownerChecking, checking: BOOLEAN];
```

If a client is about to create a large number of nodes which together would cause a heap to expand by more than **increment** (the parameter to **Create**) pages, some fragmentation may be avoided by first calling:

```
Heap.Expand: PROCEDURE [z: UNCOUNTED ZONE, pages: Environment.PageCount];
```

```
Heap.ExpandMDS: PROCEDURE [z: MDSZone, pages: Environment.PageCount];
```

The client can return the heap to the state it had when it was created by calling:

```
Heap.Flush: PROCEDURE [z: UNCOUNTED ZONE];
```

```
Heap.FlushMDS: PROCEDURE [z: MDSZone];
```

All nodes that were allocated are freed and all extensions to the heap are freed.

If many nodes have been deallocated from a heap, for example at the end of some intermediate phase of activity, it may be possible to release some of the virtual memory occupied by that heap. The operations

```
Heap.Prune: PROCEDURE [z: UNCOUNTED ZONE];
```

```
Heap.PruneMDS: PROCEDURE [z: MDSZone];
```

examine each of the spaces containing expansions to the heap **z**, releasing any containing no nodes.

If a heap was created with **ownerChecking = TRUE**, then the procedures

Heap.CheckOwner: PROCEDURE [p: LONG POINTER, z: UNCOUNTED ZONE];

Heap.CheckOwnerMDS: PROCEDURE [p: LONG POINTER, z: MDSZone];

Heap.ErrorType: TYPE = { . . . , invalidOwner, . . . };

may be called to determine if a node was allocated by the same module (global frame) as the caller of **CheckOwner**. If not, **Heap.Error[invalidOwner]** will be raised.

It may be determined whether or not **ownerChecking = TRUE** by calling

Heap.OwnerChecking: PROCEDURE [z: UNCOUNTED ZONE] RETURNS [BOOLEAN];

Heap.OwnerCheckingMDS: PROCEDURE [z: MDSZone] RETURNS [BOOLEAN];

The checking feature, described in §4.6.2.1 above, may be turned on and off by:

Heap.SetChecking: PROCEDURE [z: UNCOUNTED ZONE, checking: BOOLEAN];

Heap.SetCheckingMDS: PROCEDURE [z: MDSZone, checking: BOOLEAN];

Heap.ErrorType: TYPE = { . . . , invalidHeap, invalidNode, invalidZone, . . . };

There may be times when it is convenient to allocate untyped storage, say for a variable-length structure not defined as a Mesa **SEQUENCE**. Several procedures are provided for these cases. Wherever possible it is preferable to use **NEW** and **FREE** instead, redefining types in terms of **SEQUENCE** where necessary. The following two procedures allocate a node of the specified size, returning a pointer to the new node:

**Heap.MakeNode: PROCEDURE [
 z: UNCOUNTED ZONE ← systemZone, n: NWords] RETURNS [LONG POINTER];**

**Heap.MakeMDSNode: PROCEDURE [
 z: MDSZone ← systemMDSZone, n: NWords] RETURNS [POINTER];**

The following two procedures deallocate the specified node. If **p** is **NIL** the operation is a no-op.

Heap.FreeNode: PROCEDURE [z: UNCOUNTED ZONE ← systemZone, p: LONG POINTER];

Heap.FreeMDSNode: PROCEDURE [z: MDSZone ← systemMDSZone, p: POINTER];

4.7 Logging

Log: DEFINITIONS . . . ;

LogFile: DEFINITIONS . . . ;

These interfaces supply a general purpose facility for recording information in a client-supplied *log file*. These facilities allow logging words, blocks of words, and strings, turning the log on and off, limiting the entries placed in the log based on a severity level, initializing and resetting the log file, and controlling the action taken when it fills up. Additional facilities are provided for subsequently examining the contents of a log file. The implementation modules for the logging facility are **LogImpl.bcd** and **LogFileImpl.bcd**.

4.7.1 Writing into the log file

The procedures in the **Log** interface are used to write into the log file, and to install the log file, start and stop logging, and other control functions. The file used for the log is supplied by the client, and its properties (length, type, etc.) are not changed by the logging package; only its content is modified. This allows the client to retain control of the log file for purposes of examining it, copying it, displaying it to field service personnel, etc.

4.7.1.1 Installing, opening, and closing the log file

Install is used to initialize a log file. It is normally called only during system generation when a file system is being built.

Log.Install: PROCEDURE [file: File.File, firstPageNumber: File.PageNumber← 1];

Log.logCap: READONLY File.File;

Log.Error: ERROR [reason: Log.ErrorType];

Log.ErrorType: TYPE = MACHINE DEPENDENT {illegalLog, tooSmallFile, . . . };

Install will format the file starting at **firstPageNumber**. Pages preceding **firstPageNumber** will not be used by the logging package. **Log.Error[illegalLog]** is raised if there is already a current log file. **Log.Error[tooSmallFile]** is raised if the usable size of **file** is too small. **Install** also automatically performs an **Open** (see below). The currently installed log file is kept in the variable **logCap**.

Caution: In the current version of Pilot, the minimum usable size of a log file is 4 pages. Also, the logging package will not use more than 256 pages of a log file.

Log.Open: PROCEDURE [file: File.File, firstPageNumber: File.PageNumber← 1];

Log.ErrorType: TYPE = { . . . , invalidFile, . . . };

Open prepares the logging package to write log entries into **file**, which becomes the *currently installed log file*. This must be done before any entries may be written into the log. **Open** is typically used after a system restart to re-establish logging on an existing log file (one that has already been formatted as a log). This procedure does *not* reset the

contents of the log; new entries will be added to the end. **Log.Error[invalidFile]** is raised if **file** has not been formatted as a log file, or if logging is currently open on a *different file*. Opening the current log file is a no-op.

Log.Close: PROCEDURE [];

Log.ErrorType: TYPE = { . . . , logNotOpened, . . . };

Close causes all current log entries to be forced out to the log file and the logging facility to stop accessing it. It ceases to be the current log file. **Log.Error[logNotOpened]** is raised if there is no current log file.

4.7.1.2 Writing entries in the log file

Procedures are provided for logging three data types: a single word, a block of words, or a string.

Log.PutWord: PROCEDURE [level: Log.Level, data: UNSPECIFIED, forceOut: BOOLEAN ← FALSE];

Log.PutBlock: PROCEDURE [

level: Log.Level, pointer: LONG POINTER, size: CARDINAL, forceOut: BOOLEAN ← FALSE];

Log.PutString: PROCEDURE [

level: Log.Level, string: LONG STRING, forceOut: BOOLEAN ← FALSE];

Log.Level: TYPE = Log.State[error..remark];

Log.State: TYPE = MACHINE DEPENDENT {off, error, warning, remark};

An entry is only written to the log if its **level** is less than or equal to the current state (see §4.7.1.3). If **forceOut** is true, the buffer containing the entry is forced out to the file. The length of a log entry is restricted to a maximum of 255 words; **PutBlock** and **PutString** will truncate an entry if necessary. **Log.Error[logNotOpened]** is raised if there is no current log file. Except for their order, the logging package attaches no particular semantics to the levels; the names used are meant only to be suggestive of the ordering.

Log.SetRestart: PROCEDURE [message: UNSPECIFIED];

SetRestart allows the client to write a special entry in a log file. This "message" entry is the only entry in a log file that may be overwritten. This entry could be used by a backstop (see Chapter 9) to communicate to its client when and why the client last crashed. The client could obtain this information by reading the restart entry of its backstop's log file. **Log.Error[logNotOpened]** is raised if there is no current log file.

4.7.1.3 Controlling logging

The following procedures can be used to control what information is recorded in the log file:

Log.SetState: PROCEDURE [state: Log.State];

Log.GetState: PROCEDURE RETURNS [state: Log.State];

Log.Disable: PROCEDURE RETURNS [Log.State];

Log.Reset: PROCEDURE [];

SetState specifies what levels of log entries are to be written into the log file. Subsequently, any call that specifies a **level** less than or equal to the current state will make an entry in the log. The current state is initially set to **error**. Note that if the state is **off**, all logging calls are ignored, since **level** is never less than or equal to **off**. **GetState** returns the current value of the state. **Disable** sets the current state to **off**, with the side effect of forcing out any internal buffering to backing storage. It also returns the previous value of the state. **Reset** will reset the log file to the beginning, thereby completely emptying it; this also flushes buffers. **Log.Error[logNotOpened]** is raised if there is no current log file.

Log.SetOverflow: PROCEDURE [option: Log.Overflow];

Log.Overflow: TYPE = MACHINE DEPENDENT {reset, disable, wrap};

SetOverflow allows the client to specify what is to be done when the log file becomes full. If **reset** is specified, the log will start over at the beginning (this will invalidate all previous entries). If **disable** is specified, logging will be turned off; Log entries will continue to be accepted, but their contents will be discarded. If **wrap** is specified, the log will behave like a ring buffer, with a new entry overwriting the oldest one. Logging is initially set for **wrap** mode. **Log.Error[logNotOpened]** is raised if there is no current log file.

4.7.1.4 Properties of the current log file

The following procedures can be used to determine the properties of the current log file:

Log.GetCount: PROCEDURE RETURNS [count: CARDINAL];

Log.GetIndex: PROCEDURE RETURNS [index: Log.Index];

Log.GetLost: PROCEDURE RETURNS [lost: CARDINAL];

Log.GetUpdate: PROCEDURE RETURNS [time: System.GreenwichMeanTime];

Log.Index: TYPE = CARDINAL;

Log.nullIndex: Index = 0;

Log.ErrorType: TYPE = {..., logNoEntry, ...};

GetCount returns the current number of entries, counting from the beginning of the log file. **GetIndex** returns the current index into the log file. **GetLost** returns the number of entries that have been lost due to log overflow (for overflow mode of **disable**). **GetUpdate** returns the time of the last log entry, or raises **Log.Error[logNoEntry]** if the log is empty. **Log.Error[logNotOpened]** is raised if there is no current log file.

4.7.2 Reading a log file

The procedures defined in **LogFile** interface are used to examine a log file. They should not be applied to the current log file. If it is necessary to read the current log file, the client must **Log.Close** it first.

If the file supplied to any **LogFile** operation does not appear to be formatted as a log file, the error **LogFile.InvalidFile** is raised. If the file is the current log file, the error **LogFile.IllegalEnumerate** is raised.

LogFile.InvalidFile: ERROR;

LogFile.IllegalEnumerate: ERROR;

The following procedures can be used to determine the properties of a log file. They parallel those of the same name in the **Log** interface.

**LogFile.GetCount: PROCEDURE [file: File.File, firstPageNumber: File.PageNumber ← 1]
RETURNS [count: CARDINAL];**

**LogFile.GetLost: PROCEDURE [file: File.File, firstPageNumber: File.PageNumber ← 1]
RETURNS [count: CARDINAL];**

The following procedure is used to enumerate the entries of a log file.

**LogFile.GetNext: PROCEDURE
[file: File.File, current: Log.Index, firstPageNumber: File.PageNumber ← 1]
RETURNS [next: Log.Index];**

LogFile.Inconsistent: ERROR;

GetNext is a stateless enumerator with a starting and ending value of **nullIndex**. If **current** appears to contain garbage, **GetNext** will raise **Inconsistent**. This situation could arise if the system crashed before the last page of the log was written to the log file. Therefore, this error can be used to detect the last entry before the system crashed.

**LogFile.GetAttributes: PROCEDURE
[file: File.File, current: Log.Index, firstPageNumber: File.PageNumber ← 1]
RETURNS [time: System.GreenwichMeanTime, type: LogFile.Type,
level: Log.Level, size: CARDINAL];**

**LogFile.GetBlock: PROCEDURE [file: File.File, current: Log.Index,
place: LONG POINTER, firstPageNumber: File.PageNumber ← 1];**

**LogFile.GetString: PROCEDURE [file: File.File, current: Log.Index,
place: LONG STRING, firstPageNumber: File.PageNumber ← 1];**

LogFile.Type: TYPE = MACHINE DEPENDENT {null (0), block (1), string (2), (63)};

LogFile.DifferentType: ERROR;

GetAttributes will return the type, level and size of an entry, as well as the time at which it was written. Only two types of entries are returned: If **type** is set to **block**, **size** is the number of words in the block. If **type** is set to **string**, **size** is the number of characters in the string. A single word log entry is treated as a **block** of size one. Once the type and size of an entry are determined, **GetBlock** or **GetString** can be used to copy the entry into storage supplied by the client. If **GetBlock** is called to copy a string entry or **GetString** is called to copy a block entry, the error **LogFile.DifferentType** is raised.

```
LogFile.Reset: PROCEDURE [file: File.File, firstPageNumber: File.PageNumber ← 1];
```

Reset will reset a log file to be empty. The file could then be reestablished as the current log file using **Open**.

```
LogFile.GetRestart: PROCEDURE [file: File.File, firstPageNumber: File.PageNumber ← 1]
RETURNS [restart: LogFile.Restart];
```

```
LogFile.Restart: TYPE = MACHINE DEPENDENT RECORD [
    message(0): UNSPECIFIED, time(1): System.GreenwichMeanTime];
```

GetRestart allows the client to read a special entry from a log file and to obtain the time that entry was last written. This "restart" entry is the only entry in a log file that may be read without enumerating the entries. The **message** returned is the **restart** supplied to **Log.SetRestart**. If **SetRestart** was never called for that log file, **time** will have the value **System.gmtEpoch** and the value of **message** will be undefined. The "restart" entry might be used by a client to examine his backstop's log file to determine when and why he last crashed. For the client to interpret **message**, he must have independent knowledge of the values given to **message** by the system that wrote it.

C

C

C



I/O Devices

The facilities described in this section provide the lowest level standard access to input/output devices through Pilot. Two concepts are defined: software channel and device driver. A *software channel* is a Mesa interface to a device. It specifies all of the device-specific data and control information which a client needs to operate the device. A *device driver* is a set of programs which actually implement and export a software channel. It includes all of the necessary "interrupt" routines, interfaces with microprograms, control of hardware registers, etc., to service the device. It may be part of Pilot or it may be supplied by another organization for a special purpose device.

Initializing a software channel binds the client to a physical resource and device driver. Each channel represents a single device. Shared resources, such as common controllers, are normally hidden from view so that, for example, each drive unit connected to a common controller is treated as a distinct device. The device drivers hold the decision making power over the allocation of these shared resources. In the case that this does not provide the proper control, it will be necessary to construct a new device driver.

The concept of software channel is common to all devices and all channels have a common style. However, Pilot does not provide a central, common interface to all of them. Instead, each channel is represented by its own Mesa **DEFINITIONS** module. The common style is presented in this section in the form of the specification of a hypothetical device called **ExampleDevice**. The channel interfaces for specific devices exported by Pilot are given later in this section. In addition, client development groups may add additional channels to Pilot for specialized or private devices.

5.1 Channel structure and initialization

To create and initialize a software channel for **ExampleDevice**, the client calls

```
ExampleDevice.Create: PROCEDURE [assign: ExampleDevice.WhichDevice,  
drive: CARDINAL]  
RETURNS [ExampleDevice.ChannelHandle];
```

```
ExampleDevice.WhichDevice: TYPE = {any, specified};
```

```
ExampleDevice.ChannelHandle: TYPE = PRIVATE . . . ;
```

ExampleDevice.DeviceNotAvailable: . . . ;

The **assign** parameter indicates how to choose among multiple instances of a device. If **any** is specified, the device driver allocates any instance of that device. If **specified** is passed, then the device driver selects the drive indicated by **drive**. If the channel cannot be initialized for any reason, the routine signals *ExampleDevice.DeviceNotAvailable*.

Device drivers which support multiple instances of a device also define the operation

ExampleDevice.GetDrive: PROCEDURE [channel: *ExampleDevice.ChannelHandle*]
RETURNS [drive: CARDINAL];

This operation is used to identify the specific device associated with the **ChannelHandle**.

Deleting a channel and releasing the associated device are accomplished by

ExampleDevice.Delete: PROCEDURE [channel: *ExampleDevice.ChannelHandle*];

This operation calls *ExampleDevice.Abort* before returning. If the client wishes to complete all pending transfers he should first call *ExampleDevice.Suspend*.

The following operations allow a client to control the data transfer activity on a specific device.

ExampleDevice.Suspend: PROCEDURE [channel: *ExampleDevice.ChannelHandle*];

This operation waits for all pending transfers (i.e., as a result of previously executed calls on *ExampleDevice.Get* and *ExampleDevice.Put*) to complete before returning. Subsequent calls on **Get**, **Put**, or any control operations are ignored. However, calls on **TransferWait** for previously outstanding transfers will return normally.

ExampleDevice.Restart: PROCEDURE [channel: *ExampleDevice.ChannelHandle*];

This operation restarts a suspended channel. A channel may become suspended (with no pending operations) as a result of the **Suspend** operation or (with some operations pending) as the result of the occurrence of a sufficiently serious error.

ExampleDevice.Abort: PROCEDURE [channel: *ExampleDevice.ChannelHandle*];

This operation aborts all activity on the indicated channel. Any outstanding data transfer operations will be immediately terminated with a **TransferStatus** = [**TRUE, aborted**] (see §5.1.1.3 for **TransferStatus**).

5.1.1 Data transfer

The operations described below transmit information to and from a device. This data transfer is asynchronous so that many input and output operations can be simultaneously pending.

Each device may impose its own constraints on the alignment of data in memory. This is specified by three constants declared (statically) in the interface to the software channel.

`ExampleDevice.alignment: CARDINAL = ...;`

`ExampleDevice.granularity: CARDINAL = ...;`

`ExampleDevice.truncation: CARDINAL = ...;`

These three values must be specified and clients of devices must adhere to them. These requirements are normally imposed by certain high-performance devices to maintain physical memory bandwidth, satisfy physical constraints in the implementation of the controllers, etc. In particular, the device may constrain:

each I/O buffer to be aligned on a virtual memory address which is a multiple of alignment;

each I/O buffer in virtual memory to have a length which is an integral multiple of granularity; and

each physical record on the device to have a length which is a multiple of truncation.

Each of these constants must be a power of two in the range [0..256]. A value of zero is interpreted to represent *byte* alignment, granularity, and truncation; a value of one represents *word* alignment, granularity, and truncation; a value of four represents *quadword* alignment, granularity, and truncation; a value of sixteen represents *16-word* alignment, granularity, and truncation; and a value of 256 represents *page* alignment, granularity, and truncation.

Normally, **granularity** is greater than or equal to **truncation**. On output, the buffer must be a multiple of **granularity**, but the physical record may be truncated to a multiple of **truncation**. On input, the buffer must also be a multiple of **granularity**. If a shorter (i.e., truncated) record is read, the remainder of the buffer may be filled with garbage.

5.1.1.1 Data transfer types

The following data structures are the most general form for describing the source or destination of the data being transferred. Specific software channels may define simpler versions of these which, for example, omit the **header** or **trailer**, **startIndex**, etc.

`ExampleDevice.PhysicalRecordHandle: TYPE = LONG POINTER TO ExampleDevice.PhysicalRecord;`

`ExampleDevice.PhysicalRecord: TYPE = RECORD [header: ExampleDevice.BlockDesc,`
`body: ExampleDevice.BlockDesc, trailer: ExampleDevice.BlockDesc];`

`ExampleDevice.BlockDesc: TYPE = RECORD [blockPointer: LONG POINTER TO UNSPECIFIED,`
`startIndex, stopIndexPlusOne: CARDINAL];`

The **PhysicalRecord** specifies control information for the transfer operation in the **header** and **trailer**. The **body** specifies the buffer to or from which data is transferred. Quantities such as disk addresses and communication packet routing information are placed in the **header** and **trailer** blocks in a device dependent way.

If necessary, the **alignment**, **granularity**, and **truncation** may be specified separately for the **header**, **body**, and **trailer**.

ExampleDevice.CompletionHandle: TYPE = PRIVATE . . . ;

The **CompletionHandle** identifies the I/O transaction initiated by a **Get** or a **Put** operation. It is passed as parameter to the **TransferWait** operation, which does not return until that particular I/O operation is completed. **Get** and **Put** are asynchronous and return to the caller as soon as the request has been queued and made pending. **TransferWait** completes the operation and returns the number of bytes transferred and the resulting **TransferStatus**.

5.1.1.2 Data transfer procedures

*ExampleDevice.Get: PROCEDURE [channel: ExampleDevice.ChannelHandle,
rec: ExampleDevice.PhysicalRecordHandle]
RETURNS [ExampleDevice.CompletionHandle];*

This operation queues the **PhysicalRecord** for input transfer and returns to the client with the input transfer pending. The **CompletionHandle** must be submitted to the **TransferWait** operation in order to complete the transfer and before any of the input information can be used.

*ExampleDevice.Put: PROCEDURE [channel: ExampleDevice.ChannelHandle,
rec: ExampleDevice.PhysicalRecordHandle]
RETURNS [ExampleDevice.CompletionHandle];*

This operation queues the **PhysicalRecord** for output transfer and returns to the client with the output transfer pending. The **CompletionHandle** must be submitted to the **TransferWait** operation in order to complete the transfer and before the output record can be reused.

For both **Get** and **Put**, the I/O buffers described by the **PhysicalRecord** must not be released, altered, or reused until after the **TransferWait** operation for this transfer completes. In particular, any control information contained, for example, in the **header** or **trailer** buffers will be read or processed in place by the device rather than stored internally.

*ExampleDevice.TransferWait: PROCEDURE [channel: ExampleDevice.ChannelHandle,
event: ExampleDevice.CompletionHandle]
RETURNS [byteCount: CARDINAL, status: ExampleDevice.TransferStatus];*

This operation completes the processing of the I/O and returns the number of bytes transferred and the status to the client. The **CompletionHandle** specifies the particular pending transfer to await. If the channel has been aborted, **status** = [**TRUE**, **aborted**].

5.1.1.3 Data transfer status

Transferring data can provoke a number of errors. When a serious error occurs, the channel is suspended. In any case Pilot returns the **TransferStatus** as the result of the **TransferWait** procedure. The client can then examine this status and take corrective action. If this status indicates that the channel has been suspended, it must be restarted

after corrective action is taken and before any further data transfers are possible. A **Restart** allows I/O transactions to continue over the channel.

```
ExampleDevice.TransferStatus: TYPE = RECORD [error: BOOLEAN,  
type: ExampleDevice.TransferErrors];
```

```
ExampleDevice.TransferErrors: TYPE = {aborted, ... };
```

If no errors were encountered then **error** is **FALSE**. If errors were encountered then **error** is **TRUE** and the particular error is identified in **type**.

5.1.2 Device specific commands

Most devices need a number of device specific auxiliary operations which are not specified by the common channel style. **Rewind** for a magnetic tape is an example.

Some of these operations are for direct and simple communication with the device driver and involve no physical I/O, e.g.,

```
ExampleDevice.SetNumberOfRetries: PROCEDURE [channel: ExampleDevice.ChannelHandle,  
numberOfRetries: CARDINAL];
```

Others might invoke an I/O operation which is not a data transfer, e.g.,

```
ExampleDevice.Rewind: PROCEDURE [channel: ExampleDevice.ChannelHandle];
```

Completion of this kind of operation is detected via **StatusWait** described below.

Yet others might initiate I/O operations which are similar to data transfers and may choose to use the **CompletionHandle** and **TransferWait** mechanisms to detect completion, e.g.,

```
ExampleDevice.VerifyData: PROCEDURE [channel: ExampleDevice.ChannelHandle,  
rec: ExampleDevice.PhysicalRecordHandle]  
RETURNS [ExampleDevice.CompletionHandle];
```

5.1.3 Device status

In addition to the status information returned for each data transfer operation, Pilot maintains global information about the device itself in the **DeviceStatus** record. This contains state information about the static and long term state of the device. It is accessed via the **GetStatus** and **StatusWait** procedures.

```
ExampleDevice.DeviceStatus: TYPE = RECORD [ ... ];
```

```
ExampleDevice.GetStatus: PROCEDURE [channel: ExampleDevice.ChannelHandle]  
RETURNS [ExampleDevice.DeviceStatus];
```

```
ExampleDevice.StatusWait: PROCEDURE [channel: ExampleDevice.ChannelHandle,  
stat: ExampleDevice.DeviceStatus]  
RETURNS [ExampleDevice.DeviceStatus];
```

StatusWait waits until the current **DeviceStatus** differs from the supplied parameter **stat**. The client must examine the device status to determine what action to take.

5.2 Keyset, keyboards, and mouse

Keys: **DEFINITIONS . . . ;**

KeyStations: **DEFINITIONS . . . ;**

LevelIVKeys: **DEFINITIONS . . . ;**

LevelVKeys: **DEFINITIONS . . . ;**

JLevelIVKeys: **DEFINITIONS . . . ;**

The state of the keys on the keyboard is described by an array of bits. These are packed into an array of words maintained by Pilot but readable by the client. The following exported variable provides access to this array.

UserTerminal.keyboard: **READONLY LONG POINTER TO READONLY ARRAY OF WORD;**

The mouse buttons and the keyset are considered keys and therefore occupy positions in this array.

The interpretation of the bits of this array is not specified by Pilot, but is instead specified by one or more separate **DEFINITIONS** modules associated with each particular keyboard. This permits Pilot to support more than one kind of keyboard layout. In the current version of Pilot, there are three such **DEFINITIONS** modules. **LevelIVKeys** defines the bits for the U.S. Dandelion keyboard, **JLevelIVKeys**, the Japanese Dandelion keyboard, and **LevelVKeys**, the Dove keyboard.

The **Keys** and **KeyStations** modules are obsolete and are included only for backward compatibility.

Figures 5.2a, 5.2b, and 5.2c at the end of this section show the assignments of keys on the keyboards to bits in the **UserTerminal.keyboard** array.

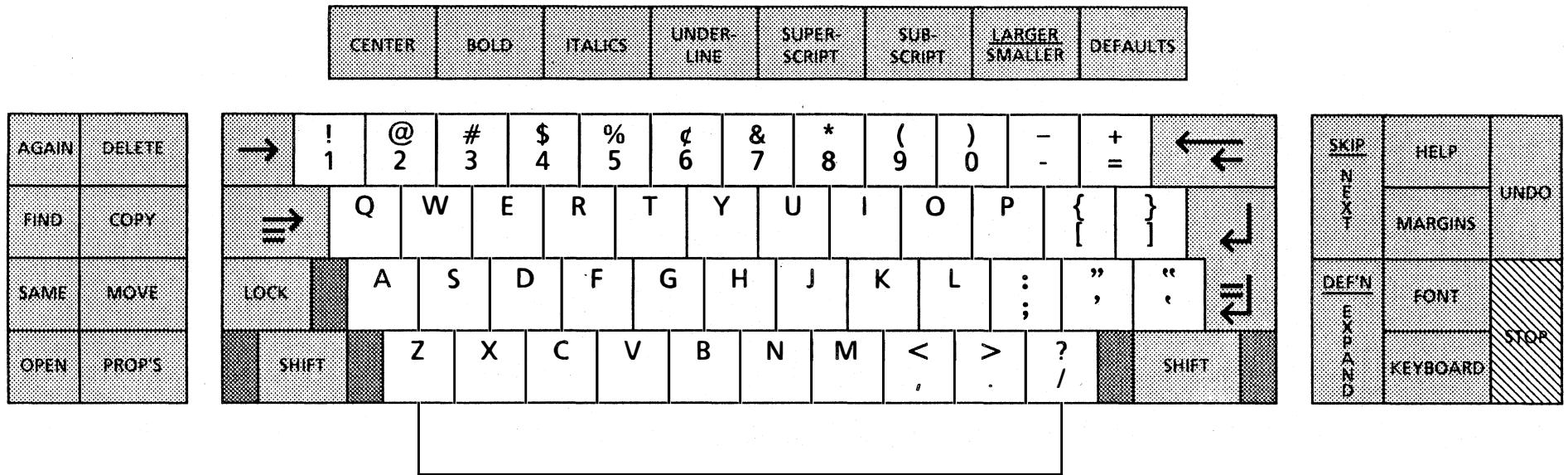
The table below lists the names given to each bit in the **UserTerminal.keyboard** array by the **LevelVKeys** interface. For historical reasons, the key names are not always the same as the names printed on the keyboards. The columns in the table have the following meaning.

Bit: the *n*th element in the **UserTerminal.keyboard** bit array.

Name: the key name used to refer to this bit.

Bit	Name	Bit	Name
0	---	56	Z
1	Bullet	57	LeftShift
2	SuperSub	58	Period
3	Case	59	SemiColon

4	Strikeout	60	NewPara
5	KeypadTwo	61	OpenQuote
6	KeypadThree	62	Delete
7	SingleQuote	63	Next
8	KeypadAdd	64	R
9	KeypadSubtract	65	T
10	KeypadMultiply	66	G
11	KeypadDivide	67	Y
12	KeypadClear	68	H
13	Point	69	Eight
14	Adjust	70	N
15	Menu	71	M
16	Five	72	Lock
17	Four	73	Space
18	Six	74	LeftBracket
19	E	75	Equal
20	Seven	76	RightShift
21	D	77	Stop
22	U	78	Move
23	V	79	Undo
24	Zero	80	Margins
25	K	81	KeypadSeven
26	Dash	82	KeypadEight
27	P	83	KeypadNine
28	Slash	84	KeypadFour
29	Font	85	KeypadFive
30	Same	86	English
31	BS	87	KeypadSix
32	Three	88	Katakana
33	Two	89	Copy
34	W	90	Find
35	Q	91	Again
36	S	92	Help
37	A	93	Expand
38	Nine	94	KeypadOne
39	I	95	DiagnosticBitTwo
40	X	96	DiagnosticBitOne
41	O	97	Center
42	L	98	KeypadZero
43	Comma	99	Bold
44	Quote	100	Italic
45	RightBracket	101	Underline
46	Open	102	Superscript
47	Special	103	Subscript
48	One	104	Smaller
49	Tab	105	KeypadPeriod
50	ParaTab	106	KeypadComma
51	F	107	LeftShiftAlt
52	Props	108	DoubleQuote
53	C	109	Defaults
54	J	110	Hiragana
55	B	111	RightShiftAlt

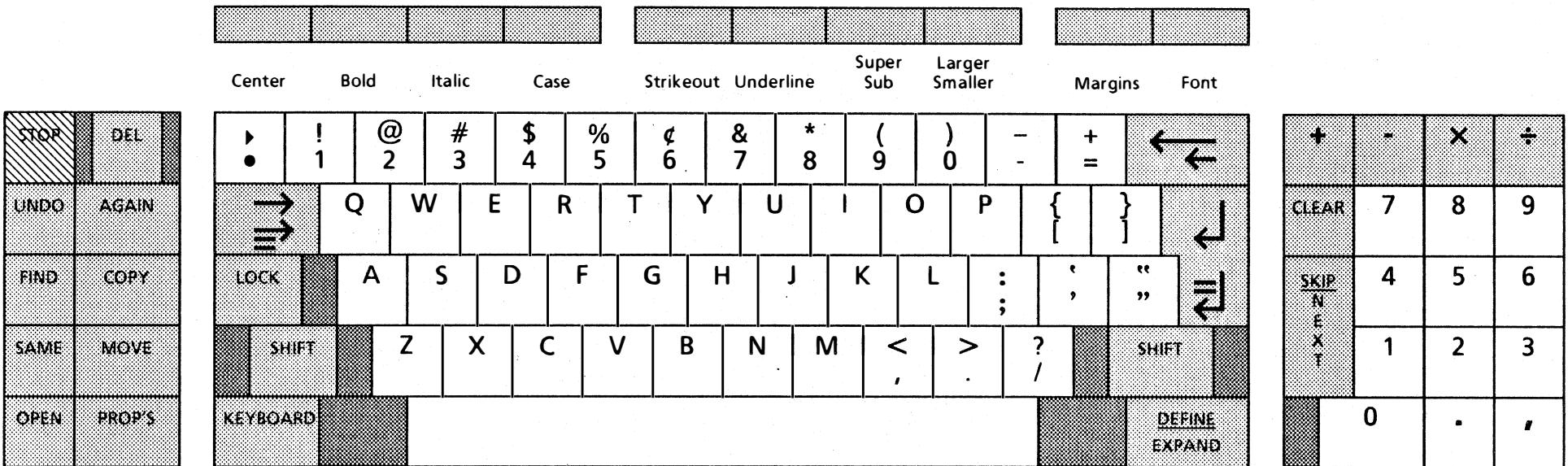


Dandelion US Key Assignments

97	99	100	101	102	103	104	109
91	62	49	48	33	32	17	16
90	89	50	35	34	19	64	65
30	78	72	37	36	21	51	66
46	52	57	56	40	53	23	55
73							

Dandelion US Key Numbering

Fig. 5.2.a - Level IV Keyboard



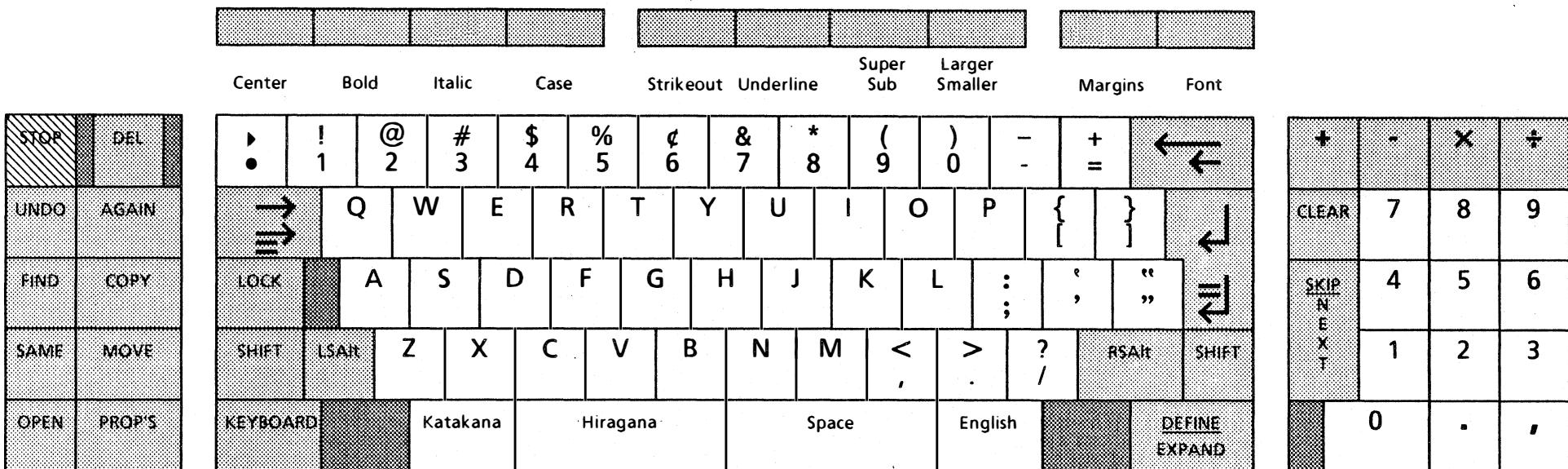
Dove US Key Assignments

97	99	100	3	4	101	2	104	80	29
----	----	-----	---	---	-----	---	-----	----	----

77	62	1	48	33	32	17	16	18	20	69	38	24	26	75	31
79	91	50	35	34	19	64	65	67	22	39	41	27	74	45	60
90	89	72	37	36	21	51	66	68	54	25	42	59	7	108	
30	78	57	56	40	53	23	55	70	71	43	58	28	76		
46	52	47												93	

Dove US Key Numbering

Fig. 5.2.b - Level V Keyboard



Dove Total Key Assignments

97	99	100	3	4	101	2	104	80	29
----	----	-----	---	---	-----	---	-----	----	----

77	62
79	91
90	89
30	78
46	52

1	48	33	32	17	16	18	20	69	38	24	26	75	31
50	35	34	19	64	65	67	22	39	41	27	74	45	60
72	37	36	21	51	66	68	54	25	42	59	7	108	
57	107	56	40	53	23	55	70	71	43	58	28	111	76
47			88		110		73		86			93	

8	9	10	11
12	81	82	83
63	84	85	87
	94	5	6
98	105	106	

Dove Total Key Numbering

Fig. 5.2.c - Level V Keyboard

5.3 The user terminal

UserTerminal: DEFINITIONS . . . ;

UserTerminalExtras: DEFINITIONS . . . ;

UserTerminal describes the state of the user input/output devices -- the display image (as represented by a one-bit-per-pixel bitmap), the display cursor, the keyboard, the mouse, and the keyset -- and allows the client to manipulate them. This interface assumes the configuration of the user terminal is as is given above. It does allow the client to deal with variations such as the number of keys or the size and resolution of the display. This interface deals with many of the lowest level attributes of the terminal. Within a typical client system, only a small user interface component will call **UserTerminal** directly. Definitions and operations of general interest are presented first, followed by more specialized ones.

UserTerminalExtras provides interim support for smooth scrolling. It will become part of **UserTerminal** in a future version of Pilot.

5.3.1 The display image

UserTerminal.screenWidth: READONLY CARDINAL [0..32767];

UserTerminal.screenHeight: READONLY CARDINAL [0..32767];

UserTerminal.pixelsPerInch: READONLY CARDINAL;

The attributes of the image are defined by the above exported variables. **screenWidth** and **screenHeight** specify the number of usable, visible picture elements in a row or column of the screen.

The bitmap image is addressed by x-y coordinates. The coordinate origin (0, 0) is the uppermost, leftmost pixel of the display; **x** increases to the right, and **y** increases downward.

UserTerminal.Coordinate: TYPE = MACHINE DEPENDENT RECORD [x, y: INTEGER];

The state of the display is defined as

UserTerminal.State: TYPE = {on, off, disconnected};

where

on indicates the display is physically on and visible to the user (and a bitmap is allocated);

off indicates the display is physically off and not visible to the user (but a bitmap is allocated);

disconnected indicates the same as **off** but with no bitmap allocated.

Clients may determine the current state of the bitmap display by calling

UserTerminal.GetState: PROCEDURE RETURNS [state: UserTerminal.State];

The bitmap display is capable of displaying black-on-white or white-on-black. Clients may determine or alter the current state of the background by using the following procedures. In the image, a pixel whose value is one is considered the figure; a pixel of zero, background.

UserTerminal.GetBackground: PROCEDURE
RETURNS [background: UserTerminal.Background];

UserTerminal.SetBackground: PROCEDURE [new: UserTerminal.Background]
RETURNS [old: UserTerminal.Background];

UserTerminal.Background: TYPE = {white, black};

Clients may momentarily *blink* (video reverse) the display by calling

UserTerminal.BlinkDisplay: PROCEDURE;

Some displays have the capability to display a border around the outside of the active display region. Clients can determine if the display has this capability by interrogating the following exported variable.

hasBorder: READONLY BOOLEAN;

If the display has a border, then clients may set the pattern to be displayed in the border by calling

UserTerminal.SetBorder: PROCEDURE [oddPairs, evenPairs: [0..377B]];

The bit pattern for an individual scan line is defined by displaying a single byte repeatedly along the entire scan line. The same pattern is shown on alternating pairs of lines. Thus, **evenPairs** is the byte used on lines -4, -3, 0, 1, 4, 5, etc.; **oddPairs** is the byte used on lines -2, -1, 2, 3, 6, 7, etc. Calling **SetBorder** when **hasBorder** is FALSE will lead to unpredictable results.

The following function is provided for clients who need to synchronize bitmap alteration with display refresh. Waiting for scan line zero is also a commonway for a user input handler to wait between polls of the keyboard and mouse buttons.

UserTerminal.WaitForScanLine: PROCEDURE [scanLine: INTEGER];

The following procedure will return a BitBlt table with the bitmap fields filled in for the current bitmap.

UserTerminal.GetBitBltTable: PROCEDURE RETURNS [bbt: BitBlt.BBTable];

The bitmap parameters are returned in **bbt** in such a fashion that a **BitBlt** using it will copy the bitmap from itself to itself. For a complete description of a **BBTable** see the description of BitBlt in the *Mesa Processor Principles of Operation*. The bits-per-line in the returned **bbt** may be different than **screenWidth** if the display implementation has non-visible padding bits appended to each line.

WaitForScanLine and **GetBitBltTable** will raise the following error if the display is disconnected (deallocated).

UserTerminal.BitmapIsDisconnected: ERROR;

Clients may alter the state of the bitmap and display by calling

**UserTerminal.SetState: PROCEDURE [new: UserTerminal.State]
RETURNS [old: UserTerminal.State];**

Setting the state to **disconnected** invalidates any **BBTables** previously returned by **GetBitBltTable**, but setting the state to **off** does not. The bitmap is zeroed (i.e., set to all background) when the state is changed from **disconnected** to **on**. Disconnecting destroys any information that may have been contained in the bitmap.

UserTerminal.CursorArray: TYPE = ARRAY [0..16] OF WORD;

The display cursor is defined by a 16x16 bit array, whose bits are **OR**'ed with the bitmap. The top row is contained in **CursorArray[0]**; the bottom row in **CursorArray[15]**. The most significant bits of each entry in the array correspond to the left portion of the cursor image; the least significant bits correspond to the right portion.

Clients can determine the current bit pattern for the cursor by calling

**UserTerminal.GetCursorPattern: PROCEDURE
RETURNS [cursorPattern: UserTerminal.CursorArray];**

The cursor pattern is set by calling

UserTerminal.SetCursorPattern: PROCEDURE [cursorPattern: UserTerminal.CursorArray];

The coordinates of the cursor can be found by the following exported variable.

UserTerminal.CURSOR: READONLY LONG POINTER TO READONLY UserTerminal.Coordinate;

The position of the cursor on the display may be altered by calling the procedure

UserTerminal.SetCursorPosition: PROCEDURE [newCursorPosition: UserTerminal.Coordinate];

5.3.2 Smooth scrolling

The smooth scrolling interface allows a client to create a window within the display area that can be scrolled up or down. Clients may create a scroll window by calling

**UserTerminalExtras.CreateScrollWindow: PROCEDURE [locn: UserTerminal.Coordinate,
width: CARDINAL, height: CARDINAL];**

UserTerminalExtras.scrollXQuantum: READONLY CARDINAL;

UserTerminalExtras.scrollYQuantum: READONLY CARDINAL;

UserTerminalExtras.Error: ERROR [type: UserTerminalExtras.ErrorType];

```
UserTerminalExtras.ErrorType: TYPE = {multipleWindows, . . . , yQuantumError,  
xQuantumError};
```

The horizontal bit-position of the scroll window within the bitmap (**loc.x**) and the width of the scroll window (**width**) must be multiples of **scrollXQuantum**. The vertical bit position of the scroll window (**loc.y**) and the height of the scroll window (**height**) must be multiples of **scrollYQuantum**. Thus, a value of 16 for **scrollXQuantum** indicates that left and right edges are word aligned within the bitmap.

If the constraints on **loc**, **height**, and **width** are not observed, **Error[xQuantumError]** or **Error[yQuantumError]** will be raised. **Error[multipleWindows]** will be raised if a scroll window already exists.

```
UserTerminalExtras.ScrollingInhibitsCursor: READONLY BOOLEAN;
```

On some processors, the presence of a smooth scrolling window inhibits display of the cursor, in which case **scrollingInhibitsCursor** is **TRUE**.

Clients cause the display to be scrolled up or down by calling

```
UserTerminalExtras.Scroll: PROCEDURE [line: LONGPOINTER TO UNSPECIFIED, lineCount: CARDINAL,  
increment: INTEGER];
```

```
UserTerminalExtras.ErrorType: TYPE = {. . . , noScrollWindow, lineCountError, . . . };
```

This procedure adds scan lines to the top or bottom of the scroll window, causing the window to scroll up or down. **line** points to the first bit within the first scan line to be moved into the scroll window. **lineCount** indicates how many scan lines are to be moved into the scroll window. **lineCount** must be a multiple of **scrollYQuantum**. The number of lines moved into the scroll window each time controls the speed of the scrolling. As each scan line is moved into the scroll window, **increment** is added to **line** to produce the bit address of the next scan line. The direction of the scroll is specified by the sign of **increment**. If **increment** is positive, lines are added to the bottom of the window, causing it to scroll up. If **increment** is negative, lines are added to the top of the window, causing it to scroll down.

During scrolling, the scan lines in the scroll window portion of the bitmap may not be in the same order in memory as they appear on the display.

If no scroll window exists, the error **Error[noScrollWindow]** will be raised. If **lineCount** is not a multiple of **scrollYQuantum**, **Error[lineCountError]** will be raised.

The scroll window may be deleted by calling

```
UserTerminalExtras.DeleteScrollWindow: PROCEDURE;
```

If **scrollingInhibitsCursor** is **TRUE**, there may be a delay before the cursor appears again while the scan lines in the scroll window are being sorted into their proper order.

The error **Error[noScrollWindow]** will be raised if no scroll window exists.

5.3.3 The keyboard and keyset

The keyboard and keyset defined in this interface are uninterpreted. That is, up/down key transitions are noted by the state of the bits in the following unencoded array:

UserTerminal.keyboard: READONLY LONG POINTER TO READONLY ARRAY OF WORD;

UserTerminalExtras2.keyboardType: READONLY KeyBoardType;

UserTerminalExtras2.KeyBoardType: TYPE = MACHINE DEPENDANT {
 learSiegle(0), level4(1), jLevel5(2), level5(3), other(LAST[CARDINAL])};

keyboardType gives the type of the keyboard attached to the system element. **learSiegle** implies that a Lear Siegler CRT is attached. **level4** implies that a Level 4 keyboard is attached; this is the keyboard usually attached to Dandelion processors. **jLevel5** is a Level 5 keyboard for JStar. **level5** is the American version of the Level 5 keyboard.

The **Extras** interface is interim for this release and will be merged with its parent interface in future releases.

5.3.4 The mouse

The coordinates of the mouse can be found by the following exported variable.

UserTerminal.MOUSE: READONLY LONG POINTER TO READONLY UserTerminal.Coordinate;

Clients can alter the coordinates of the current mouse position by calling

UserTerminal.SetMousePosition: PROCEDURE [newMousePosition: UserTerminal.Coordinate];

5.3.5 The sound generator

This procedure allows generating simple tones on processors equipped with suitable hardware:

UserTerminal.Beep: PROCEDURE [frequency: CARDINAL ← 1000,
duration: CARDINAL ← 500];

Beep sounds a tone of the given frequency (specified in hertz) for the specified duration, specified in milliseconds. The procedure is synchronous, it does not return until the beep has been generated. A **Beep** may be prematurely terminated using **Process.Abort**.

On the Dandelion, frequencies lower than 29 Hz are rounded up to 29 Hz. The practical upper limit is human audibility. The granularity of the duration is one process timeout tick (about 50 ms.). The specified frequency is actually rounded up to the next frequency which exactly divides 1.8432 MHz.

5.4 Floppy disk channel

FloppyChannel: DEFINITIONS ...

The floppy disk is supported in Pilot in two modes: as a Pilot floppy file system, and as a direct software channel. The two forms of access are mutually exclusive. This section addresses the second form, channel access.

The **FloppyChannel** interface to the floppy disk provides the client direct sector-level access to the floppy disk. This interface allows the client to check and set drive- and diskette-specific characteristics, to check drive status, and to read and write sectors or groups of sectors. Logical formatting of the disk is the responsibility of the client.

Each drive is accessed by a **Handle**.

FloppyChannel.Handle: TYPE [2];

FloppyChannel.nullHandle: READONLY Handle;

FloppyChannel.Error: ERROR [type: FloppyChannel.ErrorType];

FloppyChannel.ErrorType: TYPE = { . . . , invalidHandle, . . . };

For all of the floppy channel operations that take a **Handle** as an argument, the error **FloppyChannel.Error[invalidHandle]** is raised if the **Handle** is not valid. A **Handle** is invalid if the drive that it refers to has changed state (i.e., gone from not-ready to ready or from ready to not-ready) since the **Handle** was acquired.

The following frequently used types are available for the convenience of **FloppyChannel** clients.

FloppyChannel.Density: TYPE = {single, double};

FloppyChannel.Format: TYPE = {IBM, Troy};

FloppyChannel.HeadCount: TYPE = [0..256];

FloppyChannel.SectorCount: TYPE = [0..256];

5.4.1 Drive characteristics

The **Attributes** record contains the characteristics of the specific drive connected to the floppy disk controller and of the diskette currently installed.

FloppyChannel.Attributes: TYPE = RECORD [
deviceType: Device.Type, numberOfCylinders: CARDINAL,numberOfHeads: HeadCount,
maxSectorsPerTrack: SectorCount, formatLength: CARDINAL, ready: BOOLEAN,
diskChange: BOOLEAN, twoSided: BOOLEAN]

deviceType indicates the type of drive connected to the controller; **numberOfCylinders** is the number of cylinders available for recording on that drive; **numberOfHeads** is the number of read/write heads available on that drive; **maxSectorsPerTrack** is the maximum

number of sectors per track of the diskette (based on context setting); **formatLength** is the size of the buffer, in words, needed in order to format the diskette; **ready** indicates whether the drive contains a diskette or not; **diskChange** indicates whether the drive has gone from ready to not-ready (door open), or from not-ready to ready, one or more times since the last operation was performed; and **twoSided** indicates whether the diskette currently installed has data on both sides.

5.4.2 Diskette characteristics

```
FloppyChannel.Context: TYPE = RECORD [protect: BOOLEAN, format: Format,  
density: Density, sectorLength: CARDINAL[0..1024]];
```

The values of **format**, **density**, and **sectorLength** are determined when the diskette is formatted. Software write-protect can be selected by the client software by setting the **protect** flag. The actual write fault status is a logical OR of this variable and the physical signal being returned from the drive. The **Troy** format is the Xerox 850 format. Note that track 00 on IBM format diskettes, and all tracks of Troy format diskettes will be single density. **sectorLength** is the length in words of the sectors on the current track. The value must come from a valid set defined as {64, 128, 256, 512} for IBM format and {1022} for Troy format.

The context must be set, via **SetContext**, before any drive access procedures are called. **GetContext** returns the current context settings.

```
FloppyChannel.GetContext: PROCEDURE [handle: FloppyChannel.Handle]  
RETURNS [context: FloppyChannel.Context];
```

```
FloppyChannel.SetContext: PROCEDURE [handle: FloppyChannel.Handle,  
context: FloppyChannel.Context]  
RETURNS [ok: BOOLEAN];
```

The client must provide the context setting which matches the actual format of the diskette. **SetContext** does not cause the diskette to be reformatted.

5.4.3 Status

The **Status** of the drive and operation is returned by any drive access operation.

```
FloppyChannel.Status: TYPE = MACHINE DEPENDENT{  
goodCompletion, diskChange, notReady, cylinderError,  
deletedData, recordNotFound, headerError, dataError,  
dataLost, writeFault, otherError[LAST[CARDINAL]]};
```

The meanings assigned to the fields in the status record are:

goodCompletion The operation has completed normally.

diskChange The disk drive has apparently gone from a ready to a not ready state (door open), or vice versa, one or more times since the last operation was performed.

notReady	The drive is not ready (does not contain a diskette).
cylinderError	The cylinder specified by the disk address can not be located
deletedData	The record ID for the sector contained a deleted data address mark in the header.
recordNotFound	The record defined by the disk address could not be found.
headerError	A bad checksum was encountered on the header field.
dataError	A bad checksum was encountered on the data field.
dataLost	A sector has been found on the diskette that is larger than that of the current context.
writeFault	Logical OR of the context setting of protect, and the physical signal being returned from the drive.
otherError	An unexpected software or hardware problem has occurred.

5.4.4 Transfer operations

Transfer procedures move the specified number of sectors to or from the diskette. Seek, error recovery, and wait for completion or error are included.

```
FloppyChannel.DiskAddress: TYPE = MACHINE DEPENDENT RECORD [cylinder: CARDINAL,
head: HeadCount, sector: SectorCount];
```

The **cylinder** and **head** fields must reference a valid cylinder and head as defined by the Attributes record. The value of **sector** must be in the range defined by Context.**sectorLength**.

```
FloppyChannel.ReadSectors: PROCEDURE [handle: FloppyChannel.Handle,
address: FloppyChannel.DiskAddress, buffer: LONG POINTER, count: CARDINAL ← 1,
incrementDataPtr: BOOLEAN ← TRUE]
RETURNS [status: FloppyChannel.Status, countDone: CARDINAL];
```

```
FloppyChannel.WriteSectors: PROCEDURE [handle: FloppyChannel.Handle,
address: FloppyChannel.DiskAddress, buffer: LONG POINTER, count: CARDINAL ← 1,
incrementDataPtr: BOOLEAN ← TRUE]
RETURNS [status: FloppyChannel.Status, countDone: CARDINAL];
```

```
FloppyChannel.WriteDeletedSectors: PROCEDURE [handle: FloppyChannel.Handle,
address: DiskAddress, buffer: LONG POINTER, count: CARDINAL ← 1, incrementDataPtr:
BOOLEAN ← TRUE]
RETURNS [status: FloppyChannel.Status, countDone: CARDINAL];
```

```
FloppyChannel.ReadID: PROCEDURE [handle: FloppyChannel.Handle,
    address: FloppyChannel.DiskAddress, buffer: LONG POINTER]
    RETURNS [status: FloppyChannel.Status];
```

The **count** parameter in the above calls indicates the number of sectors to be transferred. The **incrementDataPtr** parameter determines if **buffer** is advanced on multiple sector transfers. If **incrementDataPtr** is **TRUE**, succeeding sectors are read and written advancing through the buffer. If it is **FALSE**, all transfers occur using the same sector buffer. The latter might be used to write the same data into a number of sectors, or to read in order to verify sectors.

WriteSectors and **WriteDeletedSectors** do a read-after-write to verify that the data is readable.

ReadID reads 3 words of device dependent data into the buffer. This operation is provided primarily for diagnostics.

Multiple sector transfers which begin on track 0 of an IBM-formatted diskette and continue on to subsequent tracks will produce an error if the format of the remainder of the diskette is different from the track 0 format (single density, 64-word sectors).

5.4.5 Non-transfer operations

The non-transfer operations access the drive in the same manner as the transfer operations, but no data is moved. **Nop** returns a status. **FormatTracks** formats the specified tracks.

```
FloppyChannel.Nop: PROCEDURE [handle: FloppyChannel.Handle]
    RETURNS [status: FloppyChannel.Status];
```

```
FloppyChannel.FormatTracks: PROCEDURE [handle: FloppyChannel.Handle,
    start: FloppyChannel.DiskAddress, trackCount: CARDINAL]
    RETURNS [status: FloppyChannel.Status, countDone: CARDINAL];
```

Analogous to the **PhysicalVolume** interface, **FloppyChannel** provides the following operations:

```
FloppyChannel.Drive: TYPE = CARDINAL;
```

```
FloppyChannel.GetNextDrive: PROCEDURE [lastDrive: FloppyChannel.Drive]
    RETURNS [nextDrive: FloppyChannel.Drive];
```

```
FloppyChannel.nullDrive: FloppyChannel.Drive = ...;
```

```
FloppyChannel.GetHandle: PROCEDURE [drive: FloppyChannel.Drive]
    RETURNS [handle: FloppyChannel.Handle];
```

```
FloppyChannel.InterpretHandle: PROCEDURE [handle: FloppyChannel.Handle]
    RETURNS [drive: FloppyChannel.Drive];
```

```
FloppyChannel.ErrorType: TYPE = {invalidDrive, ...};
```

GetNextDrive is a stateless enumerator of the floppy drives attached to the system element. It begins with **nullDrive** as an argument and terminates with **nullDrive** as its result. A **Handle** is obtained by calling **GetHandle**. The **Drive** corresponding to a **Handle** may be obtained by calling **InterpretHandle**. **invalidDrive** is raised by **GetHandle** and **GetNextDrive** if they are passed an invalid **Drive**.

5.5 Floppy file system

Floppy: DEFINITIONS . . . ;

Floppy is the interface for the Floppy file system. **Floppy** provides a read/write file system only. Direct mapping of floppy files to Pilot spaces is not supported by **Floppy**. The implementation module is named **FloppyImpl.bcd**.

5.5.1 Accessing files on the diskette

The floppy diskette contains a collection of files. As with Pilot volumes on rigid disks, each file is a sequence of 256-word blocks called pages. A page corresponds to a sector on the diskette. Diskette space management and the directory of extant files is kept in a structure called the *file list*. Under most circumstances, users will not need to manipulate the contents of file lists.

Floppy.FileID: TYPE [2];

Floppy.PageNumber: TYPE = [0.. -- max pages per diskette --];

Floppy.PageCount: TYPE = [0.. -- max pages per diskette --];

Files are identified by values of the type **FileID**. These are uninterpreted 32-bit quantities assigned uniquely within a given floppy diskette. **FileIDs** are *not* unique from one diskette to another. In particular, if a diskette is copied, the new diskette will have the same files with the same **FileIDs** as the old. Although it is the intention of the implementation not to reuse **FileIDs**, they are not guaranteed to be unique in time for a given diskette (i.e., it is possible for a **FileID** to be assigned to a file and later for that file to be deleted and the **FileID** to be subsequently reused).

Note: **PageNumber** and **PageCount** are actually defined as **LONG CARDINAL** since the current version of Mesa does not permit subranges of **LONG CARDINAL**.

In order to access a floppy diskette, the client must specify a handle of type:

Floppy.VolumeHandle: TYPE [2];

Floppy.nullVolumeHandle: READONLY Floppy.VolumeHandle;

Floppy.Error: ERROR [type: Floppy.ErrorType];

Floppy.ErrorType: TYPE = { . . . , invalidVolumeHandle, . . . };

A **VolumeHandle** is assigned when the floppy is opened (using **Floppy.Open**). A **VolumeHandle** becomes invalid if the floppy drive door is opened, or if the drive is closed

and reopened, even if the diskette remains the same. Values of type **VolumeHandle** are not reused within a given instantiation of Pilot (i.e., from one boot to the next).

All of the operations that take a **VolumeHandle** as an argument will raise **Floppy.Error[invalidVolumeHandle]** if presented with an invalid **VolumeHandle**.

A complete specification of a floppy file is given by

```
Floppy.FileHandle: TYPE = RECORD [volume: Floppy.VolumeHandle, file: Floppy.FileID];
```

All operations in this interface are synchronous. That is, they do not return to the client until they are complete. If a diskette is withdrawn between operations, the Pilot floppy file system will not require scavenging (however, the client files may not be well-formed).

In order to access the floppy at all, the volume must be opened.

```
Floppy.Open: PROCEDURE [drive: CARDINAL ← 0] RETURNS [vol: Floppy.VolumeHandle];
```

```
Floppy.ErrorType: TYPE = {..., notReady, noSuchDrive, invalidFormat, needsScavenging,  
invalidVolumeHandle ...};
```

The operation **Open** opens the floppy volume and prepares it for all subsequent operations. The **drive** argument indicates which floppy drive is intended if there is more than one present.

If there is no diskette in the drive or for some other reason the drive is not ready, then the error **Floppy.Error[notReady]** is raised. If **drive** specifies an unknown device then **Floppy.Error[noSuchDrive]** is raised. If the diskette is not formatted according to the standard supported by Pilot floppies, **Floppy.Error[invalidFormat]** is raised. Finally, if Pilot cannot properly read in the file list or if the volume otherwise appears to not be well formed, **Floppy.Error[needsScavenging]** is raised. In any of these cases, the volume is not opened.

```
FloppyExtrasExtras.GetDrive: PROCEDURE [volumeHandle: Floppy.VolumeHandle]  
RETURNS [drive: CARDINAL];
```

GetDrive returns the floppy drive associated with the given **VolumeHandle**. **Floppy.Error[invalidHandle]** may be raised.

```
Floppy.Close: PROCEDURE [vol: Floppy.VolumeHandle];
```

```
Floppy.ErrorType: TYPE = {..., volumeNotOpen, ...};
```

If the user withdraws the diskette from the drive, or for some other reason it becomes not-ready, the next operation on the floppy will implicitly close the volume and will raise **Floppy.Error[volumeNotOpen]**. Alternatively, an open volume may be closed by calling **Close**. **Close**, whether called implicitly or explicitly, merely causes Pilot to forget about the floppy. It does not flush buffers, write out data from its caches or tables, etc. Thus, closing a closed volume is a no-op.

The principal operations on floppy files are to read from or write to them sequences of pages.

```

Floppy.Read, Write: PROCEDURE [file: Floppy.FileHandle, first: Floppy.PageNumber,
count: Floppy.PageCount, vm: LONG POINTER];

Floppy.ErrorType: TYPE = { . . . , fileNotFound, endOfFile, writeInhibited,
hardwareError . . . };

Floppy.DataError: ERROR [file: Floppy.FileHandle, page: Floppy.PageNumber,
vm: LONG POINTER];

```

These two operations are analogous to **Space.CopyIn** and **Space.CopyOut**; i.e., they cause a sequence of pages to be copied to or from the area in virtual memory designated by **vm** (this pointer must point to the beginning of a page). The sequence is selected from the floppy file designated by **file**, starts with the page numbered **first** within that file and continues for **count** pages. Both operations are synchronous; control does not return to the client until the read or write is complete.

The area to or from which data is copied must be in mapped virtual memory, page aligned, and, if necessary, writable; otherwise, an address fault or write protect fault will result. If an attempt is made to read or write beyond the end of the floppy file, the error **Error[endOfFile]** is raised. If the **file** argument does not specify a known file on that floppy diskette, **Error[fileNotFound]** is raised. If an attempt to write to the floppy fails because the write enable sticker has been removed, the error **Error[writeInhibited]** is raised. The error **Error[hardwareError]** is raised if the drive appears to be broken or has temporarily failed in an unexpected manner.

If a read or write error occurs during transmission of the data (and the sector is not already recorded in **badSectorTable**), the signal **DataError** is raised and data transmission stops. This signal is raised after the data transmission occurs. The values returned with this signal indicate the offending file and page number and a pointer to the buffer in virtual memory containing the data read or written. The signal may not be resumed. Instead, the client should decide what to do with the bad data and bad sector, then continue its read or write operation.

```

Floppy.CopyFromPilotFile: PROCEDURE [pilotFile: File.File, floppyFile: Floppy.FileHandle,
firstPilotPage: File.PageNumber, firstFloppyPage: Floppy.PageNumber,
count: Floppy.PageCount ← Floppy.defaultPageCount];

```

```

Floppy.CopyToPilotFile: PROCEDURE [floppyFile: Floppy.FileHandle, pilotFile: File.File,
firstFloppyPage: Floppy.PageNumber, firstPilotPage: File.PageNumber,
count: Floppy.PageCount ← Floppy.defaultPageCount];

```

```

Floppy.defaultPageCount: Floppy.PageNumber = . . . ;

```

```

Floppy.ErrorType: TYPE = { . . . , incompatibleSizes, . . . };

```

These two operations are simple extensions of **Floppy.Read** and **Floppy.Write**. They copy the specified file pages between a floppy disk file and a Pilot file. The specified pages must exist in both files or **Floppy.Error[incompatibleSizes]** will be raised. If **count** is specified as **defaultPageCount** then the entire file is copied starting from the specified page. Any of the errors mentioned above for the **Read** and **Write** functions may also be raised. Both operations are synchronous.

A bad sector on the floppy diskette may be replaced by an alternate sector somewhere else on the diskette by calling the following operation.

```
Floppy.ReplaceBadSector: PROCEDURE [file: Floppy.FileHandle, page: Floppy.PageNumber]
    RETURNS [readError: BOOLEAN];
```

This operation identifies a sector in terms of a page within a file and causes it to be marked bad. An alternate copy of the sector is placed somewhere else on the diskette. Pilot will do its best to copy the information from the bad sector to the alternate one. If data errors occur during this copy, the **readError** result of this procedure is **TRUE**. If, however, Pilot believes that it has made an exact copy, the result is **FALSE**. After this operation completes, the client may overwrite the sector with any data via the operation **Write**. Bad sectors which have been replaced are invisible to client programs except for the performance of Pilot in accessing them (extra disk seeks are required and an access to a sequence of pages must be broken up).

Caution: **ReplaceBadSector** is not implemented in Pilot 11.0.

5.5.2 Snapshotting and replication of the floppy volume

To facilitate easy snapshotting and replicating of floppies, the following procedures have been added.

```
Floppy.PagesForImage: PROCEDURE [floppyDrive: CARDINAL ← 0] RETURNS [File.pageCount];
```

```
Floppy.MakeImage: PROCEDURE [
    floppyDrive: CARDINAL ← 0, imageFile: File.File,
    firstImagePage: File.PageNumber];
```

```
Floppy.CreateFloppyFromImage: PROCEDURE [
    floppyDrive: CARDINAL ← 0, imageFile: File.File,
    firstImagePage: File.PageNumber, reformatFloppy: BOOLEAN,
    floppyDensity: Floppy.Density ← default, floppySides: Floppy.Sides ← default,
    numberOffiles: CARDINAL ← 0, newLabelString: LONG STRING ← NIL];
```

```
Floppy.GetImageAttributes: PROCEDURE [
    imageFile: File.File, firstImagePage: File.PageNumber,
    name: LONG STRING ← NIL]
    RETURNS [
        maxNumberOfFiles: CARDINAL, currentNumberOfFiles: CARDINAL,
        density: Floppy.Density, sides: Floppy.Sides];
```

```
Floppy.ErrorType: TYPE =
{..., fileListLengthTooShort, floppyImageInvalid, floppySpaceTooSmall...};
```

PagesForImage is used to determine the number of pages needed to copy the contents of a floppy to a file.

The client calls **MakeImage** to snapshot a floppy. The call specifies the destination image file and the page of the destination file at which the image should begin.

To create a floppy from an image file, the client calls **CreateFloppyFromImage** specifying the drive to copy to, the image file to copy from, and various other parameters about the

floppy. The **newLabelString** parameter permits changing the floppy's name from that in the image file. If **reformatFloppy** is **TRUE**, the floppy is reformatted. If the **numberOfFiles** is not zero, and the current number of files on the image file is greater than **numberOfFiles**, then **Error[fileListLengthTooShort]** is raised. **Error[floppyImageInvalid]** is raised if the version, seal, or any of the file id's on the image file are invalid. If the size of the image file is greater than the available space on the floppy, **Error[floppySpaceTooSmall]** is raised.

Note: **DataError** may be raised by **.MakeImage** or **.CreateFloppyFromImage** if a read or write error occurs during transmission of the data.

Finally, the interface provides **GetImageAttributes** so that the client can get information about the image stored in an image file.

5.5.3 Managing the floppy volume

The floppy diskette may be reformatted using the following operation. The volume must not be open.

```
Floppy.Format: PROCEDURE [drive: CARDINAL, maxNumberOfFileListEntries: CARDINAL,
    labelString: LONG STRING, density: Floppy.Density ← default,
    sides: Floppy.Sides ← default];
```

```
Floppy.maxCharactersInLabel: CARDINAL = 40;
```

```
Floppy.Density: TYPE = {single, double, default};
```

```
Floppy.Sides: TYPE = {one, two, default};
```

```
Floppy.ErrorType: TYPE = {..., onlySingleDensity, onlyOneSide, badDisk, ...};
```

```
Floppy.AlreadyFormatted: SIGNAL [labelString: LONG STRING];
```

This operation erases the diskette, writes all information according to the standard supported by Pilot, and creates an empty file list large enough to hold the number of entries specified. A label string is also written on the diskette, in the same way as label strings are written on Pilot rigid disk volumes. If the floppy is already formatted to be a Pilot floppy volume, the resumable signal **AlreadyFormatted** is raised. This gives the client a last chance to recover from accidentally formatting an already valuable floppy. The **density** and **sides** arguments give the client optional control over these attributes of the diskette when necessary for information interchange. The errors **Error[onlySingleDensity]** and **Error[onlyOneSide]** are raised if either the diskette or the drive imposes these limitations. The defaults of these cause Pilot to choose appropriate values for the drive and the diskette. If the disk cannot be reformatted due to problems with either the diskette or the drive, **Error[badDisk]** is raised.

```
Floppy.GetAttributes: PROCEDURE [volume: Floppy.VolumeHandle,
    labelString: LONG STRING]
    RETURNS [freeSpace, largestBlock:Floppy.PageCount, fileList, rootFile: Floppy.FileHandle,
    density: Floppy.Density, sides: Floppy.Sides, maxFileListEntries: CARDINAL];
```

```
Floppy.ErrorType: TYPE = {..., stringTooShort, ...};
```

This operation gets relevant attributes about a floppy volume. The value of the label string is stored in the **labelString** argument (except that a **NIL** argument causes this to be bypassed, rather than raising an error). Other attributes are returned in the result list. The result **freeSpace** indicates the total number of free pages on the diskette, while the result **largestBlock** indicates the largest file that could be created without having to compact the diskette (see below). The **density** and **sides** attributes describe the diskette, independently of what the drive can support. The **rootFile** is a distinguished file identified by the client (see below). The **fileList** attribute describes the file list maintained by Pilot on the diskette. It is returned for completeness only; *clients are strongly discouraged from using it*. The **max fileListEntries** attribute describes the length of the list. It is fixed at format time and does not change over the life of a floppy file system instance.

A file may be created on the diskette with the following operation.

```
Floppy.CreateFile: PROCEDURE [volume: Floppy.VolumeHandle, size:Floppy.PageCount,  
    fileType: File.Type]  
    RETURNS [file: Floppy.FileHandle];
```

```
Floppy.ErrorType: TYPE = { . . . , insufficientSpace, zeroSizeFile, fileListFull . . . };
```

This operation creates a file of the specified size on the diskette. As with files on the Pilot rigid disk, each file is created with a **File.Type** to allow the client program to distinguish what kind of file it is. All files are allocated contiguously on the diskette. If there is no block of free space large enough, **Error[insufficientSpace]** is raised. An attempt to create a zero-sized file fails with the error **Error[zeroSizeFile]**. If the file list is full, the file is not created and **Error[fileListFull]** is raised. This operation, including the updating of the file list, is synchronous and does not return to the client until the file is created and the diskette is well-formed in the new state.

```
Floppy.DeleteFile: PROCEDURE [file: Floppy.FileHandle];
```

This operation deletes the specified file and makes the space available for other files. This operation, including the updating of the file list, is synchronous and does not return to the client until the file is deleted and the diskette is well-formed in the new state.

```
Floppy.GetFileAttributes: PROCEDURE [file: Floppy.FileHandle]  
    RETURNS [size: Floppy.PageCount, type: File.Type];
```

This operation gets the attributes of a file.

```
Floppy.GetNextFile: PROCEDURE [previousFile: Floppy.FileHandle]  
    RETURNS [nextFile: Floppy.FileHandle];
```

```
Floppy.nullFileID: Floppy.FileID = . . . ;
```

This operation enumerates the files on a floppy volume in the standard style of a Pilot stateless enumerator. Files are enumerated in the order that they occur on the diskette. The enumeration is started by supplying the **nullFileID** and the appropriate volume and it ends with the same value. The file list is not included in this enumeration.

```
Floppy.SetRootFile: PROCEDURE [file: Floppy.FileHandle];
```

This operation allows the client to record the **FileID** of a file in the volume data structures for later use. This might be the pointer to a client level directory or to some other data structure. If the file does not exist, **Error[fileNotFound]** is raised.

Floppy.Compact: PROCEDURE [volume: Floppy.Volume];

This operation rearranges the files on the diskette so that all of the free space occurs in one block at the end of the volume. This is necessary to recover fragmented space in those (rare) cases where a lot of file creation and deletion occurs.

Caution: **Compact** is not implemented in Pilot 11.0.

Floppy.Scavenge: PROCEDURE [volume: Floppy.Volume]

RETURNS [numberOfBadSectors: Floppy.PageCount];

Floppy.GetNextBadSector: PROCEDURE [volume: Floppy.VolumeHandle, oldIndex: CARDINAL]

RETURNS [newIndex: CARDINAL, file: Floppy.FileHandle, page: Floppy.PageNumber];

The operation **Scavenge** recovers the contents of a malformed floppy by restoring the file list, repairing bad marker pages, and recovering other data specified by the Pilot floppy standard. **Scavenge** returns the number of new bad pages it encountered in client files while scavenging (others can be handled by Pilot automatically). The operation **GetNextBadSector** allows the client to enumerate the new bad sectors, starting and ending with an index of zero.

Caution: **Scavenge** and **GetNextBadSector** are not implemented in Pilot 11.0.

FloppyExtras.Erase: PROCEDURE [

drive: CARDINAL, maxNumberOfFileListEntries: CARDINAL,

labelString: LONG STRING ← NIL];

FloppyExtras.ExtrasErrorType: TYPE = {..., notFormatted, ...};

The operation **Erase** resets all the floppy file system data structures, writes a new clean file list, re-marks bad pages, and resets all file and microcode pointers. It does not erase any data sectors (only **Format** will actually erase all sectors on the diskette). If a **labelString** is specified, it replaces the current label; otherwise the current label remains unchanged. The volume will be closed if it is open. If **drive** does not describe a drive currently in the system, **Floppy.Error[noSuchDrive]** is raised. If the length of **labelString** exceeds **Floppy.maxCharactersInLabel**, the label will be truncated to the maximum length. **Floppy.Error[badDisk]** is raised if the disk cannot be accessed. **Floppy.Error[notReady]** is raised if there is no diskette in the drive or the drive is not ready. If the diskette is write protected, **Floppy.Error[writeInhibited]** is raised. **FloppyExtras.ExtrasError[notFormatted]** is raised if the diskette has invalid formatting information.

FloppyExtras.NewScavenge: PUBLIC PROCEDURE [drive: CARDINAL]

RETURNS [okay: BOOLEAN];

FloppyExtras.ExtrasErrorType: TYPE = {..., volumeOpen, ...};

The operation **NewScavenge** recovers the contents of a malformed floppy by restoring the file list, repairing bad marker pages, and recovering other data specified by the Pilot floppy standard. The volume must not be open. The return value **okay** indicates whether

the scavenge was successful: if **okay** returns **TRUE**, the floppy was, or was made, consistent. If **drive** does not describe a drive currently in the system, **Floppy.Error[noSuchDrive]** is raised. If the diskette is write protected, **Floppy.Error[writeInhibited]** is raised. **FloppyExtras.ExtrasError[volumeOpen]** is raised if the floppy volume on the diskette is open. Other **Floppy.Errors** which result from reading or writing the floppy may also be raised.

Note: The 12.0 floppy scavenger does not repair damage. After validating the file system and internal data structures, it resets the "needs-scavenging" indicator if the floppy is consistent.

Note: In a future release, **FloppyExtras** and **FloppyExtras.Extras** will be merged into **Floppy**. At that time, the names of some interface items may change.

Several special operations are necessary to support Pilot-bootable floppies:

```
Floppy.CreateInitialMicrocodeFile: PROCEDURE [volume: Floppy.VolumeHandle,  
size: Floppy.pageCount, type: File.Type,  
startingPageNumber: Floppy.PageNumber ← 1]  
RETURNS [file: Floppy.FileHandle];
```

```
Floppy.ErrorType: TYPE = { . . . , initialMicrocodeSpaceNotAvailable, badSectors, . . . };
```

This operation is like **CreateFile** except that it creates the initial microcode file at the exact location demanded by the hardware boot facility. In particular, the page of the file numbered **startingPageNumber** will appear where the hardware expects to read the first block from the floppy diskette at boot time. The hardware of our current machines demands that the initial microcode file must be contiguous and contain no bad sectors. Thus, **CreateInitialMicrocodeFile** should normally be applied only to a clean, newly formatted diskette. If it is not possible to create such a file, either because there already is a file there or because some sector is bad, **Floppy.Error[initialMicrocodeSpaceNotAvailable]** or **Floppy.Error[badSectors]** are raised.

```
Floppy.nullBootFilePointer: Floppy.BootFilePointer = [nullFileID, 0];
```

```
Floppy.SetBootFiles: PROCEDURE [vol: Floppy.VolumeHandle,  
pilotMicrocode, diagnosticMicrocode, germ,  
pilotBootFile: Floppy.BootFilePointer ← Floppy.nullBootFilePointer];
```

```
Floppy.GetBootFiles: PROCEDURE [volume: Floppy.VolumeHandle]  
RETURNS [initialMicrocode, pilotMicrocode, diagnosticMicrocode, germ,  
pilotBootFile: Floppy.BootFilePointer];
```

```
Floppy.BootFilePointer: TYPE = RECORD [Floppy.FileID, page: Floppy.pageNumber];
```

```
Floppy.ErrorType: TYPE = { . . . , invalidPageNumber, . . . };
```

The operation **SetBootFiles** sets the pointers to the relevant boot files in the volume data structures. This track is read by the initial microcode at boot time in order to properly initialize the microcode and Pilot. Both a **FileID** and a page number are specified so that leader pages may be included in floppy boot files if desired. **SetBootFiles** will set the pointer in track zero for any of its arguments with a non-null **FileID**. Boot file pointers with

nullFileID are cleared. This operation is synchronous. If the specified file page(s) do not exist, the error **Error[invalidPageNumber]** is raised.

The remaining boot files on the diskette, apart from the initial microcode boot file, are all read by the initial microcode file. Thus, they can be located anywhere and can have bad sectors in them, and the initial microcode can interpret the bad sector table if necessary.

The operation **GetBootFiles** gets the pointers to all of the boot files, including the initial microcode boot file.

It is recommended that clients assign distinguished Pilot file types to boot files to allow the boot file pointers to be reset, if necessary, after scavenging.

Note: Booting in the manner described here is not supported by Pilot 11.0. Clients must use **MakeDLionBootFloppy** tool to create bootable floppies in Pilot 11.0.

5.6 TTY Port channel

TTYPort: DEFINITIONS . . . ;

TTYPortEnvironment: DEFINITIONS . . . ;

The TTY Port channel is a Product Common Software package which provides a Pilot client with access to the TTY Port controller and the connected device. It contains procedures for sending and receiving bytes to and from the device, and for receiving status back. Examples of devices that use the TTY Port include the Diablo 630 character printer and the Lear Siegler ADM-3 display terminal. The **TTYPort** interface is implemented by **TTYPortChannel.bcd**.

The Diablo 630 character printer is an ASCII output device containing a daisy wheel printer of the HyType II genre. The Lear Siegler ADM-3 display terminal is an ASCII I/O device of the "glass teletype" type.

5.6.1 Creating and deleting the TTY Port channel

**TTYPort.Create: PROCEDURE [lineNumber: CARDINAL]
RETURNS [TTYPort.ChannelHandle];**

TTYPort.ChannelHandle: PRIVATE . . . ;

TTYPort.nullChannelHandle: TTYPort.ChannelHandle . . . ;

TTYPort.ChannelAlreadyExists: ERROR;

TTYPort.NoTTYPortHardware: ERROR;

TTYPort.InvalidLineNumber: ERROR;

Create creates the channel to the TTY Port. If the channel already exists, **Create** generates the error **TTYPort.ChannelAlreadyExists**. If no TTY Port hardware is installed, **Create** generates the error **TTYPort.NoTTYPortHardware**. If **lineNumber** does not represent

a line present on the TTY Port controller, **Create** generates the error **TTYPort.InvalidLineNumber**.

TTYPort.Delete: PROCEDURE [channel: TTYPort.ChannelHandle];

Delete deletes the channel and releases the associated device. This operation has the effect of calling **Quiesce**, aborting all pending activity on the channel. Any uncompleted **Gets** or **Puts** will be terminated with **status = abortedByDelete**.

TTYPort.Quiesce: PROCEDURE [channel: TTYPort.ChannelHandle];

Quiesce aborts all pending activity. All uncompleted asynchronous activities (i.e., those initiated by **Get** or **Put**) will be terminated with **status** equal to **aborted**. Any additional operations on the channel, other than **Delete**, cause the error **ChannelQuiesced**.

5.6.2 Data transfer

TTYPort.Put: PROCEDURE [channel: TTYPort.ChannelHandle, data: CHARACTER]
RETURNS [status: TTYPort.TransferStatus];

Put transmits **data** to the TTY Port. **status** will be set to **success** if the character is successfully transmitted. Aborts are disabled for this operation.

TTYPort.Get: PROCEDURE [channel: TTYPort.ChannelHandle]
RETURNS [data: CHARACTER, status: TTYPort.TransferStatus];

TTYPort.Get waits until a byte of data is received from the TTY Port. **status** equals **success** if a character is successfully received. Aborts are disabled for this operation.

The procedure

TTYPort.SendBreak: PROCEDURE [channel: TTYPort.ChannelHandle];

causes a break to be sent on the specified TTY channel.

5.6.3 Data transfer status

The status of an individual data transfer (i.e., **Get** or **Put**) is indicated by a variable of type **TransferStatus**.

TTYPort.TransferStatus: TYPE = {**success**, **parityError**, **asynchFramingError**, **dataLost**,
breakDetected, **aborted**, **abortedByDelete**};

The meanings of these status codes are:

success	Normal completion.
parityError	Data has not been transferred faithfully.
asynchFramingError	Data has not been transferred faithfully (i.e., stop bits were missing).

dataLost	Data has been lost due to lack of any data buffers to hold received characters.
breakDetected	A break has occurred on the line. This bit is latched and can be cleared using the SetParameter operation (see below).
aborted	TTYPort.Quiesce has been called while the transfer is outstanding.
abortedByDelete	TTYPort.Delete has been called while the transfer is outstanding.

5.6.4 TTY Port operations

The TTY Port Channel will buffer up to 16 characters of input from its device along with their associated transfer status. To see if and how much data has been received from the device by the TTY Port, call the procedure

```
TTYPort.ChrsAvailable: PROCEDURE [channel: TTYPort.ChannelHandle]
    RETURNS [number: CARDINAL];
```

number indicates the number of input buffers containing data.

The various parameters associated with a TTY port are set with the procedure

```
TTYPort.SetParameter: PROCEDURE [channel: TTYPort.ChannelHandle,
    parameter: TTYPort.Parameter];
```

The parameters are contained in records of the following type:

```
TTYPort.Parameter: TYPE = RECORD [SELECT parameter: * FROM
    breakDetectedClear = > [breakDetectedClear: BOOLEAN],
    characterLength = > [characterLength: TTYPort.CharacterLength],
    clearToSend = > [clearToSend: BOOLEAN],
    dataSetReady = > [dataSetReady: BOOLEAN],
    lineSpeed = > [lineSpeed: TTYPort.LineSpeed],
    parity = > [parity: TTYPort.Parity],
    stopBits = > [stopBits: TTYPort.StopBits],
    ENDCASE];
```

```
TTYPort.CharacterLength: TYPE = TTYPortEnvironment.CharacterLength;
```

```
TTYPort.LineSpeed: TYPE = TTYPortEnvironment.LineSpeed;
```

```
TTYPort.Parity: TYPE = TTYPortEnvironment.Parity;
```

```
TTYPort.StopBits: TYPE = TTYPortEnvironment.StopBits;
```

```
TTYPortEnvironment.LineSpeed: TYPE = {bps50, bps75, bps110, bps134p5, bps150, bps300,
    bps600, bps1200, bps1800, bps2000, bps2400, bps3600, bps4800, bps7200, bps9600,
    bps19200};
```

TTYPortEnvironment.Parity: TYPE = {none, odd, even};

TTYPortEnvironment.CharacterLength: TYPE = {lengthIs5bits, lengthIs6bits, lengthIs7bits, lengthIs8bits};

TTYPortEnvironment.StopBits: TYPE = {none, one, oneAndHalf, two};

breakDetectedClear is used to clear the *latch bit* **breakDetected** in **TTYPort.DeviceStatus**.

CharacterLength selects the character length and is defaulted to **lengthIs8bits**.

The boolean **clearToSend** governs the state of the corresponding circuit to the TTY Port. It is defaulted to **FALSE**. After the TTY Port channel is created, **clearToSend** should remain **TRUE** at all times since the communication line is full-duplex.

The boolean **dataSetReady** governs the state of the corresponding circuit to the TTY Port. It is defaulted to **FALSE**. **dataSetReady** should be set **TRUE** when the communication line is to be connected, **FALSE** when it is to be disconnected.

TTYPort.LineSpeed selects the timer constant for the baud rate generator which provides the clocking for transmissions to and from the TTY Port. **bps1200** is the default.

TTYPort.Parity selects the parity of the transmissions. **none** is the default.

TTYPort.StopBits is the number of stop bits. **two** is the default.

5.6.5 Device status

In addition to the status information returned for each data transfer operation, state information about the TTY Port itself is kept in the **DeviceStatus** record. It is accessed via the **GetStatus** procedure.

TTYPort.GetStatus: PROCEDURE [channel: TTYPort.ChannelHandle]
RETURNS [stat: TTYPort.DeviceStatus];

The procedure

TTYPort.StatusWait: PROCEDURE [channel: TTYPort.ChannelHandle,
stat: TTYPort.DeviceStatus]
RETURNS [newstat: TTYPort.DeviceStatus];

waits until the current **DeviceStatus** differs from the supplied parameter **stat**. The client must examine **newstat** to determine what action to take.

TTYPort.DeviceStatus: TYPE = RECORD [aborted, breakDetected, dataTerminalReady,
readyToGet, readyToPut, requestToSend: BOOLEAN];

The boolean **aborted** indicates that the **TTYPort.StatusWait** was aborted by either a **TTYPort.Delete** or **TTYPort.Quietce**.

The boolean **breakDetected** indicates that a "break" was received on the communication line, where "break" is defined to be the absence of a "stop" bit for more than 190 milliseconds. This boolean is called a *latch bit* in that it is set by the channel when the

associated condition occurs, but is not cleared by the channel when the condition clears. It remains set to guarantee that the client has an opportunity to observe it. To clear it (in order to detect its subsequent setting), **breakDetectedClear** is specified as a parameter to the **TTYPort.SetParameter** procedure.

The boolean **dataTerminalReady** is **TRUE** when the associated device is powered on.

The boolean **readyToGet** is **TRUE** when the hardware input buffer (for data sent from the device) is not empty.

The boolean **readyToPut** is **TRUE** when the hardware output buffer (for data sent to the device) is not full.

The boolean **requestToSend** is held **TRUE** by the device (as in a 103-type modem) to enable transmission to the device.

5.7 TTY Input/Output

TTY: DEFINITIONS . . . ;

The **TTY** interface provides a simple character-oriented input and output facility. It admits many implementations on character-oriented terminal devices. In this way it is a lot like the **Stream** interface. This interface is Product Common Software.

Note: For most clients, the default TTY implementation will be supplied as part of this release: **TTYLearSiegle.bcd** or that provided by the Mesa Development Environment.

Note: The Lear Siegler TTY implementation has the following default settings for the TTY Port channel: 8 bit characters, 9600 baud, 2 stop bits, no parity, CTS set to ON, and DSR set to ON.

5.7.1 Starting and stopping

```

TTY.Create: PROCEDURE [name: LONG STRING ← NIL,
                     backingStream, ttyImpl: Stream.Handle ← NIL]
                     RETURNS [h: TTY.Handle];

TTY.Handle: TYPE [2];

TTY.nullHandle: TTY.Handle = LOOPHOLE[LAST[LONG CARDINAL]];

TTY.NoDefaultInstance: ERROR;

TTY.OutOfInstances: ERROR;

```

Create creates a **Handle**, which is returned to the caller. This handle is then passed as an argument to the other TTY input/output operations. The arguments **name** and **backingStream** are used by the underlying TTY implementation in an implementation-dependent fashion to implement the backing file for the TTY. If **ttyImpl** is not **NIL**, it is used as the stream implementing the TTY stream. If **ttyImpl** is **NIL**, an instance of the default TTY implementation is created. The parameter **h** is the **TTY.Handle** that will correspond to the stream underlying this TTY channel when the call to **TTY.Create**

completes. If there is no default TTY implementation, the error **NoDefaultInstance** is raised. If another **Handle** cannot be created, **OutOfInstances** is raised.

TTY.SetBackingSize: PROCEDURE [h: TTY.Handle, size: LONG CARDINAL];

This procedure sets an upper limit on the number of bytes in the backing file and forces the backing file to be used in a wraparound mode. It has no effect if the implementation does not support a backing file.

TTY.Destroy: PROCEDURE [h: TTY.Handle, deleteBackingFile: BOOLEAN ← FALSE];

Destroy invalidates **TTY.Handle**. If **deleteBackingFile** is **TRUE** and the backing file was created by **Create** then the backing file is deleted.

TTY.UserAbort: PROCEDURE[h: TTY.Handle] RETURNS [yes: BOOLEAN];

TTY.ResetUserAbort: PROCEDURE[h: TTY.Handle];

TTY.SetUserAbort: PROCEDURE[h: TTY.Handle];

UserAbort returns the value of the user abort flag. **TRUE** indicates that that user has typed some "abort" key. **TTY.ResetUserAbort** clears the user abort flag. **TTY.SetUserAbort** sets the user abort flag, just as if the user had typed the "abort" key.

Note: The Lear Siegler TTY implementation allows users to abort processes by depressing the Break key or by depressing the Control and Stop keys simultaneously.

5.7.2 Signals and errors

The signal

TTY.LineOverflow: SIGNAL [s: LONG STRING] RETURNS [ns: LONG STRING];

indicates that input has filled the string **s**. The current contents of the string are passed as a parameter. The catch phrase should return a string **ns** with more room.

The signal

TTY.Rubout: SIGNAL;

indicates that the DEL key was typed during **TTY.GetEditedString** (or procedures which call **GetEditedString**).

5.7.3 Output

To output a block of characters call

TTY.PutBlock: PROCEDURE[h: TTY.Handle, block: Environment.Block];

5.7.4 Utilities

TTY.BackingStream: PROCEDURE [h: TTY.Handle] RETURNS [stream: Stream.Handle];

TTY.NoBackingFile: ERROR;

If a backing stream was created by **TTY.Create**, this operation returns the **Stream.Handle** for it. If none was created, the error **TTY.NoBackingFile** is raised.

TTY.ChrsAvailable: PROCEDURE [h: TTY.Handle] RETURNS [number: CARDINAL];

CharsAvailable returns the number of input characters available (but not yet delivered to the client).

TTY.NewLine: PROCEDURE [h: TTY.Handle] RETURNS [yes: BOOLEAN];

NewLine returns **TRUE** when at the beginning of an output line. This procedure is mainly used when formatting output.

TTY.PutBackChar: PROCEDURE [h: TTY.Handle, c: CHARACTER];

PutBackChar places **c** at the front of the list of characters to be input to the client.

TTY.SetEcho: PROCEDURE [h: TTY.Handle, new: TTY.EchoClass]
RETURNS [old: TTY.EchoClass];

TTY.GetEcho: PROCEDURE [h: TTY.Handle] RETURNS [old: TTY.EchoClass];

TTY.EchoClass: TYPE = {none, plain, stars};

SetEcho sets how input characters are to be echoed back to the output. It returns the *previous* state of the echoing mode. If the mode is **none**, no characters are echoed; if it is **stars**, the character "*" is echoed for each input character. The default echoing mode is **plain**. Automatic echoing is done only for the procedure **TTY.GetEditedString** and the procedures implemented using **TTY.GetEditedString**.

TTY.BlinkDisplay: PROCEDURE [h: TTY.Handle];

This procedure causes the display to be blinked if the device is capable of it.

TTY.PushAlternateInputStream: PROCEDURE [h: TTY.Handle, stream: Stream.Handle];

TTY.PopAlternateInputStreams: PROCEDURE [h: TTY.Handle, howMany: CARDINAL←1];

PushAlternateInputStream adds an alternate input stream to the **Handle**. Characters will be taken from the most recently pushed alternate input stream until it is exhausted, at which point characters will be taken from the previous input stream. **PopAlternateInputStreams** removes **howMany** alternate input streams from the **Handle**.

If **howmany** is greater than the number of existing alternate input streams, all existing are removed before **PopAlternateInputStream** returns.

5.7.5 String input operations

The operation

TTY.GetChar: PROCEDURE [h: TTY.Handle] RETURNS [c: CHARACTER];

returns the next character of input when it becomes available.

TTY.CharStatus: TYPE = {ok, stop, ignore};

TTY.GetEditedString: PROCEDURE [h: TTY.Handle, s: LONG STRING,
t: PROCEDURE [c: CHARACTER] RETURNS [status: TTY.CharStatus]]
RETURNS [c: CHARACTER];

GetEditedString appends input character(s) to the string **s**. The user-supplied procedure **t** determines which character terminates the string. If **t** returns **stop**, the character **c** passed to it should terminate the string. If **t** returns **ok**, the character **c** should be appended to the string. If **t** returns **ignore**, the character **c** should not be appended to the string, but the string should not yet be terminated. Note that the client must initialize **s.length**, typically to zero. The signal **TTY.LineOverflow** is raised if **s maxlen** is reached. The following special characters are recognized on input, and are not appended to **s**:

	DEL	- raises the signal TTY.Rubout
50H, BS	↑ A, ↑ H (backspace)	- delete the last character
ETB, DC1	↑ W, ↑ Q (backward)	- delete the last word
CAN	↑ X	- delete everything
DC2	↑ R	- retype the line
SYN	↑ V	- quote the next character, used to input special characters

Echoing of characters other than the special characters and the terminating character is determined by the echoing mode set by **TTY.SetEcho** (default is **plain**). The returned character **c** is the character which terminated the string. **c** is not echoed nor included in the string.

The following three string input procedures use **TTY.GetEditedString** to read a string.

TTY.GetString: PROCEDURE [h: TTY.Handle, s: LONG STRING,
t: PROCEDURE [c: CHARACTER] RETURNS [status: TTY.CharStatus]];

TTY.GetID: PROCEDURE [h: TTY.Handle, s: LONG STRING];

TTY.GetLine: PROCEDURE [h: TTY.Handle, s: LONG STRING];

GetString reads a string into **s**. The user-supplied procedure **t** determines which character terminates the string. If **t** returns **stop**, the character **c** passed to it terminates the string. If **t** returns **ok**, the character **c** will be appended to the string. If **t** returns **ignore**, the character **c** will not be appended to the string, but the string will not yet be terminated. The terminating character (the character returned by **TTY.GetEditedString**) is echoed regardless of the echoing mode.

GetID reads a string terminated with a space or a carriage return into **s**. The terminating character (space or carriage return) is not echoed regardless of the echoing mode.

GetLine reads a string terminated with a carriage return into **s**. The carriage return is *not* appended to **s**. A carriage return is output regardless of the echoing mode.

The procedure

TTY.GetPassword: PROCEDURE [h: TTY.Handle, s: LONG STRING];

calls **GetEditedString** with echoing set to **stars**, then restores the previous echoing mode.

5.7.6 String output operations

The operation

TTY.PutChar: PROCEDURE [h: TTY.Handle, c: CHARACTER];

outputs the character **c**. If **c** is a carriage return, the next character that is output will be in the first position of the next line. Note that control characters other than a carriage return being output are not interpreted by **PutChar**, but rather translated into a two character printable sequence (e.g., ↑ A). If **c** is **Ascii.BS**, a representation of the backspace will be displayed in the window. To backspace over previously output characters, see **RemoveCharacter** below.

TTY.PutCR: PROCEDURE [h: TTY.Handle];

PutCR outputs a carriage return. The next character that is output will be in the first position of the next line.

TTY.PutBlank, PutBlanks: PROCEDURE [h: TTY.Handle, n: CARDINAL ← 1];

PutBlank(s) outputs **n** spaces.

TTY.PutDate: PROCEDURE [h: TTY.Handle, gmt: Time.Packed,
format: TTY.DateFormat ← noSeconds];

TTY.DateFormat: TYPE = Format.DateFormat;

Format.DateFormat: TYPE = {dateOnly, noSeconds, dateTime, full, mailDate};

PutDate outputs the Greenwich mean time, packed in the **Time** format, according to the format specified.

The different formats have the following interpretation:

maildate:	27 Jul 83 09:23:29 PDT (Wednesday)
full:	27-Jul-83 9:23:29 PDT
dateTime:	27-Jul-83 9:23:29
noSeconds:	27-Jul-83 9:23
dateOnly:	27-Jul-83

TTY.PutString, PutText: PROCEDURE [h: TTY.Handle, s: LONG STRING];

TTY.PutLine: PROCEDURE [h: TTY.Handle, s: LONG STRING];

TTY.PutSubString, PutLongSubString: PROCEDURE [h: TTY.Handle,
ss: String.SubString];

PutString outputs the string **s**. Whenever a carriage return is output, the next character that is output will be in the first position of the next line. **PutLine** outputs the string **s** followed by a carriage return. The other procedures output their string parameter.

The procedures

TTY.RemoveCharacter, RemoveCharacters: PROCEDURE [h: TTY.Handle,
n: CARDINAL ← 1];

backspaces over the last **n** characters output, erasing the characters from the display. In implementations lacking an actual hardware backspace facility, this is often simulated by outputting the backed-over text surrounded by backslashes.

5.7.7 Numeric input operations

The following six numeric input procedures use **TTY.GetEditedString** to read a string terminated with a space or a carriage return. The terminating character is not echoed (regardless of the echoing mode). An implementation of **TTY** might use the numeric conversion facilities offered by the **String** interface. If it did, it would raise **String.InvalidNumber** when presented with an input string that did not conform to the syntax for a number.

TTY.GetNumber: PROCEDURE [h: TTY.Handle, default: UNSPECIFIED,
radix: CARDINAL, showDefault: BOOLEAN]
RETURNS [n: UNSPECIFIED];

TTY.GetLongNumber: PROCEDURE [h: TTY.Handle, default: LONG UNSPECIFIED, radix: CARDINAL,
showDefault: BOOLEAN]
RETURNS [n: LONG UNSPECIFIED];

These operations read in a string and convert it to base **radix**. If an **ESC** is the first character typed and **showDefault** is **TRUE**, a string representing the value of **default** converted to base **radix** is displayed. If **radix** is 10 and **default** is negative, a minus sign will be prefixed, or if **radix** is 8, the character B will be postfixed.

```
TTY.GetOctal: PROCEDURE [h: TTY.Handle] RETURNS [n: UNSPECIFIED];
TTY.GetLongOctal: PROCEDURE [h: TTY.Handle] RETURNS [n: LONG UNSPECIFIED];
TTY.GetDecimal: PROCEDURE [h: TTY.Handle] RETURNS [n: INTEGER];
TTY.GetLongDecimal: PROCEDURE [h: TTY.Handle] RETURNS [n: LONG INTEGER];
```

GetOctal and **GetLongOctal** read in a string then convert it to octal. **GetDecimal** and **GetLongDecimal** read in a string then convert it to decimal.

5.7.8 Numeric output operations

```
TTY.PutNumber: PROCEDURE [ h: TTY.Handle, n: UNSPECIFIED,
                           format: TTY.NumberFormat];
TTY.PutLongNumber: PROCEDURE [ h: TTY.Handle, n: LONG UNSPECIFIED,
                                format: TTY.NumberFormat];
TTY.NumberFormat: TYPE = Format.NumberFormat;
Format.NumberFormat: TYPE = RECORD [base: [2..36] ← 10, zerofill: BOOLEAN ← FALSE,
                                    unsigned: BOOLEAN ← TRUE, columns: [0..255] ← 0 ];
```

PutNumber and **PutLongNumber** convert **n** to a string representing its value according to the format specified, and then output the string. **NumberFormat** refers to a number whose base is **base**. The field is **columns** wide (if **columns** is 0, it means use as many as needed). If **zerofill** is **TRUE**, the extra columns are filled with zeros, otherwise spaces are used. If **unsigned** is **TRUE**, the number is treated as unsigned. Output strings representing negative numbers begin with a minus sign.

```
TTY.PutOctal: PROCEDURE [h: TTY.Handle, n: UNSPECIFIED];
TTY.PutLongOctal: PROCEDURE [h: TTY.Handle, n: LONG UNSPECIFIED];
TTY.PutDecimal: PROCEDURE [h: TTY.Handle, n: INTEGER];
TTY.PutLongDecimal: PROCEDURE [h: TTY.Handle, n: LONG INTEGER];
```

PutOctal and **PutLongOctal** convert **n** to a string representing the octal value (when **n** is greater than 7, the character B is appended), and then output the string. **PutDecimal** and **PutLongDecimal** convert **n** to a string representing the signed decimal value, and then output the string.



Communication

The communication package provides Pilot clients the facility to perform *inter-* and *intra-*processor communication at a relatively high level. The structure of Pilot communications is layered. That layering follows closely the protocol levels specified in **Internet Transport Protocols, XSIS 028112**, dated December, 1981 (XNS).

Only the lowest level protocol layer, level 0, is medium dependent. The only medium supported by Pilot communications is the ethernet. Level 0 does provide the framework that permits Pilot clients to implement other level 0 drivers. It is assumed that all level 0 drivers will provide at least the following features: immediate destination addressing, data checking (CRC, LRC, etc), the ability to transmit any 8-bit data pattern, and a means of detecting physical message length.

The level 1 communication layer, known as the *Internet Datagram Protocol* (IDP), is medium independent. Access to this layer is via *sockets*. A *socket* is a logical input/output resource modeled after the Pilot software channel. A socket is an address within a machine, identified by a 16-bit number, to which *NS packets* (henceforth referred to as *packets*) can be delivered and from which packets may be transmitted. Any number of unique addresses may coexist in the same machine.

The socket facility enables reception and transmission of packets per the conventions of IDP. At this level packets are delivered with only some high probability. Packets may arrive out of order, may be duplicated, or may never arrive. The socket facility is used internally in the implementation of higher-level communication facilities, and is not itself available to Pilot clients.

Packets may be transmitted or received over one of the ethernet local networks connected to the machine, or over *any* other communication media that is part of the NS communication system. Packets have an advisable *maximum internetwork length* of 576 bytes in order to be forwardable by internetwork routers.

The full source or destination address of packets is a **System.NetworkAddress**. Addresses are the concatenation of the host's *network number* (**System.NetworkNumber**), the *host number* (**System.HostNumber**) and a socket number (**System.SocketNumber**). Source addresses include an internally generated unique socket. Initial contact with remote machines requires knowing the full address of that machine. The network and host numbers are usually obtained from a central name to address translation facility

(*clearinghouse*) and the socket is *well known*. Socket numbers in the range [0..2048] are reserved for well known sockets.

Communication over the ethernet local network, or any communication network, is different from most other devices since the network may deliver an unsolicited packet which is destined for a socket. Such packets typically consume communication buffers, which are a critical resource. If the arrival rate of packets is high, the client is advised to perform a sufficient number of receive operations to provide adequate buffering. Incoming packets will never be queued for a particular socket if that socket does not exist.

The sections on **PacketExchange** and **NetworkStream** describe interfaces to higher-level, more reliable protocols. The implementations of these interfaces are clients of the socket facility. These two interfaces supply the facilities to be used for NS communication applications. These two implementations make use of the *error protocol* which is not directly accessible to Pilot clients but is alluded to in some of the signal status codes. They also use the *routing protocol*. Client access to routing is described in the section on **Router**.

6.1 Well known sockets

NSConstants: **DEFINITIONS** = ...;

As mentioned, a portion of the socket number name space is reserved for use as *well known* sockets. Network addresses containing well known sockets are used to contact remote machines for the purpose of, or in absence of, arbitration for a *unique* network address.

For example, to echo to a remote machine, a client would specify the remote machine's address including the well known socket **NSConstants.echoerSocket**. The echo protocol is not a connection oriented protocol, therefore it does not require arbitration for a unique remote address.

In the case of the sequence packet protocol, *listeners* are created using well known sockets and machines contact them by sending packets to that well known socket. But the protocol's connection establishment procedures permit and encourage establishing the connection using a unique address, not *consuming* the well known socket.

Note: A socket number assigned from outside the well known socket number range and then made known to one or more agents does become well known to those agents. The conveyance of that information should be considered a form of arbitration, regardless of how it is done.

The following well known sockets have been assigned for specific purposes and are defined in the interface **NSConstants**. Clients should not use the listed socket number values except for the purpose indicated by their name. Applications that require well known sockets should pick an unassigned value and make it known so that use can be properly registered.

unknownSocketID: **System.SocketNumber** = ...

uniqueSocketID: **System.SocketNumber** = ...

routingInformationSocket: **System.SocketNumber** = ...

echoerSocket: System.SocketNumber = ...
errorSocket: System.SocketNumber = ...
envoySocket: System.SocketNumber = ...
courierSocket: System.SocketNumber = ...
x860ToFileServer: System.SocketNumber = ...
clearingHouseSocket: System.SocketNumber = ...
timeServerSocket: System.SocketNumber = ...
pupAddressTranslation: System.SocketNumber = ...
bootServerSocket: System.SocketNumber = ...
ubIPCSocket: System.SocketNumber = ...
ubBootServerSocket: System.SocketNumber = ...
ubBootServeeSocket: System.SocketNumber = ...
diagnosticsServerSocket: System.SocketNumber = ...
newClearinghouseSocket: System.SocketNumber = ...
electronicMailFirstSocket: System.SocketNumber = ...
electronicMailLastSocket: System.SocketNumber = ...
etherBooteeFirstSocket: System.SocketNumber = ...
etherBootGermSocket: System.SocketNumber = ...
etherBooteeLastSocket: System.SocketNumber = ...
voyeurSocket: System.SocketNumber = ...
netManagementSocket: System.SocketNumber = ...
teleDebugSocket: System.SocketNumber = ...
galaxySocket: System.SocketNumber = ...
protocolCertificationControl: System.SocketNumber = ...
protocolCertificationTest: System.SocketNumber = ...
outsideXeroxFirstSocket: System.SocketNumber = ...
outsideXeroxLastSocket: System.SocketNumber = ...

```
maxWellKnownSocket: System.SocketNumber = ...
NSConstantsExtras: DEFINITIONS = ...
authenticationInfoSocket: System.SocketNumber = ...
mailGatewaySocket: System.SocketNumber = ...
netExecSocket: System.SocketNumber = ...
wsInfoSocket: System.SocketNumber = ...
mazeSocket: System.SocketNumber = ...
pcRoutingTestSocket: System.SocketNumber = ...
maxWellKnownSocket: System.SocketNumber = ...
```

6.2 Packet exchange

PacketExchange: DEFINITIONS = ... ;

PacketExchange is an interface to an implementation of the Packet Exchange Protocol -- a level 2 Network Services Communication Protocol which is defined in *Xerox Internet Transport Protocols*. In contrast to **NetworkStream**, the **PacketExchange** interface provides access to a less reliable, connectionless protocol. The protocol is "single packet" oriented for simplicity, yet includes retransmitting and duplicate suppression for reliability. **PacketExchange** is suitable for applications where a single packet request is immediately followed by a single packet response that is the result of an idempotent operation, or where the communicating clients are capable of providing the necessary level of reliability through the very nature of their interaction.

PacketExchange is implemented by the object file **Communication.bcd**.

Packet Exchange Protocol packets may be sourced from and destined to any socket. While there is no connection established between **PacketExchange** correspondents, it is helpful to think of the entities that participate in the protocol in terms of a *requestor* and *replier*. A replier provides a service (or is a service *agent*), listening for **PacketExchange** packets from requestors. A requestor uses a service by sending requests to a replier. There is minimal state maintained by each end, only enough to remember local network addresses and to handle retransmissions and duplicates.

Note: Due to the constant timeout-retransmission mechanism being used currently, **PacketExchange** is best suited for local network communication. In the future, end-to-end delays will be used for deriving retransmission timeouts, and internetwork link utilization should improve.

Caution: **PacketExchange** is best applied to idempotent operations. This is due to the unreliable nature of the delivery of the reply and the inability to correctly process duplicate requests within the framework of the protocol.

6.2.1 Types and constants

```
PacketExchange.ExchangeClientType: TYPE = MACHINE DEPENDENT {
    unspecified(0), timeService(1), clearinghouseService(2), teledebug(10B),
    electronicMailFirstPEType(20B), electronicMailLastPEType(27B),
    remoteDebugFirstPEType(30B), remoteDebugLastPEType(37B),
    acceptanceTestRegistration(40B), performanceTestData(41B),
    protocolCertification(50B), voyeur(51B), dixieDataPEType(101B),
    dixieAckPEType(102B), dixieBusyPEType(103B), dixieErrorPEType(104B),
    outsideXeroxFirst(100000B), outsideXeroxLast(LAST[CARDINAL])};
```

The **ExchangeClientType** defines well known exchange types that may be used for filtering requests or multiplexing within a service.

```
PacketExchange.ExchangeID: TYPE = MACHINE DEPENDENT RECORD [a, b: WORD];
```

An exchange identifier is assigned to every request. This may be used by replying clients to suppress duplicate requests and is used by the requesting code to identify replies. The field will contain a value that is unique for each request using a function that has a period at least as long as the advertised *maximum packet lifetime* (60 seconds). The semantics of the **ExchangeID** are not sufficient to warrant the field's use as a request *sequence*.

```
PacketExchange.ExchangeHandle: TYPE [2];
```

```
PacketExchange.nullExchangeHandle: READONLY PacketExchange.ExchangeHandle;
```

An exchange handle is the result of one of **PacketExchange**'s create routines and used as a parameter in other procedures. **nullExchangeHandle** may be used to indicate no valid exchange handle exists.

```
PacketExchange.RequestHandle: TYPE = LONG POINTER TO READONLY
    PacketExchange.RequestObject;
```

```
PacketExchange.RequestObject: TYPE = RECORD [
    nBytes: CARDINAL,
    requestCode: PacketExchange.ExchangeClientType,
    requestorsExchangeID: PacketExchange.ExchangeID,
    requestorsAddress: System.NetworkAddress];
```

A request handle is the result of a **PacketExchange.WaitForRequest** and is used as an argument in **PacketExchange.SendReply**. Through the request handle the client can get at some information about the request that is not included in the client data block. The fields addressed by the request handle may not be modified. A request handle must be discarded after the call to **PacketExchange.SendReply**.

```
PacketExchange.WaitTime: TYPE = LONG CARDINAL;
```

```
PacketExchange.defaultWaitTime: PacketExchange.WaitTime = 60000;
```

```
PacketExchange.defaultRetransmissionInterval: PacketExchange.WaitTime = 30000;
```

PacketExchange.WaitTime is a time used in all references having to do with setting wait times in either the *requestor* or *replier*. The time specified is always in milliseconds and will be converted to an internal representation before being used. If the conversion leads to overflow, an *infinite* wait time will be used. Due to the possibility of overflow, clients should be cautious attempting to time intervals greater than approximately 40 minutes. A wait time of zero will be interpreted as an immediate timeout, i.e., one that times out without waiting if and only if the response is not already buffered in the local machine.

The **defaultWaitTime** equal to one minute is taken from the *NS Internet Transport* specification's value for *maximum packet lifetime*. The **defaultRetransmissionInterval** is used to insure that requests will be transmitted at least two times before abandoning the effort.

PacketExchange.maxBlockLength: READONLY CARDINAL;

The maximum length of the block (**Environment.Block**) that can be transmitted via **PacketExchange** is based on the *maximum internet packet size*, a value that is stated in the *NS Internet Transport* specification. Attempting to send requests or replies longer than **PacketExchange.maxBlockLength** will cause an error to be raised.

6.2.2 Signals and errors

PacketExchange.Error: ERROR [why: PacketExchange.ErrorReason];

PacketExchange.ErrorReason: TYPE = {
 blockTooBig, blockTooSmall, noDestinationSocket, noRouteToDestination,
 noReceiverAtDestination, insufficientResourcesAtDestination, rejectedByReceiver,
 hardwareProblem, aborted, timeout};

PacketExchange.Error may be raised by most of the request/reply procedures. The definitions of **ErrorReason** are as follows:

blockTooBig

The block the client attempted to transmit was too big. The size of the block must be in the range [0..**PacketExchange.maxBlockLength**).

blockTooSmall

The block specified by the client to receive a request or reply was smaller than the amount of data transmitted.

noDestinationSocket

This error code is obsolete and unimplemented.

noRouteToDestination

When attempting to transmit a request, it was found that the internet was partitioned in such a manner that the target network is not reachable, or the network field of the remote address is invalid. The remote host has not been contacted.

noReceiverAtDestination

A request was sent to a machine that does not currently have a replier listening on that socket.

Communication with the remote machine has been achieved.

insufficientResourcesAtDestination An error packet was received in response to a **PacketExchange** request. The indication is that either an intermediate internet router or the target machine does not currently have the resources to service the request.

rejectedByReceiver The request was rejected by the replier for some undetermined reason. Communication with the remote machine has been achieved.

hardwareProblem An undefined error packet was received in response to a request.

aborted This error code is obsolete and unimplemented.

timeout This error code is used for internal processing and should not be observed by **PacketExchange** clients.

PacketExchange.TimeOut: SIGNAL;

The time interval set in one of the create routines (**PacketExchange.CreateRequestor** or **CreateReplier**) or **PacketExchange.SetWaitTimes** has expired and the operation has not completed. This signal may be **RESUME'd** in order to wait another timeout interval.

6.2.3 Procedures

```
PacketExchange.CreateRequestor: PROCEDURE [
  waitTime: PacketExchange.WaitTime  $\leftarrow$  PacketExchange.defaultWaitTime,
  retransmissionInterval: PacketExchange.WaitTime  $\leftarrow$ 
    PacketExchange.defaultRetransmissionInterval]
  RETURNS [PacketExchange.ExchangeHandle];
```

CreateRequestor creates a socket on a unique local address. The requestor's wait time and retransmission interval may be specified using the parameters **waitTime** and **retransmissionInterval**. The successful return from **CreateRequestor** results in the client possessing a valid exchange handle that may then be used as an argument in a **PacketExchange.SendRequest** or **Delete**. **CreateRequestor** generates no transmissions to any host and will raise no signals.

```
PacketExchange.CreateReplier: PROCEDURE [
  local: System.NetworkAddress, requestCount: CARDINAL  $\leftarrow$  1,
  waitTime: PacketExchange.WaitTime  $\leftarrow$  PacketExchange.defaultWaitTime,
  retransmissionInterval: PacketExchange.WaitTime  $\leftarrow$ 
    PacketExchange.defaultRetransmissionInterval]
  RETURNS [PacketExchange.ExchangeHandle];
```

CreateReplier creates a **PacketExchange replier** at the well known address, **local**. Since it is expected that repliers are supplying a service to many clients, clients of **CreateReplier** may request more buffering via **requestCount**. **requestCount** represents the number of

requests that may be queued to the replier at any given time. This permits the replier process time to service a request and still not miss new requests that arrive while that processing is in progress.

PacketExchange.Delete: PROCEDURE [h: PacketExchange.ExchangeHandle];

When a *requestor* or a *replier* is no longer needed, it must be deleted. Once deleted, the exchange handle is no longer valid.

Caution: If a client process is waiting inside the packet exchange implementation (either at **WaitForRequest** or **SendRequest**) and the requestor or replier is deleted, that process (or processes) will be aborted and the **ABORTED** signal will be permitted to propagate to the caller. This action is taken despite the popular notion that deleting an instance of a facility with client processes still active inside that facility is a client error.

PacketExchange.RejectRequest: PROCEDURE [
 h: PacketExchange.ExchangeHandle, rH: PacketExchange.RequestHandle];

If a replier client does not wish to respond to a request, the request may be rejected by calling **PacketExchange.RejectRequest**. This permits the implementation to delete the small state object represented by rH, the request handle.

PacketExchange.SendReply: PROCEDURE [
 h: PacketExchange.ExchangeHandle,
 rH: PacketExchange.RequestHandle, replyBlk: Environment.Block,
 replyType: PacketExchange.ExchangeClientType ← unspecified];

To respond to a request, the client calls **PacketExchange.SendReply**, specifying the exchange handle (h) used when he called **PacketExchange.WaitForRequest** and the request handle (rH) returned by that procedure. **replyBlk** describes the data that is to be sent in response. That block cannot be larger than **PacketExchange.maxBlockLength**. The reply packet will have an exchange identifier set to the value specified in **replyType**. This procedure may signal **PacketExchange.Error**.

PacketExchange.SendRequest: PROCEDURE [
 h: PacketExchange.ExchangeHandle, remote: System.NetworkAddress,
 requestBlk, replyBlk: Environment.Block,
 requestType: PacketExchange.ExchangeClientType ← unspecified]
RETURNS [nBytes: CARDINAL, replyType: PacketExchange.ExchangeClientType];

A client that possesses a valid exchange handle (h) may send a request to a remote machine that implements a service in the form of a *replier*. The request must include an **Environment.Block** that represents the request (**requestBlk**) and describes no more than **PacketExchange.maxBlockLength** bytes. The client must also specify an area for the reply to be stored, **replyBlk**. **requestBlk** and **replyBlk** may describe the same area, and either or both may be **Environment.nullBlock** if the protocol being implemented permits it. The value of **requestType** will be copied into the exchange packet and may be used for filtering at the replier.

SendRequest will return only after a valid response has been received. When it returns, **replyBlk** will contain **nBytes** of client data, and the reply received will be of type **replyType**. This procedure may signal **PacketExchange.Error** or **PacketExchange.Timeout**. The

latter may be **RESUME**'d causing the request to reenter the timeout interval. It is important to note the difference between **RESUME**'ng and **RETRY**'ng. **RESUME**'ng will not assign a new exchange identifier permitting the replier to suppress any retranmissions as duplicates if appropriate. **RETRY**'ng will cause a new identifier to be assigned and the replier will not be able to detect it as a duplicate.

Note: **SendRequest** may be aborted via **ProcessAbort**. The **ABORTED** signal will not be caught by **SendRequest**.

```
PacketExchange.SetWaitTimes: PROCEDURE [  
    h: PacketExchange.ExchangeHandle, waitTime, retransmissionInterval:  
        PacketExchange.WaitTime];
```

SetWaitTimes permits an exchange client to adjust the timeout values associated with a exchange handle. **waitTime** affects both **PacketExchange.WaitForRequest** and **SendRequest** while **retransmissionInterval** affects only the latter. Refer to §6.2.1 for additional details about wait times. This procedure raises no signals.

```
PacketExchange.WaitForRequest: PROCEDURE [  
    h: PacketExchange.ExchangeHandle, requestBlk: Environment.Block,  
    requiredRequestType: PacketExchange.ExchangeClientType ← unspecified]  
    RETURNS [rH: PacketExchange.RequestHandle];
```

A client that has created a replier via **PacketExchange.CreateReplier** is expected then to wait for a request to arrive. That is done by calling **WaitForRequest**. It requires a **PacketExchange.ExchangeHandle**, and an **Environment.Block** (**requestBlk**) in which to receive the data of the request. The field **requiredRequestType** may be set to a unique **PacketExchange.ExchangeClientType** or allowed to default to **unspecified** indicating that the exchange client type of the request is not a significant part of the protocol. If **requiredRequestType** is not **unspecified** then only requests of type **requiredRequestType** will be accepted.

PacketExchange.WaitForRequest will return only when a suitable request has arrived. When it does, the data structure pointed to by **rH** will contain additional information about the request. That information may be used by the client to determine if the request is a duplicate or to be ignored for any reason. Once the procedure returns with **rH**, **rH** must be accounted for in one of two manners. It must either be the object of a **PacketExchange.SendReply** (the norm), or it must be dispensed with via **PacketExchange.RejectRequest**.

This procedure may signal **PacketExchange.Error** and **Timeout**. The latter signal may be **RESUME**'d. It would be quite usual to specify an infinite timeout on a replier, thus eliminating the need to service the **PacketExchange.Timeout** signal.

Note: **WaitForRequest** may be aborted via **ProcessAbort**. The **ABORTED** signal will not be caught by **WaitForRequest**.

6.3 Network streams

```
NetworkStream: DEFINITIONS . . . ;
```

A *Network stream* is the principal means by which clients of Pilot communicate between machines. **NetworkStream** provides access to the implementation of the Sequenced Packet Protocol -- a level 2 Internet Transport Protocol which is defined in *Xerox Internet Transport Protocols*. It provides sequenced, duplicate-suppressed, error-free, flow-controlled communication over arbitrarily interconnected communication networks.

The Network stream package is implemented by **Communication.bcd**.

As previously mentioned, **NetworkStream** is implemented by a sequenced packet transducer which utilizes sockets to communicate with machines on a communication network. All data transmission via a Network stream is invoked by means of **Stream** operations. Here, the most common model of communication using Network streams will be described. Subsequent sections provide a description of the actual **NetworkStream** primitives.

A Network stream provides reliable communication between any two network addresses (**System.NetworkAddresses**). The stream (connection) can be set up between the two communicators in many ways -- the most typical case involves a supplier of a service at one end, and a client of the service at the other. Creation of such a stream is inherently asymmetric.

At one end is a *server* -- that is, a process or subsystem offering some service. When a server is operational, one of its processes *listens* for connection requests on its network address (which has previously been made known to potential clients through some binding mechanism) and creates a new Network stream for each separate request it receives. The handle for the new stream is typically passed to a subsidiary process or subsystem (called an *agent*) which gives its full attention to performing the service for that particular client.

At the other end is the *client* of the service. This process or subsystem requests service by actively creating a Network stream, specifying the network address of the server as a parameter. The effect is to create a *connection* between the client and its server agent. These two then communicate by means of the new Network stream set up between them for the duration of the service.

It is not necessary that the client and server be on different machines. If they are on the same machine, Pilot will optimize the transmission of data between them and will avoid the use of physical network resources. Thus, a client does not need to know where a server is located. This scheme permits configuration flexibility -- permitting services that reside on one machine to be split across a number of machines connected together by a network, or vice versa.

The manner in which a client finds out the network address of a server, or the manner in which a server makes its network address known to potential clients is outside the scope of Pilot.

6.3.1 Types and constants

NetworkStream.WaitTime: TYPE = LONG CARDINAL;

WaitTime is used in reference to establishing intervals for timeouts. The value associated with the type is always in milliseconds.

Note: If a wait time interval is assigned a value of zero, subsequent operations will timeout immediately if data is not present when a data request is made.

Caution: The wait time is converted to an internal format to be used by the implementation. If that conversion results in an overflow, subsequent timed operations will never timeout. Clients should use caution when attempting to set timeouts of more than approximately 40 minutes.

NetworkStream.defaultWaitTime: WaitTime = 60000;

The default wait time of 60 seconds is a value taken from the *maximum internet packet lifetime*.

NetworkStream.infiniteWaitTime: READONLY NetworkStream.WaitTime;

The infinite wait time is equivalent to asserting that the operation will never time out, or there is no interest in processing timeouts. It is assumed that any process that uses this value will also be capable of aborting the affected process at some time.

NetworkStream.ClassOfService: TYPE = {bulk, transactional};

The class of service parameter permits the client to convey some hint as to the use of the transport being created. If a client hints the transport is **bulk**, the assertion is that it will be used for a high performance application, such as file transfer or the like. If the client hints **transactional** it is assumed that the transport will be used for alternating traffic, an example of which is remote procedure calls as implemented by Courier.

NetworkStream.uniqueNetworkAddr: READONLY System.NetworkAddress;

The value **uniqueNetworkAddr** may be used as a local address specification to indicate to the underlying code that any legal locally generated network address is applicable. This is equivalent to the client calling **NetworkStream.AssignNetworkAddress** and using the result as the parameter value.

NetworkStream.ConnectionID: TYPE[1];

NetworkStream.uniqueConnID: READONLY NetworkStream.ConnectionID;

NetworkStream.unknownConnID: READONLY NetworkStream.ConnectionID;

A connection identifier is a 16-bit value that is unique within a particular machine. It may not be unique across system restarts. It is used in conjunction with the network address to fully define a Sequence Packet Protocol connection. The value **NetworkStream.uniqueConnID** may be used by clients of **NetworkStream.CreateTransducer** to indicate that they want the implementation to generate a unique **ConnectionID**. This is equivalent to the client calling **NetworkStream.GetUniqueConnectionID** directly and using the result for the same parameter. **NetworkStream.unknownConnID** may be assigned to the **remoteConnID** parameter in a **NetworkStream.CreateTransducer** call. It indicates that the connection identifier will be supplied by the remote machine.

NetworkStream.ListenerHandle: TYPE [2];

The **ListenerHandle** is the result of a **NetworkStream.CreateListener** and is required as a parameter on all other listener operations.

6.3.2 Creating Network streams

Clients are provided access to a Network stream via the **Stream.Handle** and the **Stream.Object** that it references. Network streams are variants of generic Pilot streams. For the general definition of Pilot streams, see **Chapter 3**.

Network streams are *usually* created in one of two ways depending on whether the stream is supporting a client that is consuming a service or providing a service. The consumer will use **NetworkStream.Create** while the server uses the listener mechanism. Both processes are clients of **NetworkStream.CreateTransducer**. **CreateTransducer** may also be called directly by clients, provided they are familiar with the options it permits.

```
NetworkStream.CreateTransducer: PROCEDURE [
    local, remote: System.NetworkAddress,
    connectData: Environment.Block ← Environment.nullBlock,
    localConnID, remoteConnID: NetworkStream.ConnectionID,
    activelyEstablish: BOOLEAN,
    timeOut: NetworkStream.WaitTime ← NetworkStream.defaultWaitTime,
    classOfService: NetworkStream.ClassOfService ← bulk]
RETURNS [Stream.Handle];
```

NetworkStream.CreateTransducer will not return to the caller with the **Stream.Handle** until the connection is fully established. When established, the stream is ready to perform stream operations with the cooperating partner of the connection as specified in **remote**. The value **local** is usually specified as **NetworkStream.uniqueNetworkAddr**. This value is recognized by the create process and will cause a unique address to be generated. That address is generated by calling **NetworkStream.AssignNetworkAddress**. (The client is welcome to call the routine directly and use its results for the value of **local**.) It is not recommended that **local** consume a *well known socket*. The remote address must be fully specified, including the socket. The socket field of **remote** may be a well known socket. If so, the connection actually established will not consume that socket, but generate a unique network address in its place.

localConnID is usually defaulted to **NetworkStream.uniqueConnID**. Alternatively, the client may use the results of **NetworkStream.GetUniqueConnectionID** for the value of **localConnID**. Usually, the value of **remoteConnID** is set to **NetworkStream.unknownConnID**. This asserts that the value of the remote's connection identifier will be generated by the remote machine and its value conveyed during the connection rendezvous.

The boolean **activelyEstablish** is used to establish the solicitor/listener relationship normally required to arbitrate a connection. If **activelyEstablish** is **TRUE**, the create process will transmit connection requests to the remote. If it is **FALSE**, it will merely listen for the connection requests. In some cases, both parties know the entire set of connection parameters, including the connection identifiers. This implies some previous binding arbitration has occurred. It is possible, under those conditions, to create transducers on both the local and remote machines that are fully established, without transmitting any information at all.

When creating a transducer, **timeout** is used for two different purposes. If **activelyEstablish** is **TRUE**, it will be used as the time allowed for the remote to respond to the connection establishment requests. It will also be used as the value of **timeout** for stream **get** operations, i.e., the interval permitted to expire during data input operations before the stream implementation signals **Stream.TimeOut**.

The **classOfService** parameter affords the client the opportunity to hint the type of application the stream is to support. Both parties of the connection should select the same class. **transactional** will be assumed if there is disagreement.

The **Stream.Handle** returned is a variant of a generic Pilot byte stream handle. The positioning operations, **getPosition** and **setPosition**, are unimplemented and will result in **Stream.InvalidOperation**.

CreateTransducer may generate the error **NetworkStream.ConnectionFailed**. A process blocked in **CreateTransducer** may also be aborted (**Process.Abort**). **CreateTransducer** will not catch the **ABORTED** signal.

6.3.2.1 Creating client streams

```
NetworkStream.Create: PROCEDURE [
    remote: System.NetworkAddress,
    connectData: Environment.Block ← Environment.nullBlock,
    timeout: NetworkStream.WaitTime ← NetworkStream.defaultWaitTime,
    classOfService: NetworkStream.ClassOfService ← bulk]
    RETURNS [Stream.Handle];
```

Create is the most common method that a client stream client uses to solicit the creation of a transport to a server client. This procedure is a client of **NetworkStream.CreateTransducer**. **Create** assigned a value of **NetworkStream.uniqueNetworkAddress** to **local**, **NetworkStream.uniqueConnectionID** to **localConnID** and asserts **activelyEstablish** to be **TRUE** causing the process to transmit the needed request packets to solicit the connection.

6.3.2.2 Creating server streams

```
NetworkStream.CreateListener: PROCEDURE [addr: System.NetworkAddress]
    RETURNS [NetworkStream.ListenerHandle];
```

Creating a listener creates the state object (represented by the **ListenerHandle**). The state object includes a socket at **addr**. **CreateListener** does not cause any data to be transmitted. It does provide the necessary buffering and queuing to receive data. A listener will exist as such until it is deleted via **NetworkStream.DeleteListener**. **CreateListener** generates no signals.

```
NetworkStream.Listen: PROCEDURE [
    listenerH: NetworkStream.ListenerHandle,
    connectData: Environment.Block ← Environment.nullBlock,
    listenTimeout: NetworkStream.WaitTime ← NetworkStream.infiniteWaitTime]
    RETURNS[remote: System.NetworkAddress, bytes: CARDINAL];
```

Once a listener is created, the client must provide the process to actually listen. This is done by calling **NetworkStream.Listen**. When an acceptable connection request packet arrives at the address specified in **CreateListener**, **Listen** will return with the network address of the requestor (**remote**) and the number of bytes received (**bytes**) in the rendezvous. The client then has the opportunity to reject or honor the connection request. A connection request is rejected either by calling **NetworkStream.Listen** again or by deleting the listener. Both will cause an error packet to be transmitted to the requestor.

If no suitable packet arrives at the socket in **listenTimeout** milliseconds, **listen** will raise the signal **NetworkStream.ListenTimeout**. This signal may be resumed. The default value of **infiniteWaitTime** implies that the listener should never timeout, which is an acceptable (and normal) practice.

```
NetworkStream.ApproveConnection: PROCEDURE [
    listenerH: NetworkStream.ListenerHandle,
    streamTimeout: NetworkStream.WaitTime ← NetworkStream.infiniteWaitTime,
    classOfService: NetworkStream.ClassOfService ← bulk]
    RETURNS [sH: Stream.Handle];
```

When **NetworkStream.Listen** returns and the client wishes to honor the connection request, he calls **NetworkStream.ApproveConnection**. **ApproveConnection** is a client of **NetworkStream.CreateTransducer**. The local address and connection identifier are defaulted to **NetworkStream.uniqueNetworkAddr** and **NetworkStream.uniqueConnID** respectively. The values for remote address and connection identifier are taken from the appropriate fields of the packet requesting the connection. The client is given the opportunity to provide a hint about the expected application of the stream by assigning an appropriate value to **classOfService**. This hint should agree with the hint provided by the remote requestor.

In spite of the evidence that a communication path exists between the local machine and the remote requestor, this procedure may still signal **NetworkStream.ConnectionFailed**.

```
NetworkStream.DeleteListener: PROCEDURE [listenerH: NetworkStream.ListenerHandle];
```

Should the client desire to no longer listen at the socket specified in **CreateListener**, the listener should be deleted. It is advised that this be done at a time when no process is actively listening. The **Listen** process is abortable (**Process.Abort**). The procedure **DeleteListener** may signal **NetworkStream.ListenError** if **listenerH** does not represent a valid **ListenerHandle**.

Caution: If **DeleteListener** notices a process blocked in **Listen**, it will abort that process and the signal **ABORTED** will be propagated to the **Listen** client. This action is taken despite the popular notion that deleting an instance of a facility with active processes inside that facility is a client error.

6.3.3 Signals and errors

```
NetworkStream.ConnectionSuspended: ERROR [why: NetworkStream.SuspendReason];
NetworkStream.SuspendReason: TYPE = {
    notSuspended, transmissionTimeout, noRouteToDestination,
    remoteServiceDisappeared};
```

Clients of Pilot streams that are implemented by Network streams are responsible for catching not only all **Stream** signals, but also a Network streams unique signal. That signal is **ConnectionSuspended**, a name the implies the stream has been established but now is failing. The signal carries with it a reason for the suspension, and the table following describes the reasons.

notSuspended

The connection is not suspended. This state should never be observed by a client. It is included to simplify internal processing.

transmissionTimeout

A connection that was previously communicating has not seen a response from the remote machine for an extended period of time. The internal processing of SPP retransmits packets at computed intervals until they are acknowledged. If a packet is retransmitted more than 30 times without acknowledgement, the connection is abandoned. The interval between retransmissions is computed based upon previous response rates and is initially (before statistics can be gathered) based on the number of internet routers that the packet must pass through to reach the remote machine. In the absence of retransmissions and in conjunction with them, idle line probes are also transmitted at computed intervals to the remote host. The number of probes that will be transmitted without acknowledgement is fixed, and the interval between probe transmissions is computed based simply on the number of internet routers that the packet must pass through to reach the remote machine.

noRouteToDestination

A previously functional connection has discovered that the internet has become partitioned in some manner that the remote host is no longer accessible, either because it must pass through too many internet routers or a path has totally disappeared.

remoteServiceDisappeared

A previously functional connection has been notified that the remote address no longer exists. In other words, the socket on which the connection was based has been deleted.

```
NetworkStream.ConnectionFailed: SIGNAL [why: NetworkStream.FailureReason];
NetworkStream.FailureReason: TYPE = {
    timeout, noRouteToDestination, noServiceAtDestination, remoteReject,
    tooManyConnections, noAnswerOrBusy, noTranslationForDestination, circuitInUse,
    circuitNotReady, noDialingHardware, dialerHardwareProblem};
```

NetworkStream.ConnectionFailed is applicable only to clients who are attempting to establish a SPP connection. This includes clients of **NetworkStream.CreateTransducer**, **Create** and **ApproveConnection**. The implication is that the connection never was established; it does not always conclude that the remote machine was not contacted.

timeout	The time stated in the parameter <code>timeout</code> of <code>NetworkStream.CreateTransducer</code> has expired and the packets requesting connection establishment have not been acknowledged. This and only this value of <code>why</code> may be resumed.
noRouteToDestination	Attempts to find a route to the remote network failed. Either the network is temporarily partitioned in such a manner that the network is unreachable or the network number in the remote address is invalid. In any case, the remote host has not been contacted.
noServiceAtDestination	There is no listener at the address specified in the remote address. The machine did respond indicating that the internet and the machine are both responsive.
remoteReject	The process implementing the service at the remote address rejected the request for connection (see §6.3.2.2). Since the remote host sent the reject, it is obvious that the internet and the remote host are both responsive.
tooManyConnections	The number of simultaneous connections permitted on the local machine would have been exceeded by creating a new stream. In cases where <code>activelyEstablish</code> is <code>TRUE</code> (e.g., <code>NetworkStream.Create</code>), no communication with a remote machine has been attempted.
noAnswerOrBusy	This error is applicable only to circuit oriented connections. When the phone was dialed, it was either not answered or was busy. The remote machine has not been contacted.
noTranslationForDestination	This error is applicable only to circuit oriented connections. There is no phone number currently registered for access to the network specified. The remote machine has not been accessed.
circuitInUse	This error is applicable only to circuit oriented connections. The circuit that must be used to access the remote machine is currently in use. The remote machine has not been contacted.
circuitNotReady	This error is applicable only to circuit oriented connections. The circuit that must be used to access the remote machine was not ready. Possibly the modems need to be made ready or the phone needs to be manually dialed. The remote machine has not been contacted.
noDialingHardware	An attempt to access a remote network that would require a circuit oriented device, but the proper hardware does not

exist to make such a connection. The remote machine has not been contacted.

dialerHardwareProblem

An attempt to access a remote network that would require a circuit oriented device, but the hardware needed to make such a connection appears to be inoperable. The remote machine has not been contacted.

NetworkStream.ListenError: ERROR [reason: NetworkStream.ListenErrorReason];

NetworkStream.ListenErrorReason: TYPE = {
illegalAddress, illegalHandle, illegalState, blockTooShort};

NetworkStream.ListenError is applicable only to clients of the listening procedures. The definition of the error reason is as follows:

illegalAddress

The local address specified in **NetworkStream.CreateListener** is illegal. This is due to the fact that **addr** already exists on the local machine or the socket field of **addr** has a value of zero.

illegalHandle

The handle specified in one of the listener procedures is not valid. Either the handle has been deleted (**NetworkStream.DeleteListener**) or was never created.

illegalState

The state of the listener handle specified to one of the listener procedures (**NetworkStream.ApproveConnection** or **Listen**) was in an illegal state for that operation. In the case of **NetworkStream.ApproveConnection**, the state indicated that no request for connection had been received. In the case of **NetworkStream.Listen**, a process was already found to be listening, implying that two or processes are sharing the listener handle.

blockTooShort

The **Environment.Block** provided to collect the connection data in **NetworkStream.Listen** was not large enough to hold the data supplied by the requestor of the connection. (Note: The ability to pass rendezvous information is not currently implemented. Consequently, this status should never be observed.)

NetworkStream.ListenTimeout: SIGNAL;

NetworkStream.ListenTimeout will be raised if no acceptable packet arrives at the listener within the specified time interval. That interval is client specified in **NetworkStream.CreateListener** as **listenTimeout**. This signal may be **RESUME**'d, causing the interval to be reentered. It is a common practice to use **NetworkStream.infiniteWaitTime** as a value for **listenTimeout** when creating listeners to eliminate the need to process the **ListenTimeout** signal.

6.3.4 Utilities

The following utility functions are available to NetworkStream clients. In general they provide functionality unique to Network streams.

6.3.4.1 Assigning unique address components

NetworkStream.AssignNetworkAddress: PROCEDURE RETURNS [System.NetworkAddress];

AssignNetworkAddress returns to the caller a network address that is unique for the current system restart. It is constructed from the local machine's network number for the default communication device, the local machine's processor identification number, and a unique socket number that is not a well known. The result is applicable to any argument that might use a unique local address.

NetworkStream.GetUniqueConnectionID: PROCEDURE
RETURNS [iD: NetworkStream.ConnectionID];

GetUniqueConnectionID will return to the caller a connection identifier that is unique within the current system load. It may be used any place a **NetworkStream.uniqueConnectionID** would be applicable (**NetworkStream.CreateTransducer**).

6.3.4.2 Discovering addresses of established streams

NetworkStream.FindAddresses: PROCEDURE [sH: Stream.Handle]
RETURNS [local, remote: System.NetworkAddress];

A client may find the local and remote network addresses of an existing stream by calling **FindAddresses**.

6.3.4.3 Controlling timeouts

NetworkStream.SetWaitTime: PROCEDURE [sH: Stream.Handle, time: NetworkStream.WaitTime];

This procedure may be used to adjust the stream timeout of an established network stream.

Note: Since the generic Pilot stream also provides the same capability, it is suggested that use of this procedure be phased out in preference to the standard operation. This operation will be removed in the next release of Pilot.

6.3.4.4 Closing streams

An implementation of a close protocol is provided by Network streams. This method of terminating dialog on a stream is suggested in the **NS Internet Protocol Specification**. Use of these routines (or any like them) is considered optional.

NetworkStream.CloseStatus: TYPE = {good, noReply, incomplete};

NetworkStream.closeSST: Stream.SubSequenceType = 254;

NetworkStream.closeReplySST: Stream.SubSequenceType = 255;

NetworkStream.Close: PROCEDURE [sH: Stream.Handle]

RETURNS [NetworkStream.CloseStatus];

NetworkStream.CloseReply: PROCEDURE [sH: Stream.Handle]

RETURNS [NetworkStream.CloseStatus];

To initiate a close sequence, a client may call **NetworkStream.Close**. That procedure will transmit an empty packet with a **Stream.SubSequenceType** of **NetworkStream.closeSST**. The side effect of this is that all buffered data will be transmitted before the empty packet. After the **closeSST** has been transmitted, the procedure will attempt to receive a **NetworkStream.closeReplySST**. All data not of subsequence type **closeReplySST** will be ignored. When a **NetworkStream.closeReplySST** is received, the procedure will transmit **NetworkStream.closeReplySST** and return, without waiting. **NetworkStream.Close** raises no signals.

If a client protocol uses the close procedure and receives a **NetworkStream.closeSST**, it should respond by calling **NetworkStream.CloseReply**. This procedure will transmit a **NetworkStream.closeReplySST**, the side effect of which will be to force transmission of all currently buffered data. After sending the **closeReplySST**, the procedure will attempt to receive a packet with subsequence type of **closeReplySST**. **NetworkStream.CloseReply** raises no signals.

The **NetworkStream.CloseStatus** has the following definitions:

- | | |
|-------------------|---|
| good | The close protocol terminated cleanly. All data the was buffered by the stream implementation prior to initiating the close was transmitted and acknowledged at least to the level of the Network stream client. |
| noReply | There was no response to the NetworkStream.closeSST . All data buffered in the local stream implementation has been transmitted, but may not have been acknowledged. |
| incomplete | The local machine transmitted a NetworkStream.closeReplySST in response to a NetworkStream.closeSST and received no response. All data buffered in the local stream implementation has been transmitted and acknowledged. The closeReplySST is expected, but not required. |

6.3.5 Attributes of Network streams

Network streams are byte streams built on top of the *Sequenced Packet Protocol*. Due to the distributed nature of the streams clients may find some behavior unique. This section will attempt to point out the unique areas of these streams with the intent of assisting in design and debugging applications using Network streams.

All output operations (**putByte**, **putWord**, **put**, **setSST**, **sendAttention**, **sendNow**) buffer the data internally, transmitting those buffers only when they are either full or the semantics of an operation indicate they must be transmitted. When the buffers are actually transmitted it is possible that the client process will be blocked *indefinitely* if the remote partner in the connection is not consuming data. This is known as the *waiting for allocation* state. All output operations may signal **NetworkStream.ConnectionSuspended**.

All input operations (**getByte**, **getWord**, **get**) may signal any of the defined Stream errors, except **Stream.EndOfStream**. The *end of stream* concept is not implemented by Network streams. All input operations may also signal **NetworkStream.ConnectionSuspended**. Physical packet boundaries will not be visible to the byte stream client, but they may be inferred through the input operation status or signals. Any operation that signals or returns a completion status other than **normal** is at a packet boundary. On a **normal** return the stream may or may not be at a packet boundary.

Any Network stream operation may be aborted (**Process.Abort**). The **ABORTED** signal will be permitted to propagate to the stream client.

6.3.5.1 Elements of Network stream objects

Elements of a Network **Stream.Object** are::

inputOptions	The defaultInputOptions defined by the Stream interface are almost always inappropriate for Network streams. In particular, terminateOnEndRecord should be TRUE . This is due to the fact that Network streams do not implement the <i>end of stream</i> concept, but do have the concept of a message or <i>logical record</i> . If terminateOnEndRecord is FALSE , input operations will not terminate at the end of the logical record and the return status endRecord will never be observed. If a get is not permitted to terminate with an endRecord status, it will invariably find itself waiting to complete a transfer when it should be responding to the information it has in hand.
getByte	GetByte returns the byte of data from the byte stream. It asserts the input options as [FALSE, FALSE, FALSE, TRUE, TRUE, TRUE, TRUE, FALSE], making the signals Stream.SSTChange , Stream.Attention or Stream.TimeOut possible.
putByte	PutByte appends one byte of client data to the byte stream. Should that addition cause the internal buffer to be filled, it will be transmitted over the established connection.
getWord	GetWord returns the word of data from the byte stream. It asserts the input options as [FALSE, FALSE, FALSE, TRUE, TRUE, TRUE, TRUE, FALSE], making the signals Stream.SSTChange , Stream.Attention or Stream.TimeOut possible. GetWord operations that signal SSTChange or Attention are ambiguous if the signal is raised after processing half (or one byte) of the request. Such ambiguity is a client error. The sender and receiver should use the same type of alignment characteristics.
putWord	PutWord appends one word of client data to the byte stream. Should that addition cause the internal buffer to be filled, it will be transmitted over the established connection.
get	Get retrieves the number of bytes specified in block (Environment.Block). If the number of bytes requested is actually transferred, the status returned by get will always be normal unless the inputOptions have been set to terminateOnEndOfRecord and the end of the logical record is detected. Conversely, a completion code of anything other than

normal or **endRecord** implies that the transfer operation was not satisfied. The input options are settable by the client, so the various stream signals are possible depending on the options.

If **signalLongBlock** or **signalShortBlock** is **TRUE**, packet boundary semantics will be applied to the byte stream and the client will be notified by the appropriate signal **Stream.LongBlock** or **Stream.ShortBlock**. These two input options should be used only by clients that wish to directly control buffering, and those clients would be well advised to use something other than Network streams in their application,

- put** **Put** appends the specified **block** (**Environment.Block**) to the byte stream. That addition may cause any (bounded) number of internal buffers to be transmitted. The parameter **endRecord** may be set to **TRUE** causing the any currently buffered data to be transmitted and define the end of a logical record. A **put** specifying no bytes and with **endRecord** set to **TRUE** is equivalent to a **sendNow** with **endRecord** set to **TRUE**. The **endRecord** status is preserved by the Network stream and detectable by the receiving client.
- setSST** **SetSST** will cause a buffer to be transmitted with the current **sst** if and only if the new **sst** is of a different value. If there was no data buffered, an empty buffer will be transmitted carrying only the changed **sst** state. And lastly, the new **sst** will be recorded and declared to be the current **sst**. When a stream is created, it is assigned a default **sst** of value 0.
- sendAttention** **SendAttention** causes an attention byte to be appended to the byte stream. The Network stream implementation also performs some heroics in its attempts to deliver attentions. This amounts to expediting the delivery, circumventing, if necessary, SPP allocation window constraints. **SendAttention** assumes the receiver is taking equally heroic action. Due to the additional overhead in such operations, it is advised that attentions be used judiciously.
- waitAttention** **WaitAttention** allows the client process to wait for an out-of-band attention notification. If a sending client is transmitting attentions, it is the responsibility of the receiving client to process both the in-band and out-of-band attentions. Failure to do so is a client error and will cause the stream to fail. Only a small number of out-of-band attentions will be maintained (less than 10). When that number is reached, the connection will no longer be able to receive data.
- delete** **Delete** causes the current processes and buffering used by the stream implementation to be destroyed. No attempt to clean up the stream is made. The remote partner of the connection is not notified that the local has been deleted. It is a client responsibility to insure that the application data has been satisfactorily delivered before deleting the connection.
- getPosition** **GetPosition** is not implemented by Network streams.

setPosition	SetPosition is not implemented by Network streams.
sendNow	SendNow forces transmission of a buffer. That buffer may contain internally buffered data, or it may be an empty buffer. SendNow with endRecord set to TRUE defines the end of a logical record. The logical record boundary status is preserved by Network streams during transmission and detectable by receiving clients.
clientData	This field is not used by Network streams.
getSST	GetSST will return to the caller the current output SST, i.e., the SST that can be set by the client via setSST . This procedure raises no signals.

6.3.5.2 Input options

terminateOnEndRecord	If terminateOnEndRecord is FALSE , the stream implementation will ignore logical record boundaries in incoming packets, and continue to process incoming packets until the get request is satisfied. If it is TRUE , it will assume an exceptional condition at the end of a packet that carries a logical record boundary status, terminate the transfer, and return with an endRecord completion code.
signalEndOfStream	Network streams do not implement this concept.

6.3.5.3 Completion codes

These codes are returned from the **get** procedure.

normal	A normal return is one that satisfies the transfer request, i.e., the number of bytes requested.
endRecord	An endRecord status indicates that in an attempt to satisfy the input request, a buffer that carries a end of logical record status was consumed and that the input options indicated that the request should terminateOnEndRecord . The input request <i>may</i> not be complete.
sstChange	This status indicates that the <i>data stream type</i> of the data stream has changed and that the next byte of data in the byte stream will be of type sst . The get procedure's transfer is not complete.
endOfStream	The endOfStream concept is not implemented by Network streams.
attention	This status indicates that the next byte of the byte stream is an in-band attention byte. The in-band attention marks the point in the byte stream where the attention was transmitted, even though the out of band notification may have arrived at a different time. The get procedure's transfer is not complete.

6.4 Routing

Router: DEFINITIONS . . . ;

All routers transmit packets to an immediate host that is, or is closer to, the final destination host. *Internetwork* routers are responsible for keeping other internetwork routers and simple routers informed of as much of the topology as they require, and for the actual forwarding of packets from one net to another. They always know the topology of the entire internet. *Simple* routers are mostly ignorant of the network topology, and learn only enough about it to send packets sourced in the local machine toward their destination via the optimal route. Each instance of Pilot has a simple router to help direct packets to their proper destination. **Router** offers operations for using Pilot as a simple router, and for discovering information about the topology of the internetwork.

Distances between networks are measured in the number of internetwork routers a packet must be routed through from source to destination. The unit of measurement used is a *hop*. The *delay* to a network is the number of hops from the source host to the destination host. The local network is always considered to be zero hops away; a network available through a single internetwork router is one hop away.

The simple routers keep a routing table by which packet forwarding decisions are made. A routing table entry contains a destination network number, the internetwork router address to which packets bound for the destination network should be forwarded, and the delay to the network in hops. The routing table contains entries only for those destination networks that have been accessed (*i.e.*, had traffic transmitted to them) within the last ninety seconds. The table entries are created when a client tries to send a packet to a network unknown to Pilot, causing a routing table cache fault. The fault causes at least one routing request to be made of a local internetwork router. The local routing table for a simple router grows only when routing table faults occur. Thus, it is not a complete picture of the networks that are reachable.

The routing table for simple router is maintained by aging entries to which no traffic has been generated, and discarding the old entries.

Router is implemented by the configuration **Communication.bcd**.

6.4.1 Types and constants

Router.endEnumeration: READONLY System.NetworkNumber;

Returned by the **EnumerateRoutingTable** stateless enumerator, **endEnumeration** indicates the end of the list of entries in the current routing table has been reached.

Router.infinity: CARDINAL = 16;

infinity is the number of hops that defines an unreachable network. Any network that is **infinity** or more hops away from the local net is unreachable.

Router.PhysicalMedium: TYPE = {ethernet, ethernetOne, phononet, clusternet};

PhysicalMedium defines the various types of networks on the device chain.

ethernet **ethernet** is a 10 M-bit ethernet, as defined by *The Ethernet*, Version 1.0, September 30, 1980.

ethernetOne Also referred to as the *experimental* ethernet, **ethernetOne** is a 3 M-bit ethernet.

phonenet Based on the create procedure in RS232C, **phonenet** is a phone line network.

clusternet **clusternet** is a clusternet network, a group of one or more RS232C ports that is used for remote workstations.

Router.RoutersFunction: **TYPE** = {**vanillaRouting**, **interNetworkRouting**};

The type of routing function the current router has is defined by **RoutersFunction**.

vanillaRouting The function for all simple routers is **vanillaRouting**. These routers are capable only of requesting routing information, receiving the responses from the internetwork routers and maintaining a table.

interNetworkRouting The function for internetwork routers is **interNetworkRouting**. These are the routing information suppliers that know about the network topology. They respond to routing requests and periodically send out gratuitous routing information updates.

Pilot directly supports only **vanillaRouting**.

Router.startEnumeration: **READONLY System.NetworkNumber**;

Used with the **EnumerateRoutingTable** stateless enumerator, **startEnumeration** is passed to start the enumeration of the entries in the current routing table.

6.4.2 Signals and errors

Router.NetworkNonExistent: **ERROR**;

Raised by **GetNetworkID** and **SetNetworkID**, this error indicates the device specified in the call does not exist.

Router.NoTableEntryForNet: **ERROR**;

Raised only by **GetDelayToNet**, this error indicates the network specified by the client could not be found in the routing table and the information could not be obtained from an internetwork router.

6.4.3 Procedures

Router.AssignAddress: **PROCEDURE RETURNS [System.NetworkAddress]**;

This procedure returns a network address with the primary network number (*i.e.*, the first device on the device chain), the local machine's ID and a unique socket number. It is

typically used by clients who need to generate a unique address. **Note:** this address is not unique across system restarts.

Router.AssignDestinationRelativeAddress: PROCEDURE [System.NetworkNumber]
RETURNS [System.NetworkAddress];

Clients who wish to obtain their address with a unique socket number and who know what destination network they will be communicating with should call **AssignDestinationRelativeAddress**. The network number passed is the destination network number. Instead of setting the network field of the returned value to the primary network number, the procedure will set it to the number of the local network on the best known route to the destination net. The host field will be set to the processor ID of the local machine and socket field to a unique socket number.

Router.EnumerateRoutingTable: PROCEDURE[
previous: System.NetworkNumber, delay: CARDINAL]
RETURNS [net: System.NetworkNumber];

A stateless enumerator, **EnumerateRoutingTable** is used to dump that portion of the current local routing table which represents routes within a certain delay of the local network.

delay The number of hops to the remote network the client is interested in is specified by **delay**.

previous **previous** is the network number obtained from the last call. If this is the first call to the procedure, **previous** should be set to **startEnumeration**.

EnumerateRoutingTable will return the net and delay of the first net following **previous** that has a delay equal to **delay**. Pilot's simple router holds only entries for those routes recently accessed (*i.e.*, have had traffic transmitted to them within the last 90 seconds) or those that have been obtained by an explicit routing information request via **FillRoutingTable** or **GetDelayToNet**. In general, a machine can be connected to more than one local network by having more than one ethernet controller. In this case, the machine also has more than one network address. To determine the list of local networks, **EnumerateRoutingTable** can be used with **maxDelay** set to 0. The networks are enumerated in ascending order of network number.

Router.FillRoutingTable: PROCEDURE [maxDelay: CARDINAL ← Router.infinity];

FillRoutingTable solicits information on all networks within the specified number of hops from the local net.

maxDelay **maxDelay** is the maximum delay in hops of the networks that the client wishes to collect information about. The default value is **infinity**, filling the table with information about every known reachable network.

Routing information requests are broadcast on the local network. All subsequent responses from the internetwork routers, whether associated with the request or gratuitous, will cause information about networks **maxDelay** or less away to be saved in the local routing table. That information will be continuously updated if and when new information is received.

FillRoutingTable followed by **EnumerateRoutingTable** can be used to determine the networks within the desired number of hops from the local net. The filling will continue until *all* clients who have called **FillRoutingTable** call it again with a **maxDelay** of zero, indicating they are no longer interested in saving incoming entries. There *must* be a call with a **maxDelay** of zero for *every* call with a non-zero delay in order to properly maintain the table. If multiple clients have called this procedure, the greatest **maxDelay** specified will be used in determining which entries to save in the table.

Router.FindDestinationRelativeNetID: PROCEDURE[**System.NetworkNumber**]
RETURNS [**System.NetworkNumber**];

When passed the number of a destination net, **FindDestinationRelativeNetID** will return the number of the local network on the best known route to the destination network. It is useful for setting an unknown source network number when the destination network is known.

Router.FindMyHostID: PROCEDURE RETURNS [**System.HostNumber**];

This procedure returns the processor ID of the local machine.

Router.GetDelayToNet: PROCEDURE [**net: System.NetworkNumber**] RETURNS [**delay: CARDINAL**];

Clients who wish to find the current delay to a specific net may call **GetDelayToNet**.

net The number of the network that the client is interested in is specified by **net**.

delay The number of hops from the local net to **net** is specified by **delay**.

If the **net** is not found in the current routing table, Pilot requests routing information from local internetwork routers. If **net** is unknown to the local machine and cannot be obtained from the internetwork router, **Router.NoTableEntryForNet** is raised.

GetDelayToNet is useful for determining timeouts and retransmission intervals for clients, restrict broadcasts, or for determining the network topology close to the system element. It might also be useful in choosing between two servers offering the same service, based upon the delay to each element.

Router.GetNetworkID: PROCEDURE[**physicalOrder: CARDINAL, medium: PhysicalMedium**]
RETURNS [**System.NetworkNumber**];

The network number of any network directly attached to the local machine can be discovered by calling **GetNetworkID**.

physicalOrder The is the index of the network driver on the device chain is the **physicalOrder**. (the *primary* network always has a physical order of 1)

medium The type of network involved is medium.

This procedure will raise the error **NetworkNonExistent** if there is no such device.

Router.GetRouterFunction: PROCEDURE RETURNS [**RoutersFunction**];

Clients wishing to discover the function of the current router registered with Pilot may call **GetRouterFunction**. The function of the router supplied by Pilot is always **vanillaRouting**, the simple routing information requestor. Special facilities may be used to install an internetwork router on a machine.

```
Router.SetNetworkID: PROCEDURE[  
    physicalOrder: CARDINAL, medium: PhysicalMedium,  
    newNetID: System.NetworkNumber]  
RETURNS [oldNetID: System.NetworkNumber];
```

Special clients can change their network number without rebooting by calling **SetNetworkID**.

physicalOrder The order of the network on the device chain is the **physicalOrder**.

medium **medium** is the type of network.

newNetID The new network number assigned to the specified device is **newNetID**.

oldNetID The network number previously associated with the device is **oldNetID**.

A call to this procedure may raise the error **Router.NetworkNonExistent** if there is no such device.

Caution: This procedure should only be used by sophisticated clients who are knowledgeable about the network and network numbers (*i.e.*, Internetwork router implementations).

6.5 RS232C communication facilities

Pilot supports channel-level access to multiple full-duplex RS232C *ports* providing all of the standard channel procedures listed in §5.1, as well as several specific to RS232C communication. This allows the client access to the *equipment* connected to the RS232C port.

In addition to a channel interface (§6.5.3), Pilot provides facilities to start and stop the RS232C channel code (§6.5.4), and to dial telephone numbers via RS366 dialing hardware associated with RS232C ports (§6.5.5). The RS232C facilities are implemented by the configuration **RS232CIO.bcd**.

6.5.1 Correspondents

RS232CCorrespondents: DEFINITIONS . . . ;

This interface defines the possible correspondents of the RS232C channel. Each correspondent is used to set certain line parameters. The interface also defines the different outcome possibilities of the auto recognition facility of the RS232C channel.

6.5.1.1 Types and constants

RS232C.AutoRecognitionOutcome: TYPE = RS232CEnvironment.AutoRecognitionOutcome;
RS232CCorrespondents.failure: RS232CEnvironment.AutoRecognitionOutcome = ...
RS232CCorrespondents.asciiByteSync: RS232CEnvironment.AutoRecognitionOutcome = ...
RS232CCorrespondents.ebcdicByteSync: RS232CEnvironment.AutoRecognitionOutcome = ...
RS232CCorrespondents.bitSync: RS232CEnvironment.AutoRecognitionOutcome = ...
RS232CCorrespondents.nsProtocol: RS232CEnvironment.AutoRecognitionOutcome = ...
RS232CCorrespondents.illegal: RS232CEnvironment.AutoRecognitionOutcome = ...
RS232CCorrespondents.xerox800: RS232CEnvironment.Correspondent = ...
RS232CCorrespondents.xerox850: RS232CEnvironment.Correspondent = ...
RS232CCorrespondents.system6: RS232CEnvironment.Correspondent = ...
RS232CCorrespondents.cmcll: RS232CEnvironment.Correspondent = ...
RS232CCorrespondents.ttyHost: RS232CEnvironment.Correspondent = ...
RS232CCorrespondents.nsSystemElement: RS232CEnvironment.Correspondent = ...
RS232CCorrespondents.ibm3270Host: RS232CEnvironment.Correspondent = ...
RS232CCorrespondents.ibm2770Host: RS232CEnvironment.Correspondent = ...
RS232CCorrespondents.ibm6670Host: RS232CEnvironment.Correspondent = ...
RS232CCorrespondents.ibm6670: RS232CEnvironment.Correspondent = ...
RS232CCorrespondents.xerox860: RS232CEnvironment.Correspondent = ...
RS232CCorrespondents.nsSystemElementBSC: RS232CEnvironment.Correspondent = ...
RS232CCorrespondents.siemens9750: RS232CEnvironment.Correspondent = ...

The correspondent implies information about the data formatting which the channel must perform, and should be set prior to data transfers.

Note: **xerox800** is not currently supported.

6.5.1.2 Procedures

RS232CCorrespondent.AutoRecognitionWait: PROCEDURE [channel: RS232C.ChannelHandle]
 RETURNS [outcome: RS232C.AutoRecognitionOutcome];

If the line type in the parameter object (see §6.5.3.1) is set to **autoRecognition**, the client is asking the RS232C channel to attempt to determine as much as possible about the

correspondent at the other end of the communication line. The client should await the results of this auto-recognition attempt via a call to **AutoRecognitionWait**.

Additional channel parameters, as appropriate to the outcome, may then be set by calls on **SetParameter**. The value **illegal** is returned if **lineType** has not been set to **autoRecognition**.

Note: The auto recognition facility is not currently supported. A call to **AutoRecognitionWait** will always result in a outcome of **illegal**.

6.5.2 Environment

This interface defines the environment of the RS232C channel. This includes all the parameters of the line.

6.5.2.1 Types and constants

RS232CEnvironment.AutoRecognitionOutcome: TYPE = RECORD [[0..15]];

AutoRecognitionOutcome defines the range of possible results of the call to **RS232CCorrespondents.AutoRecognitionWait**. See §6.5.1 for the specific outcomes.

RS232CEnvironment.CharLength: TYPE = [5..8];

This type defines the possible number of bits in a character. It pertains only to the data bits, and does not include start, stop or parity bits.

RS232CEnvironment.CommParamHandle: TYPE = POINTER TO RS232C.CommParamObject;

RS232CEnvironment.CommParamObject: TYPE = RECORD [
 duplex: RS232C.Duplexity,
 lineType: RS232C.LineType,
 lineSpeed: RS232C.LineSpeed,
 accessDetail: SELECT **netAccess**: RS232C.NetAccess FROM
 directConn => NULL,
 dialConn => [
 dialMode: RS232C.DialMode,
 dialerNumber: CARDINAL,
 retryCount: RS232C.RetryCount],
 ENDCASE];

When an RS232C channel is created, it is necessary to specify a number of channel parameters. These parameters are supplied by means of **CommParamObject**. Additional characteristics of the channel are generally specified by calls to **RS232C.SetParameter** subsequent to the call to **Create**.

duplex A half duplex line or a full duplex line is specified by **duplex**.

lineType The **lineType** specifies the line type parameter necessary for creating the channel. It serves to define some general characteristics of the channel. Its choice is generally dictated by the equipment connected to

the RS232C port. For more detail on the effect of the **lineType** parameter, see section §6.5.3.4 on data transfer.

lineSpeed **lineSpeed** is the line speed and its choice is dictated by the modem.

accessDetail The **accessDetail** is the variant of the record that describes whether the network is the DDD network or a direct line network. For the dialing network, it determines how the phone is to be dialed and how many times the dial is to be attempted.

RS232CEnvironment.Duplexity: TYPE = {full, half};

Duplexity defines the line as being full duplex or half duplex.

RS232CEnvironment.CompletionHandle: TYPE [2];

The **CompletionHandle** identifies an action initiated by a **RS232C.Get** or **RS232C.Put**. Each **CompletionHandle** must eventually be passed to a **RS232C.TransferWait** or **RS232C.TransmitNow** operation, which does not return until that particular activity is completed or aborted.

RS232CEnvironment.Corrrespondent: TYPE = RECORD [[0..255]];

This type defines the range of correspondents. For specific correspondents, see §6.5.1.

RS232CEnvironment.DialMode: TYPE = {manual, auto};

DialMode defines how the phone is to be dialed.

RS232CEnvironment.FlowControl: TYPE = MACHINE DEPENDENT RECORD [

type(0): {none, xOnXOff},

xOn(1), xOff(2): UNSPECIFIED};

FlowControl specifies the flow control possibilities.

Note: Flow control on the channel is currently not implemented.

RS232CEnvironment.LineSpeed: TYPE = {

bps50, bps75, bps110, bps134p5, bps150, bps300, bps600, bps1200, bps2400,
bps3600, bps4800, bps7200, bps9600, bps19200, bps28800, bps38400, bps48000,
bps56000, bps57600};

The **LineSpeed** defines the speed of the line. The choice is dictated by the modem.

RS232CEnvironment.LineType: TYPE = {

bitSynchronous, byteSynchronous, asynchronous, autoRecognition};

The **LineType** defines whether the line is **bitSynchronous**, **byteSynchronous** or **asynchronous**. A special line type of **autoRecognition** means the RS232C channel will attempt to determine as much as possible about the correspondent at the other end of the communication line. (For more detail on the effect of the line type, see the discussion following **PhysicalRecord**.)

RS232C.NetAccess: TYPE = {directConn, dialConn};

The **NetAccess** specifies the options for the connection types. It is used in the **CommParamObject**.

RS232C.nullLineNumber: CARDINAL = LAST [CARDINAL];

Used with the Pilot stateless enumerator **RS232C.GetNextLine**, **nullLineNumber** defines the starting and ending values of the enumeration.

RS232CEnvironment.Parity: TYPE = {none, odd, even, one, zero};

This type defines the parity to be used.

RS232CEnvironment.PhysicalRecordHandle: TYPE = POINTER TO PhysicalRecord;

RS232CEnvironment.PhysicalRecord: TYPE = RECORD [header, body, trailer: Environment.Block];

The unit of information transferred across the RS232C Channel is the **PhysicalRecord**. The **PhysicalRecord** defines a *frame* of data consisting of an integral number of 8-bit bytes in the code set expected by the equipment connected to the RS232C port. The client may handle a frame as contiguous data, or may treat it as having up to three sections (header, body, trailer) which the channel will gather/scatter appropriately.

As it travels between the client's buffers and the communication line, certain elements of the frame are generated or stripped by the channel. Hence, the format of a frame at the interface between Pilot and the client is slightly different from the frame format as shown in the corresponding protocol documentation (e.g., BSC or HDLC). The 8-bit bytes are serialized across the the communication line with the following transformations according to the **LineType** (see §6.5.1.2), as well as the setting of various parameters (see **SetParameter**, §6.5.3.3):

bitSynchronous (HDLC, SDLC, ADCCP): Flag patterns (01111110), and synchronization information are generated (on output) and stripped (on input) by the channel for all frames. Checksum information is generated (output) and checked (on input), *but not stripped*, so the client's input buffer must provide two extra bytes. Zero insertion and removal following "11111" patterns is performed for all frames. On input, end-of-frame is defined by the recognition of a second flag pattern. On output, end-of-frame is defined by the **Put** procedure call.

byteSynchronous Synchronization information is generated (on output) and stripped (on input) by the channel. Checksum information is generated (on output) and checked (on input) *but not stripped*, so the client's input buffer must provide two extra bytes. On input, end-of-frame is determined by the client supplied parameter, **correspondent** (see §6.5.1.5). The channel generates or checks the checksum as implied by the value of this parameter. On output, end-of-frame is defined by the **Put** procedure call. In addition, a parity bit is (optionally) generated (on output) and checked/stripped (on input) by the channel for each byte.

asynchronous (except when **correspondent = ttyHost**): Checksum characters are generated (on output) and checked (on input) *but not stripped* by the channel, so the client's input buffer must provide two extra bytes. On input, end-of-frame is determined by the client supplied parameter, **correspondent** (see §6.5.1.5). The

channel generates or checks the checksum as implied by the value of this parameter. On output, end-of-frame is defined by the **Put** procedure call. In addition, parity and start/stop bits are generated (on output) and checked/stripped (on input) by the channel for each byte.

asynchronous (when **correspondent** = **ttyHost**): No checksum operations are performed. On input, end-of-frame is determined by a client supplied parameter: **frameTimeout** (see §6.5.1.5). On output, end-of-frame is defined by the **Put** procedure call, but has no other meaning. In addition, parity and start/stop bits are generated (on output) and checked/stripped (on input) by the channel for each byte.

RS232CEnvironment.ReserveType: TYPE = {**preemptNever**, **preemptAlways**, **preemptInactive**};

RS232CEnvironment.RetryCount: TYPE = [0..7];

RS232CEnvironment.StopBits: TYPE = [1..2];

RS232CEnvironment.SyncCount: TYPE = [0..7];

RS232CEnvironment.SyncChar: TYPE = **Environment.Byte**;

The following types have been added to support the multiport board and new encoding. They are the types of the new fields in the **RS232C.Parameter**.

RS232CEnvironment.ClockSource: TYPE = {**internal**, **external**};

RS232CEnvironment.EncodeData: TYPE = {**nrz**, **nrzi**, **fm0**, **fm1**};

RS232CEnvironment.IdleState: TYPE = {**mark**, **flag**};

6.5.3 RS232C channel

RS232C: DEFINITIONS . . . ;

The RS232C channel provides the Pilot client with the *lowest level* access to the RS232C controller and its connected equipment. It assumes that the client has some familiarity with *EIA Standard RS-232-C*.

6.5.3.1 Types and constants

RS232C.ChannelHandle: TYPE [2];

The result of a successful **RS232C.Create** is a **ChannelHandle**, which is used for all subsequent channel operations. The handle becomes invalid after executing a **RS232C.Delete**, and subsequent use of it will have undefined results.

RS232C.CharLength: TYPE = **RS232CEnvironment.CharLength**;

RS232C.CommParamHandle: TYPE = **RS232CEnvironment.CommParamHandle**;

RS232C.CommParamObject: TYPE = RS232CEnvironment.CommParamObject;

RS232C.CompletionHandle: TYPE = RS232CEnvironment.CompletionHandle;

RS232C.Correspondent: TYPE = RS232CEnvironment.Correspondent;

RS232C.DeviceStatus: TYPE = RECORD[statusAborted, dataLost, breakDetected, clearToSend, dataSetReady, carrierDetect, ringHeard, ringIndicator, deviceError: BOOLEAN];

The **DeviceStatus** defines the status of the RS232C device. It is accessed via **RS232C.GetStatus** and **RS232C.StatusWait**.

statusAborted This status will normally be FALSE on calls to **RS232C.GetStatus**. However, a call to **RS232C.StatusWait** may return because the channel was suspended, causing **statusAborted** to be set to TRUE.

breakDetected **breakDetected** is applicable only for **lineType = asynchronous**, and indicates that a break was received on the communication line.

clearToSend, dataSetReady, carrierDetect

These statuses correspond to states of circuits from the Data Communications Equipment (DCE) as described in *EIA Standard RS-232-C*. Normally, **dataSetReady** indicates that the data set (modem) is operational and connected to the communication line. **clearToSend** indicates that the data set is prepared to send data. On a full-duplex communication line, **dataSetReady** and **clearToSend** are normally always TRUE following connection establishment, and need to be monitored only as exception conditions. On a half-duplex line, the normal scenario for use of these booleans is as follows: the client sets **requestToSend**, waits (via **RS232C.StatusWait**) until **clearToSend** is set, and then sends data (via **RS232C.Put**). When the client expects to receive data, he must clear **requestToSend**, so that the data set will allow the communication line to operate in the receive direction.

deviceError This status is set TRUE if a non-recoverable "shouldn't happen" hardware or software error has occurred.

RS232C.DialMode: TYPE = {manual, auto};

The **DialMode** specifies the options for dialing used in the **dialConn** net access, used in the **CommParamObject**.

RS232C.Duplexity: RS232CEnvironment.Duplexity;

RS232C.FlowControl: TYPE = MACHINE DEPENDENT RECORD [type(0): {none, xOnXOff}, xOn(1), xOff(2): UNSPECIFIED];

FlowControl defines the type of flow control the channel should perform. Currently, flow control is not implemented.

RS232C.LatchBitClearMask: TYPE = RS232C.DeviceStatus;

Bits **ringHeard**, **dataLost**, and **breakDetected** are called latch bits in that they are set by the channel when the associated condition occurs, but are not cleared by the channel when the condition clears. They remain set to guarantee the client an opportunity to observe them. To clear them, a mask of type **LatchBitClearMask** must be defined, with the booleans corresponding to the proper latch bits turned on.

RS232C.LineSpeed: TYPE = RS232CEnvironment.LineSpeed;

RS232C.LineType: TYPE = RS232CEnvironment.LineType;

RS232C.NetAccess: TYPE = RS232CEnvironment.NetAccess;

RS232C.nullLineNumber: RS232CEnvironment.nullLineNumber;

RS232C.Parity: TYPE = RS232CEnvironment.Parity;

RS232C.OperationClass: TYPE = {input, output, other, all};

The **OperationClass** specifies the different classes of operations which may be aborted by **RS232C.Suspend**. **input** consists of the **Get** operation only, **output** is **Put** and **SendBreak**, **other** is **GetStatus** and **StatusWait**. If the client wishes to abort all the operations, he may use the **all** option.

```
RS232C.Parameter: TYPE = RECORD [SELECT type: RS232C.ParameterType FROM
    charLength => [charLength: RS232C.CharLength],
    clockSource => [clockSource: RS232C.ClockSource],
    correspondent => [correspondent: RS232C.Correspondent],
    dataTerminalReady => [dataTerminalReady: BOOLEAN],
    echoing => [echoing: BOOLEAN],
    encodeData => [encodeData: RS232C.EncodeData],
    flowControl => [flowControl: RS232C.FlowControl],
    frameTimeout => [frameTimeout: CARDINAL],
    idleState => [idleState: RS232C.IdleState],
    latchBitClear => [latchBitClearMask: RS232C.LatchBitClearMask],
    lineSpeed => [lineSpeed: RS232C.LineSpeed],
    maxAsyncTimeout => [maxAsyncTimeout: CARDINAL],
    parity => [parity: RS232C.Parity],
    requestToSend => [requestToSend: BOOLEAN],
    stopBits => [stopBits: RS232C.StopBits],
    syncChar => [syncChar: RS232C.SyncChar],
    syncCount => [syncCount: RS232C.SyncCount],
    ENDCASE];
```

```
RS232C.ParameterType: TYPE = {charLength, correspondent, dataTerminalReady, echoing,
    flowControl, frameTimeout, latchBitClear, lineSpeed, parity, requestToSend, stopBits,
    syncChar, syncCount};
```

The **RS232C.Parameter** contains the following additional channel parameters:

charLength	The number of data bits in a character is specified by charLength . The number of bits, right justified, are removed from and stored into the 8-bit bytes described by RS232C.PhysicalRecord . Remaining bits are ignored on Put operations, and set to zero on Get operations.
clockSource	The source of the clock (from internal baud rate generator or from the external source)--default for asynchronous is internal , default for bit synchronous and byte synchronous is external .
correspondent	correspondent is the type of correspondent the client is communicating with, which is used to set certain channel characteristics. See §6.5.1.1 for the legal RS232CCorrespondents .
dataTerminalReady	This parameter corresponds to the state of the DTR circuit to the Data Communications Equipment (DCE). It should be set by the client as described in <i>EIA Standard RS-232-C</i> . Normally, dataTerminalReady is set to FALSE when the client wishes to disconnect the communication line.
echoing	echoing specifies whether echoing of input characters should be done by the RS232C channel. If echoing is TRUE , all input characters received will be echoed by the RS232C channel. If it is FALSE , the client using the RS232C channel is responsible for echoing input characters.
encodeData	A parameter that may be used with SDLC. The default for any line type is nrz .
flowControl	flowControl specifies whether the channel should perform flow control. If type is xOnXOff , the RS232C channel will stop output when it receives an xOff character and resume output when it receives an xOn character. Note: flowControl is currently not implemented.
frameTimeout	The intra-frame timeout in milliseconds is specified by frameTimeout . On input, for all settings of parameter correspondent other than ttyHost , if the last byte of a frame does not arrive within frameTimeout milliseconds of the first byte, the frame will complete abnormally with status equal to frameTimeout . If the correspondent is set to ttyHost , then once the first byte of the frame arrives, if the next byte does not arrive within frameTimeout milliseconds, the frame will complete normally. Setting frameTimeout to zero is equivalent to setting an infinite frame timeout.
idleState	The state of the line when it is idle; i.e., whether to transmit flags or mark . The default for bit synchronous is

flag; default for byte synchronous and asynchronous is **mark**.

latchBitClear The mask used for clearing the latch bits of the **RS232c.DeviceStatus** is defined by **latchBitClear**. Only the latch bits which are set in this mask will be cleared.

lineSpeed The speed of the line is defined by **lineSpeed**. Its choice is dictated by the modem.

maxAsyncTimeout Used in conjunction with **frameTimeout** for multiport asynchronous frame timing--default is 0 (infinite timeout).

parity **parity** specifies the type of parity to be used.

requestToSend **requestToSend** corresponds to the state of the RTS circuit to the Data Communications Equipment (DCE). It should be set by the client as described in *EIA Standard RS-232-C*. For full-duplex communication lines, it should remain **TRUE** at all times. For half-duplex lines, it is used to control line turnaround. (See §6.5.3.1 for details).

stopBits specifies the number of stop bits to use on the channel when **lineType** is **asynchronous**.

syncChar specifies the synchronization character which the channel will transmit at the beginning of each frame when **lineType** is **byteSynchronous**. On input, synchronization characters preceding frames are discarded.

syncCount is the number of synchronization characters which the channel will transmit at the beginning of each frame when **lineType** is **byteSynchronous**. On input, synchronization characters preceding frames are discarded.

Not all parameters nor all syntactically legal parameter values are valid for all **LineTypes**. The following chart shows the valid values (as well as the default values) following calls to **Create** or **SetLineType**.

Valid and Default Parameter Settings

	<u>asynchronous</u>	<u>byteSynchronous</u>	<u>bitSynchronous</u>
charLength	any ¹ (8)	7,8 ³ (8)	any (8)
correspondent	xerox800, ttyHost (xerox800)	xerox850, system6, cmcl (system6) siemens9750	nsSystemElement (nsSystemElement
dataTerminalReady⁴	any (FALSE)	any (FALSE)	any (FALSE)

echoing	invalid ²	invalid	invalid
flowControl	invalid	invalid	invalid
frameTimeout	any (infinite)	any (infinite)	any (infinite)
lineSpeed	any (bps1200)	any (bps1200)	any (bps1200)
parity	any (none)	any (none)	any (none)
requestToSend⁴	any (FALSE)	any (FALSE)	any (FALSE)
stopBits	any (1)	invalid	invalid
syncChar	invalid	any (62B)	invalid
syncCount	invalid	any (2)	invalid

1. "any" means any syntactically-accepted value is valid.

2. "invalid" means either the parameter is ignored, error **RS232C.UnimplementedFeature** or error **RS232C.InvalidParameter** will be generated. See §6.5.3.2 for more information on these errors.

3. **charLength = 8** with **parity = none** is valid, and **charLength = 7** with **parity#none** is valid. All other combinations are invalid.

4. Default values are set following calls to **RS232C.Create**. Values are unchanged following calls to **RS232C.SetLineType**.

RS232C.StopBits: TYPE = **RS232CEnvironment.StopBits**;

RS232C.PhysicalRecordHandle: TYPE = **RS232CEnvironment.PhysicalRecordHandle**;

RS232C.PhysicalRecord: TYPE = **RS232CEnvironment.PhysicalRecord**;

RS232C.ReserveType: TYPE = {**preemptNever**, **preemptAlways**, **preemptInactive**} ;

The **ReserveType** is used to establish priority among clients contending for a line during a call to **RS232C.Create**. **preemptNever** is used by clients who wish to never attempt to gain ownership of a line already being used. **preemptAlways** is used to always attempt to gain ownership of such a line, and clients using **preemptInactive** will attempt to gain ownership only if the current channel is not active.

RS232C.TransferStatus: TYPE = {**success**, **dataLost**, **deviceError**, **frameTimeout**, **checksumError**, **parityError**, **asynchFramingError**, **invalidChar**, **invalidFrame**, **aborted**, **disaster**} ;

TransferStatus describes the status of an individual data transfer (i.e., **Get** or **Put**). It is returned to the client as the result of the **TransferWait** or **TransmitNow** procedure.

success **success** is the status returned normally, when the data transfer has successfully completed.

dataLost This status will occur when a **PhysicalRecord** for a **Get** operation is not large enough to accommodate the arriving frame. The channel will discard all overflow data bytes until end-of-frame is detected.

deviceError This status indicates the transfer should be considered successful, but a non-recoverable "shouldn't happen" hardware or software error has occurred.. Note that such status changes will cause the completion of any pending **RS232C.StatusWait** call (see §6.5.3.5). The **dataLost** latch bit will be set in the **DeviceStatus** record if data arrives when no **PhysicalRecord** has been allocated via a **Get** operation, and **deviceError** will be returned as the **TransferStatus** on all data transfer operations until the **dataLost** latch bit is cleared.

frameTimeout **frameTimeout** is set if the last byte of a frame does not arrive within the timeout specified in the **frameTimeout** parameter in **RS232C.Parameter**.

checksumError, parityError, asynchFramingError

These states all imply that the data has not been transferred faithfully (i.e., stop bits are missing).

invalidChar, invalidFrame, disaster

These states are not implemented.

aborted **aborted** will occur if **RS232C.Suspend** is called while the data transfer is outstanding.

6.5.3.2 Signals and errors

RS232C.ChannelInUse: ERROR;

If the channel is active and reservation (pre-emption) fails, this error is generated.

RS232C.ChannelSuspended: ERROR;

After doing a **RS232C.Suspend** on a certain class of operations, a call to any operation in that class will result in the **ChannelSuspended** error being raised.

RS232C.InvalidLineNumber: ERROR;

InvalidLineNumber is generated when the **lineNumber** supplied to the **Create** procedure is invalid.

RS232C.InvalidParameter: ERROR;

Generated by **RS232C.Create** or **RS232C.SetParameter**, this error indicates the client specified an invalid channel parameter.

RS232C.SendBreakIllegal: ERROR;

This error is raised when a client attempts to call the **SendBreak** procedure on a channel with a line type of **byteSynchronous**.

RS232C.NoRS232CHardware: ERROR;

This error indicates the **Create** procedure has been called with no RS232C hardware installed.

RS232C.UnimplementedFeature: ERROR;

UnimplementedFeature may be raised by a call to **SetParameter**, **SetLineType**, or **Create**.

6.5.3.3 Procedures for creating and deleting channels

```
RS232C.Create: PROCEDURE [lineNumber: CARDINAL,
    commParams: RS232C.CommParamHandle, preemptMe: RS232C.ReserveType]
RETURNS [channel: RS232C.ChannelHandle];
```

Each RS232C channel is a non-shareable resource that supports one full-duplex communication path. A channel is potentially contended for by Communication software and by Pilot clients accessing foreign devices. The RS232C channel resolves contention for and supports pre-emptive allocation. Clients call the **Create** procedure to reserve a channel. The channel handle returned is then used in all subsequent operations. If this procedure is called when no RS232C hardware is installed, the error **RS232C.NoRS232CHardware** will be raised.

lineNumber

The **lineNumber** specifies the RS232C line to use, which may be obtained using the **GetNextLine** procedure. If **lineNumber** does not represent a line present on the RS232C controller, the error **RS232C.InvalidLineNumber** will be raised.

preemptOthers, preemptMe

These parameters serve to establish priority among contending clients. The state of a channel will be either inactive (available or waiting for a connection) or active. If a channel is available then a **Create** will always succeed. Otherwise, the success of the **Create** depends on the relative priorities of the current "owner" of the channel and the client trying to reserve it. If the channel is active and reservation (pre-emption) fails, the error **RS232C.ChannelInUse** is generated. The following matrix defines the result of a **Create** given the values of the owner's **preemptMe** and the reserver's **preemptOthers**.

		Owner's preemptMe		
		Never	If Inactive	Always
Reserver's preempt- Others	Never	Fail	Fail	Fail
	If Inactive	Fail	Pre-empt*	Pre-empt
	Always	Fail	Pre-empt	Pre-empt

* Pre-empt if inactive

A new reservation that is waiting for a connection has a grace period starting when **Create** is called and ending after a certain time interval, during which it is not considered to be inactive. During this time it is not pre-emptable by a client specifying a **preemptOthers** equal to **preemptInactive**. This is necessary to prevent thrashing of contending listening clients who specify **preemptMe** equal to **preemptInactive**.

Caution: The grace period after a **Create** referred to above is not implemented in this version of Pilot.

It is the responsibility of the client who called **RS232C.Create** to release the channel when the channel is pre-empted, or it is no longer required. Pre-emption is detected by noticing that all **RS232C** calls return a status of **aborted**. The pre-emption algorithm assumes that the channel owner will notice this, and cooperate by releasing the channel by doing a **RS232C.Delete**.

commParams

commParams specifies the basic channel characteristics.

RS232C.SetParameter: **PROCEDURE** [channel: **RS232C.ChannelHandle**, parameter: **RS232C.Parameter**];

Additional channel parameters may be set by calling **SetParameter**.

RS232C.Delete: **PROCEDURE** [channel: **RS232C.ChannelHandle**];

This operation has the effect of calling **RS232C.Suspend**, aborting all pending activity on the channel. Any incomplete asynchronous activities (i.e., those initiated via **Get** or **Put**) will be terminated immediately with **status** = **aborted**. Note that it is the client's responsibility to call **TransferWait** or **TransmitNow** for each of these asynchronous activities in order for the call to **Delete** to complete. In general, this means that the **Delete** and the **TransferWaits** must be issued from separate processes. If the client wishes to terminate all pending activities normally, he should complete a call to **RS232C.TransferWait** or **RS232C.TransmitNow** for each pending activity before calling **Delete**. Upon return from the call to **Delete**, the **ChannelHandle** is invalid, and further calls using this handle will have undefined results. One convenient way to idle-down the channel is to set flags for all processes which have access to the **ChannelHandle**, call **RS232C.Suspend[all]**, and then **JOIN** these processes prior to calling **Delete**. The assumption is that any process receiving an **aborted** status on an **RS232C** operation will check the flags and terminate.

6.5.3.4 Data transfer procedures

The operations described below transmit information to and from the equipment connected to the RS232C port.

RS232C.Get: PROCEDURE [channel: RS232C.ChannelHandle,
rec: RS232C.PhysicalRecordHandle] RETURNS [RS232C.CompletionHandle];

The **Get** operation queues the **PhysicalRecord** for input transfer and returns to the client with the input transfer pending. The handle obtained via the **RS232C.Create** procedure is specified by **channel**. **rec** is the input buffer for the incoming data frame.

RS232C.Put: PROCEDURE [channel: RS232C.ChannelHandle,
rec: RS232C.PhysicalRecordHandle] RETURNS [RS232C.CompletionHandle];

The **Put** operation queues the **PhysicalRecord** for output transfer and returns to the client with the output transfer pending. Both **Get** and **Put** are *asynchronous*, in the sense that they return to the caller as soon as the request has been queued, but complete at a later time. For each direction (i.e., input and output), pending activities are processed and completed in the order in which they are issued. The returned **CompletionHandle** identifies an activity initiated by a **Get** or **Put** operation. Each **CompletionHandle** must eventually be passed as a parameter to the **TransferWait** or **TransmitNow** operation, which does not return until that particular activity is completed or aborted.

channel The handle obtained via the **RS232C.Create** procedure is specified by **channel**.

rec **rec** is the output buffer for the frame of data to be sent.

The I/O buffers described by the **PhysicalRecord** must not be released, altered, or re-used until after the **TransferWait** or **TransmitNow** operation for the associated transfer completes.

RS232C.TransferWait: PROCEDURE [channel: RS232C.ChannelHandle,
event: RS232C.CompletionHandle]
RETURNS [byteCount: CARDINAL, status: RS232C.TransferStatus];

Forking a process to perform a **TransferWait** allows the client program to proceed with parallel processing.

channel **channel** is the handle obtained via the **RS232C.Create** procedure.

event **event** is the completion handle that identifies the activity upon which to wait. (i.e., the handle returned from the **Get** or **Put** data transfer operation.)

byteCount The number of data bytes transferred upon completion of the call is specified by **byteCount**.

status The status of the completed call is specified by **status**. See §6.5.3.1 for the different status values that may be returned.

TransferWait awaits completion of the activity initiated by **Get** or **Put** and returns to the client the number of bytes transferred and the status. For **Puts**, the return from this procedure indicates that the client's buffers are available for reuse, but does not guarantee that the associated data has been transmitted on the communication line.

```
RS232C.TransmitNow: PROCEDURE [channel: RS232C.ChannelHandle,
    event: RS232C.CompletionHandle]
    RETURNS [byteCount: CARDINAL, status: RS232C.TransferStatus];
```

Instead of **TransferWait**, **TransmitNow** may be used to force **Put** operations to complete. Return from this procedure guarantees that the data has been transmitted on the communication line.

```
RS232C.Suspend: PROCEDURE [channel: RS232C.ChannelHandle,
    class: RS232C.OperationClass];
```

This procedure aborts all pending activity of the specified **OperationClass** and causes subsequent calls to generate the error **RS232C.ChannelSuspended** until a call to **Restart** is issued.

Suspend does not return until the abort of all pending activities of the specified **OperationClass** is complete. In the case of asynchronous operations it is the client's responsibility to call **TransferWait** or **TransmitNow** for each of these operations in order for the call to **Suspend** to complete. In general, this means that the **Suspend** and the **TransferWait** must be issued in separate processes. Since **Delete** and **SetLineType** have the effect of a call to **Suspend**, this is also true of calls to them.

```
RS232C.Restart: PROCEDURE [channel: RS232C.ChannelHandle,
    class: OperationClass];
```

This operation clears the effect of a call to **Suspend**. A suspend may occur as a result of an explicit **Suspend** operation or as a result of the occurrence of a sufficiently serious error.

6.5.3.5 Utility procedures

```
RS232C.GetNextLine: PROCEDURE [lineNumber: CARDINAL]
    RETURNS [nextLine: CARDINAL];
```

RS232C line numbers may be obtained by the **GetNextLine** procedure, a Pilot stateless enumerator with starting and ending values of **nullLineNumber**.

```
RS232C.GetStatus: PROCEDURE [channel: RS232C.ChannelHandle]
    RETURNS [stat: RS232C.DeviceStatus];
```

In addition to the status information returned for each data transfer operation, Pilot maintains global information about the device itself. It is accessed via the **GetStatus** and **StatusWait** procedures. **GetStatus** returns the current status of the device.

```
RS232C.StatusWait: PROCEDURE [channel: RS232C.ChannelHandle,
    stat: RS232C.DeviceStatus] RETURNS [newstat: RS232C.DeviceStatus];
```

StatusWait waits until the current **DeviceStatus** differs significantly from the supplied parameter **stat**. Changes in status to **statusAborted**, **dataLost**, **breakDetected**, **clearToSend**, **dataSetReady**, **carrierDetect** and **ringHeard** are defined to be significant. A change to **ringIndicator** is not. The client must examine the device status to determine what action to take.

RS232C.SetLineType: PROCEDURE [channel: RS232C.ChannelHandle,
lineType: RS232C.LineType];

SetLineType is used to change the **LineType** subsequent to creating the channel. Note that the process of deleting a channel and then creating it again has the effect of setting **dataTerminalReady** to FALSE, thereby hanging up a switched telephone line connected to the modem. A call to **SetLineType** does not have this effect.

The **SetLineType** operation has the effect of a call to **RS232C.Suspend**. If the client wishes to complete all pending activities normally, he should first call **RS232C.TransferWait** for each pending activity, prior to calling **SetLineType**. Parameter information (as supplied via prior calls to **SetParameter**) is reset to default values (see chart below), and should be resupplied.

RS232C.SendBreak: PROCEDURE [channel: RS232C.ChannelHandle];

SendBreak transmits a break on the communication line, where break is defined to be the absence of a "stop" bit for more than 190 milliseconds if **lineType** equals **asynchronous**, or an abort (between 7 and 14 "1" bits) if **lineType** equals **bitSynchronous**. **SendBreak** is illegal for **lineType** equal to **byteSynchronous**, and will result in the error **RS232C.SendBreakIllegal**. Note that sending a break while data transfer operations are outstanding has unpredictable results.

6.5.4 Procedures for starting and stopping the channel

RS232CControl: DEFINITIONS . . . ;

The **RS232CControl** interface allows the client to start and stop the RS232C channel code. When the configuration RS232CIO is started, the channel code is started.

RS232CControl.Stop: PROCEDURE [suspendActiveChannels: BOOLEAN];

Stop stops the RS232C channel code. No new channel creations are allowed and any attempt to create a channel results in the error **RS232C.NoCommunicationHardware**. In addition, if **suspendActiveChannels** is **TRUE**, all channels are suspended.

RS232CControl.Start: PROCEDURE;

Start allows channel creation after a **Stop** call.

RS232C.NoCommunicationHardware: ERROR;

This error is raised when a client attempt to create a channel after **RS232C.Stop** has been called.

6.5.5 Auto-dialing

Dialup: DEFINITIONS . . . ;

The **Dialup** interface allows the client to specify a telephone number for the auto-dialing hardware to dial. Upon successful completion of the dialing operation, data transfers across the associated RS232C channel will be directed to the equipment answering the

telephone call. (Note that the hardware association between modems and dialers is configured by the user, and is assumed to be known to the client of the **Dialup** and **RS232C** interfaces.) To cause a telephone number to be dialed, the client calls

```
Dialup.Dial: PROCEDURE [dialerNumber: CARDINAL, number: LONG POINTER TO Number,
    retries: RS232C.RetryCount,dialerType: Dialup.DialerType] RETURNS [Dialup.Outcome];
```

```
Dialup.Number: TYPE = RECORD [number: PACKED SEQUENCE n: CARDINAL OF Environment.Byte];
```

```
Dialup.RetryCount: TYPE = [0..7];
```

```
Dialup.Outcome: TYPE = { success, failure, aborted, formatError, transmissionError,
    dataLineOccupied, dialerNotPresent, dialingTimeout, transferTimeout };
```

```
Dialup.DialerType: TYPE = {RS366, Ventel, smartmodem, other};
```

dialerNumber specifies a logical dialer number corresponding to a physical dialer attached to a port either on the local processor or the *Xerox 872/873 Communication Server*. Dialing operations will also require some form of logical identifier to distinguish among multiple modems serviced by the same dialer.

number is a sequence of bit patterns representing the digits to be dialed. With the exception on the **pause**, the dialup implementation attaches no semantics to any of the bit patterns it receives - they are simply passed to the dialer hardware. It is the responsibility of the client to know what bit patterns represent special characters such as EON and SEP for his particular hardware. The Modem then has the responsibility for detecting Answer Tone. In the absence of the EON digit, transfer is made automatically upon detection and processing of Answer Tone.

retries indicates how many times the Dialup routine will retry the dialing operation following **failure** outcomes (see below).

The values of **Outcome** are to be interpreted as follows:

success

The dialing operation was successful. For dialers capable of detecting answer tone, this means that the call was answered by a compatible modem and control was successfully transferred by the dialer to the associated local modem. For dialers not so equipped (i.e., when EON is used to control transfer to the modem), this means that all the digits in the **number** were dialed, and control was successfully transferred to the modem. Note that **success** refers to the dialing operation, and *should not* be taken to mean that the associated modem is ready to transfer data. This should be determined by examining **DataSetReady** and **clearToSend** in the **RS232C.DeviceStatus**.

failure

The dialing operation resulted in no answer, a busy signal, or the telephone was answered by something other than a compatible modem (e.g., a human being).

aborted

The dialing operation was aborted (via **Dialup.AbortCall**).

formatError	The parameter number was formatted incorrectly.
transmissionError	The transfer of the dialing information to the dialing hardware did not succeed. This should not happen in normal operation, and indicates a hardware problem which should be investigated.
dataLineOccupied	The telephone line to which the dialing hardware is connected was off-hook. This situation indicates an operational problem which should be investigated.
dialerNotPresent	The dialer hardware did not respond. This situation indicates a hardware problem (or a <i>lack-of-hardware</i> problem) which should be investigated.
dialingTimeout	The dialer did not respond to a request during dialing. This situation indicates a hardware problem which should be investigated.
transferTimeout	No meaningful reply was received from the dialer following dialing the last digit. The dialer neither detected a failure (i.e., busy or not answer) nor successfully transferred control to the modem. This situation indicates a hardware problem which should be investigated.

Dialup.AbortCall: PROCEDURE [dialerNumber: CARDINAL];

If the client wishes to abort the dialing operation (from another process) prior to the return from Dial, he may call **AbortCall**, causing the call to Dial to return with an **Outcome** of failure.

Dialup.GetDialerCount: PROCEDURE RETURNS [numberOfDialers: CARDINAL];

The procedure **GetDialerCount** returns the total number of RS-366 (dialer) ports available.

Dialup.pause: Environment.Byte = LAST[Environment.Byte];

When passed in the parameter **number**, **pause** causes the dialer to wait 6 seconds before dialing subsequent digits. **pause** is designed to be used in place of SEP on dialers that cannot detect Dial Tone. This bit pattern does not actually get passed to the dialer hardware.

6.6 Courier

The term *remote procedure calling* refers to a software framework that facilitates the design, implementation and documentation of distributed services. Remote procedure calling casts the network protocols that underlie distributed services into a model closely resembling the invocation of procedures in nondistributed programs. Thus a client request for some service resembles a procedure call, and the information returned by the service resembles the return from a procedure.

Mesa clients of Courier are provided with a set of facilities that closely parallel those provided by Mesa running on a single machine. Just as Mesa provides powerful facilities for modelling and controlling the interaction of programs through type-safety and signals, Courier provides facilities for modelling and controlling the interaction of systems distributed among an arbitrary number of machines. The principal limitation of Courier is that it supports only a subset of the Mesa data types.

6.6.1 Definition of terms

The following terms are used throughout this section and have specific meanings in the Courier context.

<i>disjoint data</i>	In general, any Mesa structure that causes Courier to access data outside the current parameter area is disjoint. Examples of disjoint data are StringBodys and ARRAYS described by LONG DESCRIPTORS .
<i>parameter area</i>	A segment of contiguous virtual memory that contains Mesa data types and is being processed by a description routine. Parameter areas are defined by a LONG POINTER and a size.
<i>remote program</i>	A remote program usually represents a complete <i>service</i> , and the remote procedures it contains represent the <i>operations</i> of that service.
<i>RPC</i>	This is an acronym for <i>remote procedure call</i> . In this context it refers to the actual processing of arguments and results, that function being separate from <i>bulk data</i> , for instance.
<i>server</i>	The Courier server is the Mesa Courier client that provides or exports a service.
<i>transport</i>	The transport is used by Courier to carry the remote procedure call messages and by clients to carry bulk data. The transport is in the form of a Pilot stream, and is usually assumed to be a Network stream.
<i>user</i>	The Courier user is the Mesa Courier client that requests, consumes or imports a service.

6.6.2 Binding

Courier provides mechanisms for *late binding* at both the user and server machine. The mechanisms are less rigorous than their Mesa counterpart, but they do exist.

6.6.2.1 Binding to a service

Binding at the user machine is controlled by the procedures **Courier.Create** and **Delete** and the existence of a valid **Courier.Handle**. The handle (and thus the binding) returned by **Create** remains valid on the machine it was created until it is deleted or the system is restarted.

```
Courier.SystemElement: TYPE = System.NetworkAddress;  
  
Courier.Handle: TYPE = LONG POINTER TO READONLY Courier.Object;  
  
Courier.Object: TYPE = RECORD[remote: Courier.SystemElement,  
    programNumber: LONG CARDINAL, versionNumber: CARDINAL,  
    zone: UNCOUNTED ZONE, SH: Stream.Handle,  
    classOfService: NetworkStream.ClassOfService];  
  
Courier.Create: PROCEDURE [remote: Courier.SystemElement,  
    programNumber: LONG CARDINAL, versionNumber: CARDINAL,  
    zone: UNCOUNTED ZONE, classOfService: NetworkStream.ClassOfService]  
RETURNS [Courier.Handle];  
  
Courier.Delete: PROCEDURE [cH: Courier.Handle];  
  
Courier.Error: ERROR [errorCode: Courier.ErrorCode];  
  
Courier.ErrorCode: TYPE = {..., invalidHandle, ...};
```

Successful completion of the **Create** procedure results in the returning of a **Courier.Handle**. The holder of that handle is then declared to be *bound* to the remote service specified. The service in turn is specified as a concatenation of a **Courier.SystemElement**, a *remote program number* and a desired *version*. **Courier.Create** also records other (interesting) aspects of the client's access to the remote service, namely the **UNCOUNTED ZONE** to be used for storing disjoint data structures, and an indication of the type of transport needed to effectively communicate with the service.

Note: **Create** merely records the request for binding locally. Thus it may not do all the checking that one would expect. The first attempt to establish a dialogue with the remote service, hence completing the binding, is not made until the first **Courier.Call** (see §6.6.3).

Note: The transport used by Courier for communication with the remote machine is (usually) under full control of Courier itself. The transport may be shared by other Courier clients, created or deleted at Courier's discretion. Therefore the binding *does not* include the transport (except when it is being used by bulk data transfer (see §6.6.5)).

The success of **Create** results in the caller possessing a **Courier.Handle**, and, indirectly, the **Courier.Object** to which it points. The only information in the object not provided by the client is a **Stream.Handle**, used by **BulkData** (see §6.6.5). The possession of the **Courier.Handle** entitles the holder to make procedural requests, one at a time, of a remote service. The handle remains valid until explicitly deleted (**Courier.Delete**). Once deleted, the handle is void and may not be used in any operation (including **Delete**) again. Attempts to use an invalid handle will result in the signal **Courier.Error[invalidHandle]** being raised.

Note: **Delete** doesn't delete the transport immediately. Courier will retain the transport for some period of time hoping another client will be able to use it, thus eliminating the overhead of deleting and creating the transport. Courier goes to great pains to (properly) utilize the transport, which is perceived to be very heavy-weight relative to the needs of most RPC operations. Delayed creates, reusing existing transports, and delayed deletes are all attempts to optimize the transport's use. Regrettably, it also reduces the debuggability by at least an order of magnitude.

6.6.2.2 Server binding

```
Courier.ExportRemoteProgram: PROCEDURE [
    programNumber: LONG CARDINAL, versionRange: Courier.VersionRange,
    dispatcher: Courier.Dispatcher, serviceName: LONG STRING ← NIL,
    zone: UNCOUNTED ZONE, classOfService: NetworkStream.ClassOfService];

Courier.VersionRange: TYPE = RECORD [low, high: CARDINAL];

Courier.Dispatcher: TYPE = PROCEDURE [
    cH: Courier.Handle, procedureNumber: CARDINAL,
    arguments: Courier.Arguments, results: Courier.Results];

Courier.Arguments: TYPE = ...

Courier.Results: TYPE = ...

Courier.ErrorCode: TYPE = {..., duplicateProgramExport, ...};

Courier.NoSuchProcedureNumber: ERROR;
```

In order to make a service available on a machine, the server client must first register that service (export) via **ExportRemoteProgram**. That action provides a template needed by Courier to complete the binding process as is it needed. It registers information about the service (*program number, version range, class of transport* and the **UNCOUNTED ZONE**) much like **Courier.Create**. One difference is that the client specifies a *version range* when registering the service. This permits servers to provide backwards compatibility by allowing a single export to support any number of versions. Courier uses the information provided by the call as information to fabricate **Courier.Handles** (and the **Courier.Objects** behind them). Each handle thus created may be treated exactly as a handle returned by **Courier.Create**, with the exception that the lifetime of the handle is defined by Courier. The object is not created by a client and therefore should not be deleted by the client. It is assumed void when the client returns from his *dispatcher*.

Courier will signal duplicate program exports (identical program number and version range) by raising **Courier.Error[duplicateProgramExport]**. Be aware that duplicate exports require an *exact* match. Registering a secondary export with overlapping version ranges will succeed, but will give non-deterministic results.

The active part of the exported service is the client's *dispatcher*. Courier calls this procedure from a **FORKED** process and in no way serializes incoming requests. The dispatcher is client-implemented and is responsible for the final stage of binding at the server machine. The last element in the binding process is the **procedureNumber**. The client must verify that the procedure is really exported by the service, and if it is not, it

should signal **Courier.NoSuchProcedureNumber**, thus rejecting the call. If the procedure number is valid, the service should proceed with the argument processing and perform the defined service.

```
Courier.UnexportRemoteProgram: PROCEDURE [
    programNumber: LONG CARDINAL, versionRange: Courier.VersionRange];

Courier.ErrorCode: TYPE = {..., noSuchProgramExport, ...};
```

Once registered via **Courier.ExportRemoteProgram**, the service is expected to respond to remote requests as the protocol for that service specifies. That responsiveness should continue until **UnexportRemoteProgram** is called. At that time, the service is no longer available and all subsequent requests will be rejected. **UnexportRemoteProgram** will not affect calls currently in progress.

6.6.3 Remote procedure calling

The major purpose of Courier is to provide a simple remote procedure call facility. It endeavors to relieve the client of many of the communications aspects of providing a remote service, leaving a call model that can be likened to Mesa in many respects.

6.6.3.1 Client call

```
Courier.Call: PROCEDURE [
    cH: Courier.Handle, procedureNumber: CARDINAL,
    arguments, results: Courier.Parameters ← Courier.nullParameters,
    timeoutInSeconds: LONG CARDINAL ← LAST[LONG CARDINAL],
    requestDataStream: BOOLEAN ← FALSE,
    streamCheckoutProc: PROCEDURE[cH: Courier.Handle] ← NIL]
RETURNS [sH: Stream.Handle];
```

Courier.Parameters: TYPE = RECORD [location: LONG POINTER, description: Courier.Description];

Courier.nullParameters: Courier.Parameters = [NIL, NIL];

Courier.Description:;

```
Courier.ErrorCode: TYPE = {...,
    transmissionMediumHardwareProblem, transmissionMediumUnavailable,
    transmissionMediumNotReady, noAnswerOrBusy, noRouteToSystemElement,
    transportTimeout, remoteSystemElementNotResponding, noCourierAtRemoteSite,
    tooManyConnections, invalidMessage, noSuchProcedureNumber, returnTimedOut,
    callerAborted, unknownErrorInRemoteProcedure, streamNotYours,
    parameterInconsistency, invalidArguments, noSuchProgramNumber,
    protocolMismatch, invalidHandle, ...};
```

The basis of Courier's RPC facility is embodied in the **Courier.Call**. **Call** completes the specification of the desired service by merging the binding information (**cH**), a procedure within that generic service (**procedureNumber**) and the parameters (**arguments**) to be supplied to the procedure. Due to the implied distributive nature of the call, the client is requested to provide an estimate of how much time will elapse before a response is declared lost (**timeoutInSeconds**). The remaining two arguments (**requestDataStream** and

`streamCheckoutProc`) are relevant to bulk data transfer (see §6.6.5), as is the `Stream.Handle` returned by the `Call`.

6.6.3.1.1 Call initial processing

```
Courier.ErrorCode: TYPE = {...,  
    transmissionMediumHardwareProblem, transmissionMediumUnavailable,  
    transmissionMediumNotReady, noAnswerOrBusy, noRouteToSystemElement,  
    transportTimeout, remoteSystemElementNotResponding, noCourierAtRemoteSite,  
    tooManyConnections, protocolMismatch, invalidHandle, ...};
```

Initial contact with the remote machine will be made when the *first* call is made to that machine. A transport will be created *before* the **arguments** are processed. That initial contact will not include information about the particular service involved in the procedure call. Thus, it establishes the ability to communicate with the remote machine but does not verify that the desired service is actually exported.

Caution: Due to Courier's transport caching and swapping algorithms, it is almost impossible for a Courier client to tell when such initial contact is being established. Therefore, for the purposes of signal catching and the like, it is prudent to assume *every* `Courier.Call` is an initial contact.

6.6.3.1.2 Argument processing

```
Courier.ErrorCode: TYPE = {...,  
    transportTimeout, invalidMessage, noSuchProcedureNumber,  
    parameterInconsistency, invalidArguments, noSuchProgramNumber, invalidHandle,  
    ...};
```

`Courier.VersionMismatch: ERROR [versionRange: Courier.VersionRange];`

`Courier.VersionRange: TYPE = RECORD [low, high: CARDINAL];`

The remote machine will be made aware of the full binding information during or immediately after processing the procedure's **arguments**. Should the binding fail, Courier will raise `Courier.Error` with an appropriate error code (`noSuchProcedureNumber` or `noSuchProgramNumber`) or `Courier.VersionMismatch`. In the case of `VersionMismatch`, the user client is afforded the opportunity to select a version that is implemented by the server and retry the operation.

Courier **Call** parameters are not an exact Mesa model of procedure parameters. As described in §6.6.6, Courier needs help to map Mesa data types to Courier data types. To provide that help, the Courier client must provide the **location** of the parameter area and a **description** routine to describe them in the form of a `Courier.Parameters` record. The Courier equivalent for a Mesa procedure with no arguments or results is a `Courier.Call` that has its **arguments** and **results** parameters assigned (or defaulted) a value of `Courier.nullParameters`.

Note: The **description** routine will not be called if either the **location** or the **description** field of the `Parameters` record is `NIL`.

6.6.3.1.3 Waiting for results

```
Courier.ErrorCode: TYPE = {...,  
    transportTimeout, returnTimedOut, unknownErrorInRemoteProcedure, ...};  
  
Courier.RemoteErrorSignalled: ERROR [  
    errorNumber: CARDINAL, arguments: Courier.Arguments];  
  
Courier.Arguments: TYPE = PROCEDURE [  
    argumentsRecord: Courier.Parameters ← Courier.nullParameters];
```

Once the **arguments** of a **Call** have been successfully transmitted, Courier will not return to the client until it either receives the **results** of the procedure call, receives notification that the call has failed, or abandons the call. During the period while Courier is waiting for the results, the **transport** is *bound* to the call. Should that **transport** fail, the local machine will abandon the call and raise the signal **Courier.Error[transportTimeout]**. Courier watches to insure that the call returns in a client specified period of time (**timeoutInSeconds**). Should that time expire, Courier will raise the error **Courier.Error[returnTimedOut]**.

Caution: **timeoutInSeconds** must be translated to internal units by the underlying software. Should that conversion result in an overflow, the call will *never* timeout. The default of **LAST[LONG CARDINAL]** falls in this category.

Note: Timing is begun after the user client returns from the **streamCheckoutProc**, or if it is **NIL**, immediately after completing **arguments** processing. It does not include the time to create the **transport**, to process **arguments**, to transfer **bulk data** (see §6.6.5) or to process **results**.

The server client may raise **Courier.SignalRemoteError** instead of returning results. This will translated to **Courier.RemoteErrorSignalled** at the user machine. The concatenation of the binding information and the **errorNumber** is equivalent to a unique Mesa signal and can be used to dispatch on proper code to process the signal's arguments, which must be done by calling **arguments** with an appropriate **Courier.Parameters** record. Like a Mesa signal, once **RemoteErrorSignalled** is raised, the client should no longer expect the **Call** to return results as well.

Caution: **Courier.RemoteErrorSignalled**'s arguments must be processed before **UNWINDING**. To **UNWIND** would cause Courier to lose all the state being maintained for the call.

Note: Like arguments or results, a signal with no arguments is a call to **arguments** with a parameter of **Courier.nullParameters**.

Notes: Courier user clients must distinguish the difference between **Courier.Error** and **Courier.RemoteErrorSignalled**. The former is a Courier failure, while the latter is a (more useful) conveyance of status from a remote service.

6.6.3.1.4 Freeing results

```
Courier.Free: PROCEDURE [parameters: Courier.Parameters, zone: UNCOUNTED ZONE];
```

Any time Courier translates Courier data types to Mesa data types, it may be necessary for Courier to allocate storage for disjoint data structures. The storage will be allocated by Courier on the client's behalf from the **zone** specified during the binding (**Courier.Create[..., zone: UNCOUNTED ZONE,...]**) using the standard **Heap** machinery. The client is responsible for deallocating these nodes, and to make that task easier, Courier provides the **Free** procedure. Once the client has processed the results, this procedure may be called, freeing all nodes allocated during the store (see §6.6.6) operation.

Note: It is never wrong to call **Courier.Free** after processing the results, even if no storage was allocated in the store process. It is considered an optimization requiring knowledge of the Courier and Mesa data structures involved to not do so.

6.6.3.2 Server's dispatcher

```
Courier.Dispatcher: TYPE = PROCEDURE [
    cH: Courier.Handle, procedureNumber: CARDINAL,
    arguments: Courier.Arguments, results: Courier.Results];

Courier.Results: TYPE = PROCEDURE [
    resultsRecord: Courier.Parameters ← Courier.nullParameters,
    requestDataStream: BOOLEAN ← FALSE]
    RETURNS [sH: Stream.Handle];
```

The dispatcher is the server client's link with the RPC mechanism. The dispatcher procedure is registered by the service implementor via **Courier.ExportRemoteProgram**. When a user places a call, Courier will search its internal lists of exports for an appropriate export and call the registered **dispatcher** using a process spawned by Courier. The client dispatcher is passed information similar to that which the user client passes to **Courier**: a **cH** (**Courier.Handle**), a **procedureNumber** indicating the exact procedure requested from the service, and two procedures (**arguments** and **results**) that the client uses to link the appropriate parameter areas and description routines to the procedure's parameters.

6.6.3.2.1 Completing the binding

```
Courier.NoSuchProcedureNumber: ERROR;
```

The dispatcher's first responsibility is to complete the binding. The only unbound element is the **procedureNumber**. The dispatcher must verify that the **CARDINAL** number supplied is valid for the service, and if not, raise the signal **Courier.NoSuchProcedureNumber**. Once the dispatcher verifies that the procedure does exist, it is obligated to service the remote calls in the manner prescribed by the protocol it implements.

6.6.3.2.2 Processing the remote procedure call

The server client code first processes the arguments of a procedure by calling the supplied **arguments** procedure with an appropriate **Courier.Parameters** record. If there are no

procedure arguments, **arguments** must still be called with parameters of **Courier.nullParameters**. If actual arguments do exist, the **location** field of the **Parameters** record is assumed to point to an uninitialized but writable section of virtual memory. The argument data will be translated from Courier data types to Mesa data types with help from the description routine.

After processing the procedure's arguments, the service is expected to perform some predefined function, a function not known to Courier, and one that may include bulk data transfer (see §6.6.5). The byte stream is available (**cH.sH**) to the server client after Courier returns from **arguments**. The client is assumed to be *finished* with the stream when it calls **results**.

When the service is complete, it must call **results**, either with a client defined parameter record or with **Courier.nullParameters**. The **results** returns a **Stream.Handle**. That handle will be **NIL** unless **requestDataStream** is assigned a value of **TRUE**. A non-**NIL** handle may be used for bulk data transfer. Client use of the stream in this case is assumed to be complete when it returns from the dispatcher.

6.6.3.2.3 Freeing the arguments

When storing (see §6.6.6) the **arguments** of a procedure call, Courier may allocate nodes of storage on behalf of its clients to store disjoint data structures. At sometime before returning from the dispatcher, the client must free that storage. This may be done as described in §6.6.3.1.4.

6.6.4 Errors

There is a considerable amount of error processing being performed by Courier. Most signals that might be raised by underlying implementations used by Courier are translated to **Courier.Error** with a (hopefully) meaningful **errorCode**. Other errors are implemented by Courier and may be raised by clients.

6.6.4.1 Errors raised by Courier

The following is a list of signals that Courier may raise and the client must catch. The discussions define the conditions under which they may be raised and suggest proper client reactions.

Courier.Error: ERROR [errorCode: Courier.ErrorCode];

Courier.ErrorCode: TYPE = {
 transmissionMediumHardwareProblem, transmissionMediumUnavailable,
 transmissionMediumNotReady, noAnswerOrBusy, noRouteToSystemElement,
 transportTimeout, remoteSystemElementNotResponding, noCourierAtRemoteSite,
 tooManyConnections, invalidMessage, noSuchProcedureNumber, returnTimedOut,
 callerAborted, unknownErrorInRemoteProcedure, streamNotYours,
 truncatedTransfer, parameterInconsistency, invalidArguments,
 noSuchProgramNumber, protocolMismatch, duplicateProgramExport,
 noSuchProgramExport, invalidHandle, noError};

This is the most common Courier signal. It should never be raised by and must always be caught by the client. Unless specifically noted, the following codes may be observed by both user and server clients.

transmissionMediumHardwareProblem This is most likely to happen during initial attempts at establishing a connection, but could happen at any time. It is also most likely to be related to circuit oriented devices. At any rate, it is highly unlikely that anything can be gained by retrying the operation. Call your support personnel for assistance.

transmissionMediumUnavailable

Always associated with circuit oriented devices, it indicates that the device is either currently or permanently unavailable. One should check the hardware to verify its configuration, and if properly configured, retry at a later time.

transmissionMediumNotReady

Always associated with circuit oriented devices, suggests that the medium is operational, but unable to accept data. Possible remedies are to manually dial the phone or ready the modems.

noAnswerOrBusy

This error applies to circuit oriented media only and indicates that the local hardware was operational, but the remote either did not answer or was already busy. Retry at a later time (on the order of minutes).

noRouteToSystemElement

The network on which the remote machine resides is not reachable at this time. The internet may have been temporarily partitioned (due to system failure) such that the network is no longer reachable. Retry the operation at a later time (on the order of minutes).

transportTimeout

An active connection has suddenly become unusable. It may be due to the remote machine becoming inoperable or to an error prone connection somewhere in the internet.

remoteSystemElementNotResponding

Trying to establish a connection failed after a reasonable amount of time and attempts. Either the remote machine is inoperable or it does not exist on the specified network. Check the network topology and the state of the machine in question. This error will be observed only by user clients.

noCourierAtRemoteSite

Observed only at the user machine, an attempt to establish a connection with a remote machine succeeded, but it was found that Courier was not

listening, an indication that no services are exported by that machine.

tooManyConnections

Courier has a limit as to how many transports it will support simultaneously. Creating the transport for this connection would exceed that limit. Try again at a later time (on the order of seconds). This error will only be observed on the user machine, but may reflect a condition on either the user or server machine.

invalidMessage

A message received from a remote machine was of the wrong format. This is an error in either Courier's or in the Courier client's protocol implementation. Retrying the operation will probably not be fruitful. This error will be observed only by user clients.

noSuchProcedureNumber

The remote service does not implement the procedure specified. This is a client protocol violation. Retrying will not help. This error will only be observed at the user.

returnTimedOut

A remote procedure call did not complete in the specified amount of time. Courier has abandoned the call. This could be due to an overloaded server, so retrying at a later time (minutes) may work. This error is observed only by user machines.

callerAborted

This error, observed only on server machines, indicates that the service has taken too long to formulate its reply. The calling machine has abandoned the call. The results cannot be delivered. The server *never* retries operations.

unknownErrorInRemoteProcedure

Observed only at the user, an undefined error has occurred. The server machine's integrity is in doubt and retrying could compound the problem.

streamNotYours

A client of *inter-call* (§6.6.5.2) style bulk data transfer has attempted to call **Courier.ReleaseDataStream** when it did not have the stream checked out. If the client had previously used the (a) stream, the integrity of the Courier RPC transport is in doubt. The problem should rectify itself, but several RPCs may fail first. This is a client implementation error.

truncatedTransfer

This error code will only be observed by implementors of the bulk data transfer protocol. Bulk data transfer protocol implementors are clients of the filtered byte stream provided by Courier for that purpose. That protocol requires that data be transmitted with a **SubSequenceType** other than 0. This error implies that the stream client attempted to consume some data of **SubSequenceType** of 0.

parameterInconsistency

Client parameter processing error. This is probably due to a malformed Mesa data item or an invalid implementation of the client's protocol (in the description routine). In such cases is doubtful that retrying the operation will help, and it might hurt. It is also possible (but highly unlikely) that the transport has failed to deliver the data correctly.

invalidArguments

Either Courier or the client description routine has noted a discrepancy in the format of the arguments and raised **Courier.InvalidArguments**. Courier caught the signal and either sent a reject (if it was raised remotely) or translated it into a **Courier.Error**.

noSuchProgramNumber

The program number that the user wishes to bind to is not exported at the server in any version. Retrying will not be helpful. Verify that the correct machine is being accessed for the service desired. This error will only be observed on user machines.

protocolMismatch

Observed only at the user during initial transport creations, indicates that the user and server are running incompatible versions of the Courier protocol. No retrying is in order. Check the network topology and the versions of software running at the respective machines.

duplicateProgramExport

This error code is observed only when attempting to export a service. It indicates that the **programNumber** and **versionRange** parameters of **Courier.ExportRemoteProgram** matched *exactly* with those already known by Courier.

noSuchProgramExport

This error code is observed only when attempting to unexport a remote service. It indicates that the **programNumber** and **versionRange** specified in the unexport request

(**Courier.UnexportRemoteProgram**) did not have an equivalent known to Courier.

invalidHandle

An operation requiring a **Courier.Handle** checked the handle and found it to be invalid. The handle was probably already deleted, or (even worse) never created. Don't retry the operation.

noError

This should never be observed by any Courier client. It is included to simplify internal processing.

Courier.VersionMismatch: ERROR [versionRange: Courier.VersionRange];

The remote service version number is passed as part of every remote procedure call. If the Courier server discovers that the machine does export the program, but not the particular version, it will notify the user machine of the range of versions supported by the server. The user then has the option to observe that range, and if it implements a compatible version, to retry the operation with appropriate parameters.

Note: This feature is only implemented for servers of Courier version 3 or higher.

Courier.RemoteErrorSignalled: ERROR [
errorNumber: CARDINAL, arguments: Courier.Parameters ← Courier.nullParameters];

RemoteErrorSignalled is Courier's equivalent to a Mesa signal. The signal is initiated in the signaller (server) machine by the client raising the signal **Courier.SignalRemoteError** (see §6.6.4.2), thus aborting the call. At the user, the abort message is used to reconstruct the context of the signal, renaming it **Courier.RemoteErrorSignalled**. The argument **errorNumber** of the signal permits the client to dispatch to the appropriate processing code. The remaining context of the signal must be retrieved by calling **arguments**. If the semantics of the signal indicate no arguments exist, then **arguments** should be called with a defaulted value of **Courier.nullParameters**. The arguments of the signal must be processed before the **UNWIND** is generated.

6.6.4.2 Signals clients may raise

Courier.NoSuchProcedureNumber: ERROR;

During the client dispatcher's final phase of binding, it may find that the **procedureNumber**, specified as one of the **Courier.Dispatcher** arguments, is invalid. It must then raise this signal, and Courier will transfer that information to the caller and reject the call. This signal must not be raised by the client except in the dispatcher. At the user the information will be translated to **Courier.Error[noSuchProcedureNumber]**.

Courier.InvalidArguments: ERROR;

Client description routines may notice unacceptable parameters. If this is so, the client may raise **InvalidArguments**. This signal will be translated by Courier to **Courier.Error[invalidArguments]** at the user. Both server and user code may raise this

signal; the server will not translate the error locally, but it will reject the call, send the information to the user, where **Error[invalidArguments]** will be raised.

```
Courier.SignalRemoteError: ERROR [
    errorNumber: CARDINAL, arguments: Courier.Parameters ← Courier.nullParameters];
```

SignalRemoteError is the mechanism Courier client servers use to emulate the generation of a Mesa signal. Courier intercepts the signal and translates it into an *abort* message that includes the **errorNumber** and any additional **arguments** the client may have specified. If the semantics of the signal are that no arguments exist, **arguments** should be assigned (or defaulted) a value of **Courier.nullParameters**.

Note: Courier will call the client's argument description routine before **UNWINDING** from the catch phrase.

6.6.5 Bulk data

Courier supports applications whose communication requirements are primarily transactional in nature. However, not all network communication is transaction oriented. File transfer, for example, is more appropriately modelled as bulk data transfer. In order to blend this bulk transfer requirement with the transactional nature of remote procedure calling, Courier provides access to an established *byte stream*, permitting the client to use that byte stream for those applications that require it.

6.6.5.1 Intra-call bulk transfer

```
Courier.Call: PROCEDURE[...]
    streamCheckoutProc: PROCEDURE [cH: Courier.Handle], ...]...;
```

```
Courier.Object: TYPE = RECORD [..., sH: Stream.Handle, ...];
```

The Courier user and server client have the stream made available via the **Courier.Object** that is in turn accessible through the **Courier.Handle**. The stream contained therein is slightly limited when compared to a generic Pilot stream. It may be used *only* between **argument** and **result** processing and it will not permit the client to set the *Subsequence Type* to a value of zero, nor will it permit the client to delete the stream. Attempts to do these will result in the error **Stream.InvalidOperation**.

Note: The client is responsible for processing all signals that might be raised by a Pilot stream.

The user client is given control after the processing of the **arguments** if the **streamCheckoutProc** has a value other than **NIL**. At the server, the client has control between the processing of the **arguments** and **results** and may use the stream at that time. The state of the stream provided the client is a *default* stream (i.e., **timeout = 60 seconds**, **sst = 0**, **input options = Stream.defaultInputOptions**) It is assumed the client is finished with the bulk transfer when it returns from the **streamCheckoutProc** procedure (user) or calls **results** (server). The state of the returned stream is undefined and Courier expects to have to reset the parameters for its subsequent use.

6.6.5.2 Inter-call bulk transfer

```
Courier.Call: PROCEDURE [..., requestDataStream: BOOLEAN, ...]  
    RETURNS[sH: Stream.Handle];  
  
Courier.Results: TYPE = PROCEDURE [..., requestDataStream: BOOLEAN, ...]  
    RETURNS[sH: Stream.Handle];  
  
Courier.ReleaseDataStream: PROCEDURE [cH: Courier.Handle];  
  
Courier.ErrorCode: TYPE = {..., streamNotYours, ...};
```

This version of bulk transfer provides the client with an *unfiltered* stream, unrestricted by Courier in any way, either as a result of the `Courier.Call` at the user or as a result of calling `results` at the server. If the parameter `requestDataStream` is `FALSE`, the value returned for `sH` will be `NIL`. If the parameter `requestDataStream` is `TRUE`, the stream provided is a default stream as described in §6.6.5.1. The user client is assumed finished with the stream when he calls `Courier.ReleaseDataStream`. Attempting to release a stream that was never checked out will result in the error `Courier.Error[streamNotYours]` being raised. Until that time the transport cannot be used for any other purpose, including another remote procedure call. At the server Courier assumes ownership of the stream when the client returns from his dispatcher. The client may perform any stream operation desired *except delete* and those not supported by the transport (such as positioning in the case of Network streams).

6.6.6 Description routines

Courier description routines are used to translate Mesa data types to and from Courier data types. Courier provides the machinery to perform this translation process via a *notes object* passed by reference to each description routine.

The notes object contains the context within which the description routine is operating.

```
Courier.Description: TYPE = PROCEDURE [notes: Courier.Notes];  
  
Courier.Notes: TYPE = POINTER TO Courier.NotesObject;  
  
Courier.NotesObject: TYPE = RECORD [...];
```

Courier requires client assistance to map Mesa data types into Courier data types. The client provides that assistance in the form of a *description routine*. Description routine procedures are of type `Courier.Description`. The notes object is passed by reference to all client description routines. It contains context about the process being performed and a series of procedures to perform the bulk of the work involved in mapping Mesa data types to and from Courier data types.

6.6.6.1 Mesa data type restrictions

The Courier Protocol supports a set of data types that closely corresponds to the set of common Mesa data types. Because the Courier Protocol is intended for a heterogeneous

internet, however, not all Mesa types are supported. Also, for those Mesa data types that are supported, there are a few restrictions that arise from the need to maintain a set of data types that are reasonably easy to support on other types of systems.

Below are suggested mappings of Courier data types to compatible Mesa data types. Since Courier has a Mesa heritage, finding a semantically equivalent Mesa data type for every Courier data type is a fairly simple task.

6.6.6.1.1 Fully compatible data types

The following data types have equivalent representations in Courier and Mesa.

<u>Courier data type</u>	<u>Corresponding Mesa data type</u>
CARDINAL	CARDINAL
INTEGER	INTEGER
UNSPECIFIED	UNSPECIFIED

6.6.6.1.2 Data type compatibility supported by Courier clients

The following Courier data types have a representation in Mesa, but is not a common data type. Courier does not support the noting of these data types within the description routine. It is the responsibility of the Courier client to use the restricted form shown below.

<u>Courier data type</u>	<u>Corresponding Mesa data type</u>
BOOLEAN	MACHINE DEPENDENT RECORD [zeros: [0..7777B], value: BOOLEAN]
{ $id_1(v_1), \dots id_n(v_n)$ }	MACHINE DEPENDENT { $id_1(v_1), \dots id_n(v_n)$, LAST[CARDINAL]}
RECORD[$id_1: Type_1, \dots id_n: Type_n$]	MACHINE DEPENDENT RECORD [$id_1: Type_1, \dots id_n: Type_n$]

6.6.6.1.3 Data type compatibility supported by Courier via notes

The following Courier data types have a representation similar to that of Mesa. The differences are resolved at the time the description routine notes instances of them.

<u>Courier data type</u>	<u>Corresponding Mesa data type</u>
LONG CARDINAL	LONG CARDINAL
LONG INTEGER	LONG INTEGER
STRING	LONG STRING
ARRAY n OF Type	ARRAY [0..n) OF Type
CHOICE n OF {list}	MACHINE DEPENDENT RECORD [$id_0: Type_0,$ $id_1: Type_1,$... $id_n: SELECT n FROM$ $tag_0 = > [Type_{n}],$ $tag_1 = > [Type_{n+1}],$... $tag_m = > [Type_{n+m}]$] DESCRIPTOR FOR ARRAY OF Type
SEQUENCE n OF Type	

6.6.6.2 Description context

The notes object contains the context within which the description routine is operating.

Courier.NotesObject: TYPE = RECORD [type: {fetch, store, free}, ...];

The first field of the notes object informs the client what type of operation is to be performed. The notes object procedures are designed such that most of the operations are performed as *side-effects* enabling a single description procedure to perform all three of the following operations without caring about the specific operation **type**. In some cases, however, the client needs to be aware of the current operation.

fetch

To **fetch** is to translate Mesa data types to Courier data types. This is sometimes referred to as *serialization* or *marshalling* of data. This action occurs when call parameters are processed by the user or when return parameters are processed by the server.

store

To **store** is to translate Courier data types to Mesa data types. This is sometimes referred to as *deserialization* or *unmarshalling*. This action occurs when call parameters are being processed by the server or when result parameters are being processed by the user. In such cases, Courier will allocate nodes of storage for disjoint data structures (**LONG STRING**, **LONG DESCRIPTOR FOR ARRAY**, **DisjointData**) from the current **zone**. In some cases, the client may wish (or have) to allocate nodes directly, as in the case of **NoteSpace**.

free

The Courier client is required to free the storage nodes allocated by Courier during a store operation. It is possible and recommended that the client do that via the **Courier.Free** operation. When a description routine is being called with **type** of **free**, the client has the opportunity to release nodes that he may have allocated unknown to Courier, such as nodes for the **NoteSpace** operation.

Courier.NotesObject: TYPE = RECORD [..., zone: UNCOUNTED ZONE, ...];

The description client is also made aware of the *heap* that the program wishes to use to allocate or free storage. This field is a copy of the zone registered by the client during **Courier.Create** and **Courier.ExportRemoteProgram**. The client will find that the **zone** field is most useful during storing and freeing operations.

6.6.6.3 Data noting procedures

Each note routine contained in the notes object is provided to perform mapping to and from explicit Courier and Mesa data types. Each routine has at least three properties. First, it has a specific Mesa to Courier mapping function. Second, it contains the *site* of the data being described. Third, the note procedure consumes an implicit amount of the parameter area.

6.6.6.3.1 NoteSize

Courier.NotesObject: TYPE = RECORD [..., noteSize: Courier.NoteSize, ...];

Courier.NoteSize: TYPE = PROCEDURE [size: CARDINAL] RETURNS [site: LONG POINTER];

The first responsibility of a description routine is to note the *size* of the record being described. This size (in words) coupled with the starting address of the record defines a parameter area whose contents must be *noted*, either explicitly through one of the data noting procedures supplied in the *notes object*, or implicitly by skipping over a portion of the parameter area with other explicit notes, or by returning from the description routine. No data noting procedures may be called before **NoteSize** and **NoteSize** may not be called more than once per description routine.

6.6.6.3.2 NoteLongCardinal, NoteLongInteger

Courier.NotesObject: TYPE = RECORD [...,
noteLongCardinal: Courier.noteLongCardinal,
noteLongInteger: Courier.noteLongInteger, ...];

Courier.NoteLongCardinal: TYPE = PROCEDURE [
site: LONG POINTER TO LONG CARDINAL];

Courier.NoteLongInteger: TYPE = PROCEDURE [
site: LONG POINTER TO LONG INTEGER];

All **LONG CARDINAL** and **LONG INTEGER** data types contained in the parameter area must be explicitly noted. Two words are consumed from the parameter area with each call.

6.6.6.3.3 NoteString

Courier.NotesObject: TYPE = RECORD [..., noteString: Courier.NoteString, ...];

Courier.NoteString: TYPE = PROCEDURE [site: LONG POINTERTO LONG STRING];

All **LONG STRING** data types contained in the parameter area must be explicitly noted. Two words are consumed from the parameter area with each call. Storage for the **StringBody** will be allocated from the notes object **zone** by the store operation.

Note: The maxlen attribute of the Mesa **StringBody** will be lost in the fetching operation. Consequently, stored strings will always have a maxlen equal to the length.

Caution: Strings that are **NIL** or have a length of zero when fetched are always stored as strings with zero length. The client must be aware that such stored strings are **READONLY**. They must not be modified in any way. They must not be freed except by the **Courier.Free** operation.

6.6.6.3.4 NoteChoice

```
Courier.NotesObject: TYPE = RECORD [..., noteChoice: Courier.NoteChoice, ...];
```

```
Courier.NoteChoice: TYPE = PROCEDURE [
    site: LONG POINTER,
    size: CARDINAL,
    variant: LONG DESCRIPTOR FOR ARRAY OF CARDINAL,
    tag: LONG POINTER ← NIL];
```

NoteChoice provides the Courier client with a somewhat restricted use of the Mesa variant record. In addition to the **site** parameter, the procedure call also specifies the undiscriminated length of the variant record. It is that length that will be consumed from the parameter area by the procedure call. The client is also required to supply an array descriptor for an array of variant record discriminated lengths. A fourth optional parameter specifies the address of the variant record's **tag** field. If that field is omitted, assigned a value of **NIL**, or a value equal to that of the **site** parameter, Courier assumes that the variant tag is the first element of the variant record. Otherwise it assumes a record with a static portion followed by a variant portion.

Note: The variant **tag** must be word aligned and 16-bits wide.

6.6.6.3.5 NoteArrayDescriptor

```
Courier.NotesObject: TYPE = RECORD [...,
    noteArrayDescriptor: Courier.NoteArrayDescriptor, ...];
```

```
Courier.NoteArrayDescriptor: TYPE = PROCEDURE [
    site: LONG POINTER, elementSize, upperBound: CARDINAL];
```

This procedure notifies Courier that a Mesa **LONG DESCRIPTOR** exists at **site**. The procedure call consumes three words of the parameter area. But since descriptors define disjoint data in the form of an array, the virtual memory defined by that array is not from the original (or current) parameter area. For that reason, another parameter area is fabricated using the descriptor's **BASE** and **LENGTH**, the latter being multiplied by the length of each element as passed by the client. The newly defined parameter area must be completely consumed before any more of the previous parameter area can be processed. For store operations, the storage for the parameter area will be allocated from the notes object **zone**. The last parameter, **upperBound**, is the maximum **LENGTH** that Courier should accept within the descriptor.

Note: Descriptors having **BASE** = **NIL** or **LENGTH** = 0 during the fetch will always be stored as **DESCRIPTOR[NIL, 0]**.

6.6.6.3.6 NoteDisjointData

```
Courier.NotesObject: TYPE = RECORD [...,
    noteDisjointData: Courier.NoteDisjointData, ...];
```

```
Courier.NoteDisjointData: TYPE = PROCEDURE [
    site: LONG POINTER TO LONG POINTER, description: Courier.Description];
```

NoteDisjointData permits the client to note data that is only referenced via a **LONG POINTER** in the parameter area. It is provided as a convenience to clients to eliminate local copying of parameters or as *data hiding* mechanism. **NoteDisjointData** consumes two words from the parameter area. The second argument of the procedure is another description routine. Courier will call that routine, and it will in turn call **noteSize**. The beginning of the disjoint area and the size define a new parameter area. That parameter area will be allocated from the notes object **zone** during store operations. Pointers are not Courier data types. The pointer will be dereferenced and the dereferenced object processed during a fetch operation. An appropriate object will be allocated from the notes object **zone** and a pointer to that object will be placed in the client parameter area during store operations. No notion of a pointer (or its absence) will be conveyed to the storing machine by Courier.

Caution: This scheme does not lend itself to processing of linked list and other recursive data structures that are associated via pointers. Linked lists may be processed if properly approached. Some other bit of information must be transmitted, usually a **BOOLEAN**, that indicates the last element of a list has been processed so the recursion can be broken by the storing client.

6.6.6.3.7 NoteParameters

```
Courier.NotesObject: TYPE = RECORD [...,
    noteParameters: Courier.NoteParameters, ...];
```

```
Courier.NoteParameters: TYPE = PROCEDURE [
    site: LONG POINTER, description: Courier.Description];
```

NoteParameters is much like **NoteDisjointData** except there is no pointer involved. The second argument of the procedure call is again a description routine. The closely following call to **noteSize** coupled with the site of the **noteParameter** defines a new parameter area. That new parameter area must be totally contained within the previous parameter area. In the former case the amount of space specified in the **noteSize** operation will be consumed from the current parameter area.

6.6.6.3.8 NoteSpace

```
Courier.NotesObject: TYPE = RECORD [...,noteSpace: Courier.NoteSpace, ...];
```

```
Courier.NoteSpace: TYPE = PROCEDURE [site: LONG POINTER, size: CARDINAL];
```

NoteSpace permits a Courier client to process a block of unspecified data. It does not define a new parameter area, hence no data can be noted within the space defined by **NoteSpace**. The data is not linked to the parameter area in any way. Consequently, the store space must be allocated by the client, unlike other disjoint data types. The procedure call consumes no portion of the parameter area.

Caution: **NoteSpace** does not cause unnoted data to be processed. The space being described is completely divorced from the current client parameter area.

6.6.6.3.9 NoteDeadSpace

```
Courier.NotesObject: TYPE = RECORD [...,  
    noteDeadSpace: Courier.NoteDeadSpace, ...];
```

```
Courier.NoteDeadSpace: TYPE = PROCEDURE [site: LONG POINTER, size: CARDINAL];
```

NoteDeadSpace is used to consume a portion of the parameter area without generating any Courier data, just the opposite of **NoteSpace**. The amount of parameter area to be consumed is client specified.

Note: **NoteDeadSpace** does cause unnoted data to be processed. Consequently, it is the procedure of choice used to force unnoted data to be processed (e.g., **notes.noteDeadSpace[site, 0]** will cause all *unnoted* data in the current parameter area to be processed and then consume zero more words of that parameter area).

6.6.6.3.10 NoteBlock

```
Courier.NotesObject: TYPE = RECORD [...,  
    noteBlock: Courier.NoteBlock, ...];  
Courier.NoteBlock: TYPE = PROCEDURE [block: Environment.Block];
```

NoteBlock provides Courier clients with a mechanism that enables them to process byte oriented data. This procedure will process only the bytes defined by the **Environment.Block**, and will consume nothing from the current parameter area. Nor will Courier allocate storage during store operations for the disjoint area implied by the operation.

Caution: It is expected that this procedure will be used by clients as a building block for complicated description routines. When using **NoteBlock** such clients are responsible for insuring that an even number of bytes actually gets processed with each *complete* operation, even if it means appending a *null* byte to the end of a stream of bytes. Courier data types *always* begin on 16-bit (word) boundaries.

6.6.6.3.11 Unnoted

Unnoted data is a concept rather than a procedure. Parameter areas are represented internally and conceptually as **ORDERED LONG POINTERS**, constructed initially by the *location* parameter of a **Courier.Parameters** and the *size* parameter of a **notes.noteSize** procedure call. Subsequent parameter areas may be created when describing disjoint data structures (e.g., **DisjointData**, **DescriptorForArray**). All the data noting procedures specify a site that is an address within the bounds of a parameter area. The current data point within the record is known to be the last site specified plus the amount of data consumed by the last note procedure. The portion of the parameter area between that *left edge* and the current site is *unnoted* data and is processed as such, implying that the Courier and Mesa data types are compatible.

6.6.7 Miscellaneous facilities

```
Courier.SerializeParameters: PROCEDURE [  
    parameters: Courier.Parameters, sh: Stream.Handle];
```

Courier.DeserializeParameters: PROCEDURE[
 parameters: Courier.Parameters, sh: Stream.Handle, zone: UNCOUNTED ZONE];

These two procedures provide access to the *description routine* facilities of Courier outside the bounds of a remote procedure call. **SerializeParameters** performs a fetch operation, converting Mesa data types defined by the **parameters** record to Courier data types and *putting* them on the stream defined by **sh**. The client is responsible for all signals that may be raised by the stream implementation. **DeserializeParameters** is the counterpart of **SerializeParameters**. It performs a store operation, converting Courier data types *gotten* from the stream **sh** to Mesa data types defined by the **parameters** record. Since this is a store operation, Courier may have to allocate storage for disjoint data structures. If so, the storage will be allocated from **zone**. As with any store operation, the client assumes responsibility for that storage and may deallocate it via **Courier.Free**.

Courier.LocalSystemElement: PROCEDURE RETURNS[Courier.SystemElement];

This procedure returns a full network address of the local machine. The *socket* field of the address will always be Courier's well-known socket.

Courier.EnumerateExports: PROCEDURE RETURNS[
 enum: LONG DESCRIPTOR FOR Courier.Exports];

Courier.FreeEnumeration: PROCEDURE[
 enum: LONG DESCRIPTOR FOR Courier.Exports];

Courier.Exports: TYPE = ARRAY CARDINAL OF Courier.ExportItem;

Courier.ExportItem: TYPE = MACHINE DEPENDENT RECORD[
 programNumber: LONG CARDINAL,
 versionRange: Courier.VersionRange,
 serviceName: LONG STRING,
 exportTime: System.GreenwichMeanTime];

EnumerateExports will make a *copy* of the current internal structures representing the results of all previous **Courier.ExportRemoteProgram** requests. With one exception the elements of the array returned were supplied by the **ExportRemoteProgram** client. The exception, **exportTime**, is the time that the **ExportRemoteProgram** request was made. The storage for the enumeration array is allocated from a zone internal to Courier, so the client is obligated to free that space at some time, which he may do with **Courier.FreeEnumeration**.



Editing and Formatting

This chapter contains those facilities, usually Common Software packages, that are concerned primarily with formatting and editing. §7.1 describes an interface that defines some common ASCII characters; §7.2 describes a package for converting between some common Mesa types and strings; §7.3 discusses the standard string processing procedures; and §7.4 describes operations for converting between strings and Pilot's internal form of time.

7.1 ASCII character definitions

Ascii: DEFINITIONS ... ;

The Ascii package consists only of a definitions file.

All of the control characters of the form *control uppercase-letter* are defined in the form:

Ascii.Control*uppercase-letter*: CHARACTER = '*uppercase-letter* - 100B;

For example,

Ascii.ControlB: CHARACTER = 'B - 100B;

In addition, a few special control keys are defined as their commonly used names:

Ascii.BEL: CHARACTER = 'G - 100B;

Ascii.BS: CHARACTER = 'H - 100B;

Ascii.CR: CHARACTER = 'M - 100B;

Ascii.DEL: CHARACTER = 177C;

Ascii.ESC: CHARACTER = 33C;

Ascii.FF: CHARACTER = 'L - 100B;

Ascii.LF: CHARACTER = 'J - 100B;

Ascii.NUL: CHARACTER = 0C;

Ascii.SP: CHARACTER = ' ';

Ascii.TAB: CHARACTER = 'I - 100B';

7.2 Formatting

Format: DEFINITIONS . . . ;

The Format package provides procedures to format various types into strings. The procedures require the client to supply a string output procedure and a piece of data to be formatted. Where appropriate, a format specification is also required. The client may also specify client instance data to be used by the string output procedure. The Format package is a Product Common Software package. The implementation module is **FormatImpl.bcd**.

7.2.1 Binding

The Format package must be bound with the String and Time packages.

7.2.2 Specifying the destination of the output

The editing procedures defined in Format allow a client to pass in a procedure that will be called once editing of the particular item has been completed. This procedure will be called with an output string and with the **clientData** passed to the editing procedure. This procedure must be declared to be of type

Format.StringProc:PROCEDURE [s: LONG STRING, clientData: LONG POINTER ← NIL];

Every editing procedure in Format requires a parameter of this type and **clientData** to be passed to the editing procedure. If **NIL** is supplied for this procedure, the output is directed to the default output, known as a sink. The default output sink can be changed with the procedure

```
Format.SetDefaultOutputSink: TYPE =
PROCEDURE [new:Format.StringProc, clientData: LONG POINTER ← NIL]
RETURNS [old:Format.StringProc, oldClientData: LONG POINTER];
```

7.2.3 String editing

**Format.Char: PROCEDURE [proc: Format.StringProc, char: CHARACTER,
clientData: LONG POINTER ← NIL];**

Char calls on **proc** with a string of length 1 containing **c**.

**Format.LongSubStringItem: PROCEDURE [proc: Format.StringProc, ss: String.LongSubString,
clientData: LONG POINTER ← NIL];**

Format.LongString, Text: PROCEDURE [proc: Format.StringProc, s: LONG STRING,
clientData: LONG POINTER ← NIL];

Format.SubString: PROCEDURE [proc: Format.StringProc, ss: String.SubString,
clientData: LONG POINTER ← NIL];

LongSubStringItem repeatedly calls **proc** with strings filled from **ss**.

LongString (or **Text**) calls **proc** with string **s**.

SubString calls **Format.LongSubStringItem** with **proc** and a pointer to a **String.SubStringDescriptor** whose **base** is **ss.base**, **offset** is **ss.offset** and **length** is **ss.length**.

Format.Blank, Blanks: PROCEDURE [proc: Format.StringProc, n: CARDINAL ← 1,
clientData: LONG POINTER ← NIL];

Format.Block: PROCEDURE [proc: Format.StringProc, block: Environment.Block,
clientData: LONG POINTER ← NIL];

Format.CR: PROCEDURE [proc: Format.StringProc, clientData: LONG POINTER ← NIL];

Format.Line: PROCEDURE [proc: Format.StringProc, s: LONG STRING,
clientData: LONG POINTER ← NIL];

The procedure **Blank(s)** calls **proc** with a string containing **n** spaces. **Block** calls **proc** with the contents of **block**. **CR** calls **proc** with a string containing a carriage return. The procedure **Line** calls **proc** with **s**, then with a string containing a carriage return.

7.2.4 Editing numbers

The format into which numbers are to be edited is governed by a record of the form

Format.NumberFormat: TYPE = RECORD [base: [2..36] ← 10,
zerofill: BOOLEAN ← FALSE, unsigned: BOOLEAN ← TRUE, columns: [0..255] ← 0];

Format.OctalFormat: Format.NumberFormat = [base: 8, zerofill: FALSE,
unsigned: TRUE, columns: 0];

Format.DecimalFormat: Format.NumberFormat =
[base: 10, zerofill: FALSE, unsigned: FALSE, columns: 0];

The number editing procedure described below will edit the number parameter as follows: the number will be edited in base **base** in a field **columns** wide (zero means use as many as needed). If **zerofill** is **TRUE**, the extra columns are filled with zeros, otherwise spaces are used. If **unsigned** is **TRUE**, the number is treated as a cardinal.

Two **NumberFormat** records are defined for convenience. **OctalFormat** specifies editing the number as a cardinal in base eight number, using as many columns as needed, no zero fill. **DecimalFormat** specifies editing the number as an integer in base ten number, using as many columns as needed, no zero fill.

Format.Number: PROCEDURE [proc: Format.StringProc, n: UNSPECIFIED,
format: Format.NumberFormat, clientData: LONG POINTER ← NIL];

Format.LongNumber: PROCEDURE [proc: Format.StringProc, n: LONG UNSPECIFIED,
format: Format.NumberFormat, clientData: LONG POINTER ← NIL];

Number and **LongNumber** convert **n** to a string of the base specified in **format**. If **format.unsigned** is FALSE and **n** is negative, the character "-" is output. If the numeric string length is less than **format.columns** then **proc** is called, perhaps multiple times, to output the necessary number of leading zeros (if **format.zerofill**) or spaces, before being called to output the numeric string. If the numeric string length is greater than **format.columns**, then **proc** is called.

Format.Decimal: PROCEDURE [proc: Format.StringProc, n: INTEGER,
clientData: LONG POINTER ← NIL];

Format.LongDecimal: PROCEDURE [proc: Format.StringProc, n: LONG INTEGER,
clientData: LONG POINTER ← NIL];

Decimal and **LongDecimal** convert **n** to signed base ten. **proc** is then called.

Format.Octal: PROCEDURE [proc: Format.StringProc, n: UNSPECIFIED,
clientData: LONG POINTER ← NIL];

Format.LongOctal: PROCEDURE [proc: Format.StringProc, n: LONG UNSPECIFIED,
clientData: LONG POINTER ← NIL];

Octal and **LongOctal** convert **n** to base eight. When **n** is greater than 7, the character B is appended. **proc** is then called.

7.2.5 Editing dates

DateFormat allows the user to specify the format in which the date is to be edited by the procedure **Format.Date**.

Format.DateFormat: TYPE = {dateOnly, noSeconds, dateTime, full, mailDate};

The different formats have the following interpretation:

maildate:	27 Jul 83 09:23:29 PDT (Wednesday)
full:	27-Jul-83 9:23:29 PDT
dateTime:	27-Jul-83 9:23:29
noSeconds:	27-Jul-83 9:23
dateOnly:	27-Jul-83

The **maildate** format is the ANSI standard format for dates. Note the leading zero on the time (when appropriate) and the omitted hyphens from the date. Also note that fewer time zones have standard abbreviations (Pacific through Eastern and Greenwich).

Format.Date: PROCEDURE [proc: Format.StringProc, pt: Time.Packed,
format: Format.DateFormat ← noSeconds, zone: Time.TimeZone ← ANSI, clientData:
LONG POINTER ← NIL];

Date converts *pt* to a string of the form "27-Jul-83 9:23:29 PDT" which is truncated based on the specified **format**. **proc** is then called. The zone parameter indicates in which format numeric time zones are represented (see §7.4.2 for a description of the representations).

7.2.6 Editing network addresses

The following procedures can be used to edit network addresses into various forms. The exact form of the editing is specified with the type

Format.NetFormat: TYPE = {octal, hex, productSoftware};

octal converts the number to octal, **hex**, to hex, and **productSoftware** converts the item to a decimal number and then inserts a "-" every three characters, starting from the right. An example of number in product software format is 4-294-967-295.

Format.HostNumber: PROCEDURE [proc: Format.StringProc,
hostNumber: System.HostNumber, format: Format.NetFormat,
clientData: LONG POINTER ← NIL];

Format.NetworkAddress: PROCEDURE [proc: Format.StringProc,
networkAddress: System.NetworkAddress, format: Format.NetFormat,
clientData: LONG POINTER ← NIL];

Format.NetworkNumber: PROCEDURE [proc: Format.StringProc,
networkNumber: System.NetworkNumber, format: Format.NetFormat,
clientData: LONG POINTER ← NIL];

Format.SocketNumber: PROCEDURE [proc: Format.StringProc,
socketNumber: System.SocketNumber, format: Format.NetFormat,
clientData: LONG POINTER ← NIL];

A network address will be edited into the form *network-number # host-number # socket-number* where the editing of the various components will be determined by **format**.

7.3 Strings

String: DEFINITIONS . . . ;

The **String** interface provides facilities for string manipulation. It is Product Common Software. The implementation modules for **String** are **StringImplA.bcd** and **StringImplB.bcd**.

Note: The following procedures have been retained in the **String** interface for compatibility. Their use is strongly discouraged. Please see **String.mesa** for details of their definition: **StringLength**, **EmptyString**, **EqualString**, **EqualString**, **EquivalentString**, **EquivalentStrings**, **CompareStrings**, **EqualSubStrings**, **EquivalentSubStrings**.

7.3.1 Sub-strings

A **SubStringDescriptor** describes a region within a string. The first character is **base[offset]** and the last character is **base[offset + length - 1]**.

```
String.SubStringDescriptor: TYPE = RECORD [base: LONG STRING,  
offset, length: CARDINAL];
```

```
String.SubString. LONG POINTER TO SubStringDescriptor;
```

7.3.2 Overflowing string bounds

```
String.StringBoundsFault: SIGNAL [s: LONG STRING] RETURNS [ns: LONG STRING];
```

StringBoundsFault signal is raised when any of the append procedures described below would have to increase the length of their argument string's **length** to be larger than its **maxlength**. The catch phrase may allocate a longer string **ns** and return it to **StringBoundsFault**. The operation will then be restarted as if **ns** had been the original argument. If **StringBoundsFault** is resumed with the value **NIL**, the procedure that raised the signal will fill in the original string with as many characters as will fit.

7.3.3 String operations

The procedure

```
String.WordsForString: PROCEDURE [nchars: CARDINAL] RETURNS [CARDINAL];
```

calculates the number of words of storage needed to hold a string of length **nchars**. The value returned includes any system overhead for string storage.

There are two case changing procedures:

```
String.LowerCase, UpperCase: PROCEDURE [c: CHARACTER] RETURNS [CHARACTER];
```

These procedures change the parameter character to lower or upper, respectively. The procedures are no-ops if the character is not a letter.

```
String.AppendChar: PROCEDURE [s: LONG STRING, c: CHARACTER];
```

AppendChar appends the character **c** to the end of the string **s**. **s.length** is updated; **smaxlength** is unchanged.

```
String.AppendString: PROCEDURE [to, from: LONG STRING];
```

AppendString appends the string **from** to the end of the string **to**. **to.length** is updated; **to maxlength** is unchanged.

```
String.AppendSubString: PROCEDURE [to: LONG STRING, from: String.SubString];
```

AppendSubString appends the substring in **from** to the end of the string in **to**. **to.length** is updated; **to maxlength** is unchanged.

String.Copy: PROCEDURE [to, from: LONG STRING,];

The procedure **Copy** sets the length of **to** to zero and then appends **from** to **to**.

String.DeleteSubString: PROCEDURE [s: String.SubString];

DeleteSubString deletes the substring described by **s** from the string **s.base**. **s.base.length** is updated; **s.base maxlen** is unchanged.

String.Empty: PROCEDURE [s: LONG STRING,] RETURNS [BOOLEAN];

The procedure **Empty** returns **TRUE** if **s** is **NIL** or if **s.length** is 0 and **FALSE** otherwise.

String.Equal: PROCEDURE [s1, s2: LONG STRING] RETURNS [BOOLEAN];

Equal returns **TRUE** if **s1** and **s2** contain exactly the same characters.

String.Equivalent: PROCEDURE [s1, s2: LONG STRING] RETURNS [BOOLEAN];

Equivalent returns **TRUE** if **s1** and **s2** contain the same characters except for case shifts. Strings containing control characters may not be compared correctly.

String.EqualSubString: PROCEDURE [s1, s2: String.SubString]
RETURNS [BOOLEAN];

EqualSubString is analogous to **Equal**.

String.EquivalentSubString: PROCEDURE [s1, s2: String.SubString] RETURNS [BOOLEAN];

EquivalentSubString is analogous to **Equivalent**.

String.Compare: PROCEDURE [s1, s2: LONG STRING, ignoreCase: BOOLEAN ← TRUE]
RETURNS [INTEGER];

Compare lexically compares two strings and returns -1, 0, or 1 if the first is less than, equal to, or greater than the second. An optional parameter may be supplied to have case differences ignored.

String.Length: PROCEDURE [s: LONG STRING,] RETURNS [CARDINAL];

The procedure **Length** returns zero if **s** is **NIL** and **s.length** otherwise.

String.StringToNumber: PROCEDURE [s: LONG STRING, radix: CARDINAL ← 10]
RETURNS [UNSPECIFIED];

String.InvalidNumber: SIGNAL;

StringToNumber interprets the characters of **s** as an integer or cardinal and returns its value. The form of a number is:

{spaces | controlCharacters} {-} {baseNumber} {'B' | 'b' | 'D' | 'd'} {scaleFactor}

where {} indicates an optional part and "!" indicates a choice, and *baseNumber* and *scaleFactor* are sequences of digits. The value returned is \pm *baseNumber*. * *radix*^{**}*scaleFactor*. *controlCharacters* are characters whose Ascii code is less than 40B. The *radix* used depends on the contents of *s* and *radix*: if the string has a 'B or 'b, *radix* will be 8; if the string has a 'D or 'd, *radix* will be 10; otherwise, *radix* will be *radix*. The number *scaleFactor* is always expressed in radix 10. If *s* does not have a valid form or *s.length = 0*, **String.InvalidNumber** is raised. Values of *radix* other than 8 or 10, the use of the digits 8 and 9 when radix 8 is in effect, and the specification of a number whose value falls outside of the range of the target type all produce undefined results.

String.StringToDecimal: PROCEDURE [s: LONG STRING] RETURNS [INTEGER];

String.StringToOctal: PROCEDURE [s: LONG STRING] RETURNS [UNSPECIFIED];

StringToDecimal is equivalent to **StringToNumber[s, 10]**. **StringToOctal** is equivalent to **StringToNumber[s, 8]**.

String.StringToLongNumber: PROCEDURE [s: LONG STRING, radix: CARDINAL \leftarrow 10]
RETURNS [LONG UNSPECIFIED];

StringToLongNumber is analogous to **StringToNumber**, except that returns a **LONG UNSPECIFIED** instead of an **UNSPECIFIED**.

String.AppendNumber: PROCEDURE [s: LONG STRING, n, radix: CARDINAL \leftarrow 10];

AppendNumber converts the value of *n* to text using *radix* and appends it to *s*. *radix* should be in the interval [2..36].

String.AppendDecimal: PROCEDURE [s: LONG STRING, n: INTEGER];

AppendDecimal converts the value of *n* to radix 10 text and appends it to *s*. A leading minus sign will be supplied as appropriate.

String.AppendOctal: PROCEDURE [s: LONG STRING, n: UNSPECIFIED];

AppendOctal converts the value of *n* to radix 8 text and appends it to *s*. A "B" will be appended.

String.AppendLongNumber: PROCEDURE [s: LONG STRING, n: LONG UNSPECIFIED,
radix: CARDINAL \leftarrow 10];

AppendLongNumber is analogous to **AppendNumber**.

String.AppendLongDecimal: PROCEDURE [s: LONG STRING, n: LONG INTEGER];

AppendLongDecimal is analogous to **AppendDecimal**.

7.3.3.1 String operations that perform storage allocation

String.MakeString: PROCEDURE [z: UNCOUNTED ZONE, maxLength: CARDINAL]
RETURNS [LONG STRING];

The procedure **MakeString** returns a string large enough to contain **maxLength** characters, allocated from the zone **z**.

String.MakeMDSString: PROCEDURE [z: MDSZone, maxLength: CARDINAL] RETURNS [STRING];

The procedure **MakeMDSString** returns a string large enough to contain **maxLength** characters, allocated from the MDS zone **z**.

String.FreeString: PROCEDURE [z: UNCOUNTED ZONE, s: LONG STRING];

The procedure **FreeString** deallocates the string **s** to the zone **z**. The string must either be **NIL** or have been allocated from **z**.

String.FreeMDSString: PROCEDURE [z: MDSZone, s: STRING];

The procedure **FreeMDSString** deallocates the string **s** to the MDS zone **z**. The string must either be **NIL** or have been allocated from **z**.

String.AppendCharAndGrow: PROCEDURE [to: LONG POINTER TO LONG STRING, c: CHARACTER, z: UNCOUNTED ZONE];

The **AppendCharAndGrow** procedure appends the character **c** onto the string pointed to by **to**. Automatic expansion of the string is provided when required, that is, a new string will be allocated and the old will be returned to the zone **z**. **to** must point to a string allocated from the zone **z**, and the client should have no other outstanding references to **to**↑.

String.AppendExtensionIfNeeded: PROCEDURE [
 to: LONG POINTER TO LONG STRING, extension: LONG STRING, z: UNCOUNTED ZONE]
RETURNS [BOOLEAN];

The **AppendExtensionIfNeeded** procedure checks the passed string pointed to by **to** to see if it contains an extension (contains a period followed by at least one character). If not, it appends **extension** (inserting a period if **extension** does not begin with a period). Automatic expansion of the string is provided when required, that is, a new string will be allocated and the old will be returned to the zone **z**. **to** must point to a string allocated from the zone **z**, and the client should have no other outstanding references to **to**↑. **AppendExtensionIfNeeded** returns **TRUE** if the extension was added and **FALSE** if not.

String.AppendStringAndGrow: PROCEDURE [to: LONG POINTER TO LONG STRING,
 from: LONG STRING, z: UNCOUNTED ZONE, extra: CARDINAL ← 0];

The **AppendStringAndGrow** procedure appends the string **from** to the string pointed to by **to**. Automatic expansion of the string is provided when required, that is, a new string will be allocated and the old will be returned to the zone **z**. If the string must be expanded, it will be expanded to the new required length plus **extra**. **to** must point to a string allocated from the zone **z**, and the client should have no other outstanding references to **to**↑.

String.CopyToString:
PROCEDURE [s: LONG STRING, z: UNCOUNTED ZONE, longer: CARDINAL ← 0]
RETURNS [newS: LONG STRING];

The **CopyToString** procedure copies a string into a new string allocated from the zone **z**. The new string will be made **longer** characters longer than the length of **s**. If **s** is **NIL** and **longer** is zero, **newS** will be **NIL**.

String.ExpandString:

```
PROCEDURE [s: LONG POINTER TO LONG STRING, longer: CARDINAL, z: UNCOUNTED ZONE];
```

The **ExpandString** procedure expands a string by **longer** characters. **s** must point to a **STRING** allocated from zone **z**..

String.Replace:

```
PROCEDURE [to: LONG POINTER TO LONG STRING, from: LONG STRING, z: UNCOUNTED ZONE];
```

The **Replace** procedure replaces the string pointed to by **to** with a copy of the string **from**. **to** will be automatically expanded or shortened as needed, that is, a new string will be allocated and the old will be returned to the zone **z**. If **from** is **NIL**, **to** will be **NIL**. **to** must point to **NIL** or to a string allocated from the zone **z**, and the client should have no other outstanding references to **to** ↑.

7.4 Time

Time: DEFINITIONS ...;

The Time package provides functions to acquire and edit times into strings. The Time package is Product Common Software.

The implementation module is **TimeImpl.bcd**.

7.4.1 Binding

This package uses the String package and must be bound with **StringsImplA.bcd**.

7.4.2 Operations

Time.TimeZoneStandard:TYPE = {Alto, ANSI};

The **ANSI** time zone standard labels time zones by the number of hours each zone is *ahead* of GMT. The **Alto** standard uses the number of hours *behind* GMT. For example, the eastern standard time zone is represented as +5 in the Alto standard, and -5 in the ANSI standard. **Alto** is retained for Alto-based protocol compatibility only.

The current time and date is kept in a record of the following form:

```
Time.Unpacked: TYPE = RECORD[
    year: [0..2104], month: [0..12], day: [0..31],
    hour: [0..24], minute: [0..60], second: [0..60],
    weekday: [0..6], dst: BOOLEAN, zone: System.LocalTimeParameters];
```

Time.Packed: TYPE = System.GreenwichMeanTime;

The fields are filled by procedures described below which operate on the time and date as kept internally by Pilot. **year** = 0 corresponds to 1968. For **month**, January is numbered

0, etc. Days of the month have their natural assignments. For **weekday**, Monday is numbered 0. **zone** indicates time zones. **Packed** is retained for Alto compatibility.

Time.Current: PROCEDURE RETURNS [time: System.GreenwichMeanTime];

Time.Unpack: PROCEDURE [time: System.GreenwichMeanTime ← Time.defaultTime,
ltp: Time.LTP ← Time.useSystem]
RETURNS [unpacked: Time.Unpacked];

Time.LTP: TYPE = RECORD [
r: SELECT t: * FROM
useSystem = > [],
useThese = > [ltp: System.LocalTimeParameters]
ENDCASE];

useSystem: useSystem Time.LTP = [useSystem[]];
useGMT: useThese Time.LTP = [useThese[[west, 0, 0, 0, 0]]];

Time.defaultTime: System.GreenwichMeanTime = System.gmtEpoch;

Time.Invalid: ERROR;

Current is equivalent to **System.GetGreenwichMeanTime**. **Unpack** takes the Pilot-standard Greenwich mean time and a target time zone and computes the values for the fields in **Unpacked**. Passing **defaultTime** returns the current time. If **Pack** gets bad data, **Time.Invalid** is raised. If the local time parameters are not available to Pilot, **System.LocalTimeParametersUnknown** is raised.

Caution: In UtilityPilot, **System.SetLocalTimeParameters** must be called before using **Unpack**.

The operation

Time.Pack: PROCEDURE [unpacked: Time.Unpacked, useSystemLTP: BOOLEAN ← TRUE]
RETURNS [time: System.GreenwichMeanTime];

converts an **Unpacked** into the Pilot-standard **GreenwichMeanTime**. If the local time parameters are not available to Pilot, **System.LocalTimeParametersUnknown** is raised.

The operation

Time.Append: PROCEDURE [s: LONG STRING, unpacked: Time.Unpacked,
zone: BOOLEAN ← FALSE, zoneStandard: Time.TimeZoneStandard ← ANSI];

appends the time in human readable form to **s**. It adds the time zone if **zone** is **TRUE**.

The operation

Time.AppendCurrent: PROCEDURE [s: LONG STRING, zone: BOOLEAN ← FALSE,
ltp: Time.LTP ← Time.useSystem, zoneStandard: TimeZoneStandard ← ANSI];

is equivalent to **Time.Append[s, Time.Unpack[Time.defaultTime, ltp], zone, zoneStandard]**.

7.5 Memory stream

MemoryStream: DEFINITIONS...;

MemoryStream is a Pilot byte stream implementation that sources or sinks its bytes from a client specified block of virtual memory. A primary application is to support clients of **Courier.SerializeParameters** and **DeserializeParameters**.

7.1.1 Errors

IndexOutOfRangeException: ERROR;

Attempting to set the position of the stream, either explicitly with **MemoryStream.SetIndex**, or implicitly with **put** operation, beyond the limits of the **Environment.Block** specified in the **Create** will cause **IndexOutOfRangeException** to be raised.

7.1.2 Procedures

MemoryStream.Create: PROCEDURE [b: Environment.Block] RETURNS [sH: Stream.Handle];

Create defines the block of virtual memory upon which subsequent stream operations may operate. **MemoryStream** makes no assertions about the content of that block of memory.

The **Environment.Block** specified in **Create** limits the acceptable values for positioning operations as well as the amount of data that may be put to the stream (see section 1.1).

MemoryStream.Destroy: PROCEDURE [sH: Stream.Handle];

Destroy deletes the state used to support the stream instance. It does not affect the content or existance of the block of virtual memory specified in the **Create**.

Note: **Destroy** may also be accessed via the stream object's **delete** procedure.

MemoryStream.SetIndex: PROCEDURE [sH: Stream.Handle, position: Stream.Position];

SetIndex sets the position of stream for the next data operation. Attempting to set a position beyond the limits of the block specified in the **Create** will cause the error **IndexOutOfRangeException** to be raised (see section 1.1)

Note: **SetIndex** may also be accessed via the stream object's **setPosition** procedure.

MemoryStream.GetIndex: PROCEDURE [sH: Stream.Handle] RETURNS [position: Stream.Position];

GetIndex returns the current position of the stream. The usual application for this information is again in conjunction with **Courier.SerializeParameters** and is used to find the length of serialized data.

Note: **GetIndex** may also be accessed via the stream object's **getPosition** procedure.

System Generation and Initialization

This section is a general description of the organization of Pilot and its related components and of the various aspects of system initialization. It addresses the topics:

- what the components of a release of Pilot are
- the various aspects of initializing Pilot
 - these pertain to the routine operation of Pilot and client programs in an already established environment
- the special considerations of initializing an environment on a new machine or disk
- the general areas of initializing a communication network
- the general areas of introducing a new machine into a network

8.1 System components

There are seven kinds of software components in a release of Pilot of interest to the client programmer:

The Pilot kernel: Pilot is released as **PilotKernel.bcd**, a file containing the object code of the fundamental parts of the Pilot operating system. Pilot imports the *device faces* from the heads (below) and exports most of the interfaces described in this manual. UtilityPilot is a variant of the Pilot kernel which is released as **UtilityPilotKernel.bcd**. It is intended to support small applications and utilities which must run in real memory. (see Appendix D for more details);

The Communication package: the code allowing Pilot clients to perform inter- and intra-processor communication.

The heads: for each processor, one or more files containing the object code of the modules which export the device faces.

The germ: a bootstrap loader which can load a Pilot boot file into a Mesa processor and place it into execution. There are one or more germs for each kind of processor. Programmers normally have no direct contact with the germ.

Microcode: the code which, together with the heads, implements the Mesa processor on a given kind of hardware. Programmers normally have no direct contact with microcode.

The optional packages: a collection of object files containing the object code of various packages released with and used in conjunction with Pilot.

Development tools: a collection of Pilot boot files and object files which provide support for developing Pilot-based software. Among these are CoPilot, the debugger; Tajo, an executive and environment for general purpose programming; and Othello, the Pilot disk and volume utility.

The documentation accompanying a Pilot release describes in detail the file names of the available components, the functions they implement, and the interfaces they export. Please refer to that documentation for details.

Caution: There may be a number of interfaces which are exported by the Pilot components, but are not documented in this manual. They exist for the convenience of the implementation and for special purposes outside the scope of this document. *Unauthorized use of these interfaces is not supported and is strongly discouraged.* They are subject to change without general notice or review, and projects which use them improperly are subject to considerable risk from one release of Pilot to the next.

8.2 Pilot initialization

The primary method of preparing a Pilot client system for operation is to bind it with `PilotKernel.bcd`, the appropriate heads, and the desired optional packages into a single object file representing the whole system. This object file is then processed by a program called `MakeBoot`, described in the *Mesa User's Guide*, to create a boot file. The boot file may be installed on a rigid disk, floppy disk, or Ethernet server for loading in response to some hardware operation, or it may be invoked by software using the facilities of the `TemporaryBooting` interface. If the boot file is invoked by software, it is possible for the invoking program to pass a limited form of parameters called *switches* for interpretation by the booted system.

An alternative method of invoking a program is to boot a system and cause that to *load* the object file of the desired program, using the facilities in the `Runtime` interface, which are implemented by `RuntimeLoader.bcd`. This is especially appropriate if the same boot file can load a lot of different programs or if the programs being loaded are under development and constantly evolving. For example, the Mesa development environment, provides facilities for the user to dynamically load programs.

When a boot file is invoked, the state of the processor is reset. The part of the boot file representing initially resident code and data is copied into memory and the virtual memory mapping hardware is set accordingly. The configuration of I/O devices and of real memory must be determined and tables established accordingly. The heads must be initialized to reset the I/O devices. Then Pilot begins to execute. It opens the system physical and logical volumes, creates or finds certain files for its own use, creates and maps spaces for code and data, scavenges volumes if necessary, and performs other necessary initialization functions. Initialization of Pilot on a new or recently erased volume typically takes a bit longer than initialization of an established volume where the various files and control information already exist.

Pilot (i.e., `PilotKernel.bcd`) initializes disks containing Pilot volumes as follows: the *system volume* is the logical volume on which the boot file resides. The physical volume containing the system volume is automatically brought on-line and the system logical

volume is opened. Clients may bring other physical volumes on-line and open the logical volumes contained on them, and they may take existing physical volumes offline after first closing all of the contained logical volumes. (It is not meaningful to close the system volume, as Pilot uses this for its own operation.)

UtilityPilot, on the other hand, assumes that there is *no* system volume, and no volumes are brought on-line at initialization time. This is necessary so that a client can initialize a new disk to be a physical volume without first depending upon it. Once a disk is formatted to be a physical volume, it may be brought on-line in the usual way. Initialization of volumes is described in the next section.

Finally, after initialization is complete, Pilot starts the client by calling the procedure **Run** from the interface **PilotClient**. This is the only procedure imported by Pilot from the client system.

It is intended to eventually provide a facility whereby the state of a running system can be captured in a boot file for later or repeated restart. This facility will be useful for reducing the initialization time of both Pilot and client once the operating environment is established. The normal mode of operation will be for a boot file created by MakeBoot to initialize the Pilot and client environment, to create files and gather information as necessary, then to take a snapshot of this state on a second boot file. The second boot file would be the one installed for normal booting when the system element is turned on or restarted. There are a number of constraints in this mode of operation, not all of which are fully understood at this time. Among them are:

The boot file created this way is valid only on the system element on which it is created and only while the hardware configuration remains the same. It will be invalidated if the amount of memory changes, the processor ID (i.e., the electronic serial number from which all Universal ID's are made) is changed as a result of repairs, if critical devices are removed, etc.;

Files which are known or mapped at the time the boot file was created must not be deleted subsequently;

There should be no outstanding activity on any of the devices;

There should be no outstanding connections or activity in the communication network at the time this special boot file is created.

Thus, such a boot file is specific to the machine and circumstances in which it is created. It is therefore called a *local boot file*. By contrast, a boot file created by MakeBoot may be transported to any machine (of the right configuration) and executed there. Such files are called *universal boot files*.

8.3 Volume initialization

FormatPilotDisk: DEFINITIONS . . . ;

OthelloOps: DEFINITIONS . . . ;

There are several steps in initializing a disk for use as a Pilot volume:

The disk must be *formatted* into sectors corresponding to Pilot pages with appropriate headers, labels, and data blocks;

The disk must be scanned, any unusable pages must be recorded, and a physical volume must be created;

One or more logical volumes must be created on the physical volume;

Various microcode, germ, and boot files must be copied onto the logical volumes and, pointers must be set to indicate that these files be invoked when the machine is booted.

In the development environment, formatting is normally done by EIDisk (the disk diagnostic); all other initialization is done by Othello, the disk utility. Product application have their own UtilityPilot-based disk initialization utilities. Applications may also provide facilities in their Pilot-based systems for initializing, for example, removable volumes as part of routine operation.

An important part of formatting a disk is to scan the disk for unusable pages (the format package provides a scanning procedure) and to mark them as bad. Pilot will avoid placing any data or control information on such bad pages for the life of the physical volume. A page of a physical volume may be marked bad at a later time, but this will cause the information on that page to be lost. (The facilities of the **Scavenger** interface (see §4.4) can be used to recover some of the lost information.) Note that a characteristic of rigid disks is that a disk is expected to have some unusable pages at the time of manufacture, but that the rate of pages going bad during operation over the life of the disk is expected to be infinitesimal.

The **Volume** interface provides facilities for creating logical volumes on a physical volume. A logical volume has a *volume type* indicating its intended use to contain normal Pilot clients, the debugger, the debugger's debugger, or for non-Pilot purposes. Logical volumes of different types are kept separate by Pilot so that a system will not affect its debugger. Once a logical volume has been created, it may be opened and files copied onto it.

Finally, a disk may need to be prepared for booting. There are typically four kinds of files that need to be fetched to the disk: the initial microcode, the Pilot microcode, the germ, and the boot file. The initial microcode is microcode that typically lives in a special place on the disk (outside any logical volume) and is invoked by the hardware booting logic of the machine. It is the program that reads the Pilot microcode and the germ from the disk. The Pilot microcode is the main microcode for the operation of the machine, and lives in a file on a logical volume, along with the germ and boot file. A formatting package provides the facility for installing the initial microcode (since its location is specific to the type of device), and the interface **OthelloOps** provides facilities for installing and setting pointers to the microcode, germ, and boot files. These pointers are necessary so that the initial microcode can find the Pilot microcode and germ, and so that the germ can find the Pilot boot file.

This section describes those interfaces and object files distributed with Pilot that allow clients to create their own volume initializers. `OthelloOpsImpl.bcd` implements the `OthelloOps` operations, and `FormatPilotDiskImpl.bcd` implements the `FormatPilotDisk` operations. Both packages are clients of Pilot and UtilityPilot.

8.3.1 Formatting physical volumes

Before a physical volume can be presented to the `CreatePhysicalVolume` operation for the first time, it must be *formatted* into sectors corresponding to Pilot pages with appropriate headers, labels and data blocks. As a side effect, formatting finds many of the bad pages on the disk so that they can be marked as bad *after* a Pilot physical volume has been created.

Pilot disk families are formatted using the following operation

```
FormatPilotDisk.RetryLimit: TYPE = [0..254];  
  
FormatPilotDisk.noRetries: FormatPilotDisk.RetryLimit = 0;  
  
FormatPilotDisk.retryLimit: FormatPilotDisk.RetryLimit = LAST[FormatPilotDisk.RetryLimit];  
  
FormatPilotDisk.Format: PROCEDURE [h: PhysicalVolume.Handle,  
    firstPage: FormatPilotDisk.DiskPageNumber, count: LONG CARDINAL,  
    passes: CARDINAL ← 10, retries: FormatPilotDisk.RetryLimit ← noRetries];  
  
FormatPilotDisk.FormatBootMicrocodeArea: PROCEDURE [h: PhysicalVolume.Handle,  
    passes: CARDINAL, retries: FormatPilotDisk.RetryLimit];  
  
FormatPilotDisk.DiskPageNumber: TYPE = PhysicalVolume.PageNumber;  
  
FormatPilotDisk.NotAPilotDisk: ERROR;  
  
FormatPilotDisk.FormattingMustBeTrackAligned: ERROR;  
  
FormatPilotDisk.BadPage: SIGNAL [p: FormatPilotDisk.DiskPageNumber];
```

`Format` formats `count` pages of the disk `h` starting at page `firstPage`. If a problem occurs when verifying headers, labels, or data, `retries` is the number of times to retry the format operation on that page. `Passes` is the number of times to go over the disk for bad pages. If any are found, `BadPage` will be raised. If `h` does not denote a Pilot disk drive, `NotAPilotDisk` will be raised. If `h` denotes a drive of the SA1000 family or Quantum family, the run of pages to be formatted must start at the beginning of a track and end on the last page of a track or `FormattingMustBeTrackAligned` will be raised. `PhysicalVolume.Error[alreadyAsserted]` will be raised if the volume is online (i.e., asserted to be a Pilot volume).

`FormatBootMicrocodeArea` formats the area of the disk on `h` where microcode will reside. If Pilot is unable to install microcode on the disk drive denoted by `h`, `CantInstallUCodeOnThisDevice` is raised. See the previous paragraph for description of other parameters and errors raised.

```
FormatPilotDisk.DiskInfo: PROCEDURE [h: PhysicalVolume.Handle] RETURNS [
    firstPilotPage: FormatPilotDisk.DiskPageNumber, countPages: PhysicalVolume.PageCount,
    pagesPerTrack: CARDINAL, pagesPerCylinder: CARDINAL];
```

If **h** does not denote a Pilot Disk drive, the error **FormatPilotDisk.NotAPilotDisk** is raised. **firstPilotpage** is the first page on the device where Pilot volumes may begin. **countPages** is the total number of pages on that volume.

Note: For clients who use the **FormatPilotDisk** interface to install microcode, **NotAPilotDisk** is now raised by any procedures that previously raised **FormatPilotDisk.CantInstallUCodeOnThisDevice**.

8.3.2 Checking drives for bad pages

The following procedure permits scanning an already-formatted disk to determine if there are any bad pages on the disk. The client may then inform Pilot of these bad pages, via **PhysicalVolume.MarkPageBad**, so that Pilot will no longer reference them.

```
FormatPilotDisk.Scan: PROCEDURE [h: PhysicalVolume.Handle,
    firstPage: FormatPilotDisk.DiskPageNumber, count: LONG CARDINAL,
    retries: FormatPilotDisk.RetryLimit ← 10];
```

Scan scans the indicated section of the disk for bad pages, **retries** number of times per each bad page, and then reports them by raising the signal **BadPage**. The signal may be resumed to continue the scan. If **h** does not denote a Pilot disk drive, the error **NotAPilotDisk** will be raised. **PhysicalVolume.Error[alreadyAsserted]** will be raised if the volume is online.

8.3.3 Microcode and boot files

This section discusses boot files, which contain ready-to-run Pilot-based systems that can be loaded by a germ for execution, and microcode files, which contain the Mesa emulator for a given machine. Both boot files and microcode files must be *installed*, i.e., made known to Pilot, the germ and microcode. The **FormatPilotDisk** and **OthelloOps** interfaces provide facilities for dealing with boot files and microcode. The **TemporaryBoot** interface provides the means of actually invoking a boot file.

Note: Installing germ and microcode files on an SA800 disk is not directly supported by the current version of Pilot. They may be installed using the utility program **MakeDLionBootFloppyTool** (see *Mesa User's Guide* for details).

The lowest level of microcode is the initial microcode, the microcode that is read by the hardware booting logic of the system element. It is installed by the operation

```
FormatPilotDisk.InstallBootMicrocode: PROCEDURE [h: PhysicalVolume.Handle,
    getPage: PROCEDURE RETURNS[LONG POINTER]];
```

```
FormatPilotDisk.MicrocodeInstallFailure: SIGNAL [m: FormatPilotDisk.FailureType];
```

```
FormatPilotDisk.FailureType: TYPE = {emptyFile, firstPageBad, flakeyPageFound,
    microcodeTooBig, other};
```

The microcode is installed on the disk **h**. This operation finds sequential pages of the microcode file by repeatedly invoking **getPage**. The end of the microcode file is indicated when **getPage** returns **NIL**. The pointer returned by **getPage** must denote a resident page. If an error is found in the microcode file, **FormatPilotDisk.MicrocodeInstallFailure** is raised and the attempt to install the microcode has failed (any previous microcode is destroyed unless **emptyFile** is the error). If **FormatPilotDisk.MicrocodeInstallFailure** is resumed, **getPage** will be called until **NIL** is returned but the data will be ignored. **emptyFile** indicates that the microcode file was empty, i.e., **getPage** returned **NIL** the first time that it was called. If the first page of the microcode is bad, **firstPageBad** is raised. If some page of the disk reserved for the boot microcode is found to be unusable, **flakeyPageFound** is raised, indicating a problem with the disk. If an attempt is made to install too large a microcode file, **microcodeTooBig** will be raised. The error **other** is raised if the installation failed in some other way. If **h** does not denote a pilot disk drive, the error **FormatPilotDisk.NotAPilotDisk** will be raised. If Pilot is unable to install microcode on the disk drive denoted by **h**, **FormatPilotDisk.CantInstallUCodeOnThisDevice** is raised.

There are four types of boot files; clients may have as many of each as they desire.

```
OthelloOps.BootFileType: TYPE = {hardMicrocode, softMicrocode, germ, pilot,  
    pilotSnapshot};
```

A **softMicrocode** boot file contains Pilot microcode; it is typically loaded by the initial microcode and contains the Mesa emulation microcode. A **germ** boot file contains a germ, which is a bootstrap loader used to load a Pilot boot file and start it executing. Both **pilot** and **pilotSnapshot** boot files contain the image of a Pilot suitable for loading by a germ into a processor for execution. A **pilot** boot file is produced by **MakeBoot** and a **pilotSnapshot** boot file is produced by special facilities. **hardMicrocode** boot files are not currently used.

Before a Pilot file may be installed as a boot file, it must be made bootable by invoking

```
OthelloOps.MakeBootable: PROCEDURE [file: File.File,  
    type: OthelloOps.BootFileType, firstPage: File.PageNumber];
```

```
OthelloOps.InvalidVersion: ERROR;
```

This operation modifies **file** so that it is readable by the boot loader or microcode (the operation does not change the contents of the file, it only modifies the file labels). **file** must be writable and permanent and the logical volume that contains it must be open. If **file** is unknown to Pilot, either **File.Unknown** or **Volume.Unknown** will be raised. If the specified boot file is not compatible with the version of Pilot doing the **MakeBootable**, **InvalidVersion** will be raised. In this case, the file is still made bootable so as to permit installation of boot files with incompatible version numbers. **MakeBootable** may also raise **Volume.NotOpen** and **Volume.NotOnline**.

Before changing the size of a file that has been made bootable, the following operation should be invoked

```
OthelloOps.MakeUnbootable: PROCEDURE [file: File.File,  
    type: OthelloOps.BootFileType, firstPage: File.PageNumber];
```

The same restrictions as for **MakeBootable** apply. **file** may be deleted without invoking **MakeUnbootable** first.

Associated with every logical and physical volume is a default boot file of each type. These may be set and retrieved by invoking the operations

```
OthelloOps.SetPhysicalVolumeBootFile: PROCEDURE [file: File.File,  
type: OthelloOps.BootFileType, firstPage: File.PageNumber];
```

```
OthelloOps.SetVolumeBootFile: PROCEDURE [file: File.File,  
type: OthelloOps.BootFileType, firstPage: File.PageNumber];
```

The logical volume containing **file** must be open. If **file** is unknown to Pilot, either **File.Unknown** or **Volume.Unknown** will be raised. The information set by these operations may be retrieved by invoking

```
OthelloOps.GetVolumeBootFile: PROCEDURE [lvID: Volume.ID,  
type: OthelloOps.BootFileType]  
RETURNS [file: File.File, firstPage: File.PageNumber];
```

```
OthelloOps.GetPhysicalVolumeBootFile: PROCEDURE [pvID: PhysicalVolume.ID,  
type: OthelloOps.BootFileType]  
RETURNS [file: File.File, firstPage: File.PageNumber];
```

Logical volume **lvID** must be on-line, i.e., contained on a physical volume that is known to Pilot. If the physical volume is only partially online, **Volume.NotOnline** will be raised. If the **lvID** is not open, **Volume.NotOpen** will be raised. **Volume.NeedsScavenging** and **Volume.ReadOnly** may also be raised. If **lvID** is unknown to Pilot, **Volume.Unknown** is raised. If **pvID** is unknown to Pilot, **PhysicalVolume.Error[physicalVolumeUnknown]** is raised.

Pilot can be told to forget that a logical or physical volume has a default boot file of some type by invoking

```
OthelloOps.VoidVolumeBootFile: PROCEDURE [lvID: Volume.ID,  
type: OthelloOps.BootFileType];
```

```
OthelloOps.VoidPhysicalVolumeBootFile: PROCEDURE [pvID: PhysicalVolume.ID,  
type: OthelloOps.BootFileType];
```

Logical volume **lvID** must be open or **Volume.Unknown** will be raised. Physical volume **pvID** must be on-line or **PhysicalVolume.Error[physicalVolumeUnknown]** will be raised.

Every boot file of type **pilot** can have an explicit pointer to a debugger for that boot file, i.e., a debugger that will be invoked whenever that boot file is loaded and calls a debugger. Normally, Pilot finds a debugger on a volume of the next higher type than the volume being booted. This is not sufficient if the debugger needs to be called very early in Pilot initialization, or if the boot file is built on top of UtilityPilot, which *never* looks for a debugger.

```
OthelloOps.SetDebugger: PROCEDURE [debuggeeFile: File.File,  
debuggeeFirstPage: File.PageNumber, debugger: Volume.ID,
```

```
debuggerType: Device.Type, debuggerOrdinal: CARDINAL]
RETURNS [OthelloOps.SetDebuggerSuccess];

OthelloOps.SetDebuggerSuccess: TYPE = {success, nullBootFile, noDebugger,
cantWriteBootFile, notInitialBootFile, cantFindStartListHeader,
startListHeaderHasBadVersion, other};
```

The file **debuggeeFile** must permit writing and denote a file on a volume that is open. The first page of the boot file within the file **debuggeeFile** is denoted by **debuggeeFirstPage** (normally this is zero). The debugger will be found on the device denoted by **debuggerType** and **debuggerOrdinal**. The debugger is on volume **debugger** of the physical volume contained on that device. The returned value **success** indicates that the pointers were set.

If **nullBootFile** is returned, **debuggeeFile** is either unknown or the volume it resides on is unknown, not online or not open. If no installed debugger can be found on **debugger**, **noDebugger** is returned. If Pilot is unable to modify the boot file denoted by **debuggeeFile**, **cantWriteBootFile** is returned. The boot file denoted by **debuggeeFile** must not be a restart file since they can not have their debugger pointers set. If it is, **notInitialBootFile** is returned. A return of **cantFindStartListHeader** indicates that the boot file header has probably been damaged, or that the boot file has been shortened. If the specified boot file was created by an earlier version of either Pilot or MakeBoot, Pilot is unable to access it and **startListHeaderHasBadVersion** is returned. If Pilot is unable to set the debugger pointers for some other reason (i.e., the boot file is too short, missing pages exist, or the bootfile is of the wrong version), this operation will return **other**. If **debugger** is unknown to Pilot, **Volume.Unknown** will be raised.

8.3.4 Miscellaneous operations

A Pilot physical volume consists of the pieces of one or more logical volumes. Each such piece is known as a *subvolume*. The subvolumes on a physical volume can be enumerated by invoking

```
OthelloOps.GetNextSubVolume: PROCEDURE [pvID: PhysicalVolume.ID,
thisSv: OthelloOps.SubVolume]
RETURNS [nextSv: OthelloOps.SubVolume];

OthelloOps.SubVolume: TYPE = RECORD [lvID: Volume.ID,
subVolumeSize: Volume.PageCount,
firstLVPgNumber: OthelloOps.LogicalVolumePageNumber,
firstPVPageNumber: PhysicalVolume.PageNumber];

OthelloOps.LogicalVolumePageNumber: TYPE = LONG CARDINAL;

OthelloOps.nullSubVolume: OthelloOps.SubVolume = [volume.nullID, 0, 0, 0];

OthelloOps.SubVolumeUnknown: ERROR [sv: OthelloOps.SubVolume];
```

This operation is a stateless enumerator and begins and ends with **nullSubVolume** is the argument and ends when **nullSubVolume** is the result. If **thisSv** can not be found on **pvID**, **SubVolumeUnknown** is raised. A **SubVolume** identifies a logical volume, **lvID**. The number of pages that this piece of that logical volume contains is given by **subVolumeSize**.

The subvolume begins at page number **firstLVPageNumber** within **lvID**, and at page number **firstPVPageNumber** within **pvID**. If **pvID** is unknown to Pilot, **PhysicalVolume.Error[physicalVolumeUnknown]** is raised.

Note: This operation is designed to deal with logical volumes that span multiple physical volumes. Since the current version of Pilot does not provide the facility to create such logical volumes, **firstLVPageNumber** is always 0, and **subVolumeSize** always gives the actual size of **lvID**.

Pilot reserves the right to delete some or all temporary files on a logical volume when that volume is opened for writing. The following operation is guaranteed to delete *all* temporary files on a logical volume.

OthelloOps.DeleteTempFiles: PROCEDURE [Volume.ID];

OthelloOps.VolumeNotClosed: ERROR;

The specified volume must be closed or **VolumeNotClosed** will be raised. **Volume.Unknown**, **Volume.ReadOnly**, **Volume.NotOnline**, **Volume.NeedsScavenging** may be raised by this procedure.

The number of pages available on a storage device on a given drive holds is given by

**OthelloOps.GetDriveSize: PROCEDURE [h: PhysicalVolume.Handle]
RETURNS [nPages: LONG CARDINAL];**

The following operation converts a character string denoting which switches should be down when booting a boot file into a **System.Switches**.

**OthelloOps.DecodeSwitches: PROCEDURE [switchString: LONG STRING]
RETURNS [switches: System.Switches];**

OthelloOps.BadSwitches: ERROR;

The semantics of the switch string passed to **DecodeSwitches** as follows: the characters "-" and "~" mean set the next specified switch to **System.UpDown[up]**; a phrase of the form "\xxx", exactly three in length, is interpreted as the octal value of the switch that is to be set. Note that the order of switches is significant in that only the last (rightmost) setting (or clearing) of a particular switch is retained. Thus, the switches "ab~a", "ab-a" and "b" are all equivalent. If a character is not a valid switch name, **BadSwitches** is raised.

It is possible to set default switches in boot files and to associate an expiration date with a boot file:

OthelloOps.SetGetSwitchesSuccess: TYPE = OthelloOps.SetDebuggerSuccess[success..other];

**OthelloOps.GetExpirationDateSuccess: TYPE =
OthelloOps.SetDebuggerSuccess[success..other];**

**OthelloOps.SetExpirationDateSuccess: TYPE =
OthelloOps.SetDebuggerSuccess[success..other];**

```
OthelloOps.GetExpirationDate: PROCEDURE [file: File.File, firstPage: File.PageNumber]
RETURNS [OthelloOps.GetExpirationDateSuccess, System.GreenwichMeanTime];
```

```
OthelloOps.SetExpirationDate: PROCEDURE [file: File.File, firstPage: File.PageNumber
expirationDate: System.GreenwichMeanTime]
RETURNS [OthelloOps.SetExpirationDateSuccess];
```

```
OthelloOps.GetSwitches: PROCEDURE [file: File.File, firstPage: File.PageNumber]
RETURNS [OthelloOps.GetSwitchesSuccess, System.Switches];
```

```
OthelloOps.SetSwitches: PROCEDURE [file: File.File,
firstPage: File.PageNumber, switches: System.Switches]
RETURNS [OthelloOps.SetSwitchesSuccess];
```

The expiration date is used as a validity check on the processor clock. When a boot file is booted, Pilot attempts to ensure that the processor clock is set correctly. If the processor clock can not be set from the Ethernet, or is not set to a time less than or equal to the boot file's expiration date, Pilot will refuse to boot and will hang with an appropriate maintenance panel code. The logical volume on which the boot file resides must have been opened in order to invoke these procedures.

Note: These comments only apply to Pilot. For UtilityPilot, the client is always responsible for ensuring that the processor clock is set correctly.

Each boot file may also contain default boot switches. These are set and retrieved by **SetSwitches** and **GetSwitches**. When a boot file is booted, Pilot will set the system switches to the value passed to it by the client booting program if they are not equal to **System.defaultSwitches**; otherwise, it sets them to the boot file's default switches.

To aid the client in setting the processor clock to a valid value:

```
OthelloOps.IsTimeValid: PROCEDURE RETURNS [valid: BOOLEAN];
```

```
OthelloOps.SetProcessorTime: PROCEDURE [time: System.GreenwichMeanTime];
```

```
OthelloOps.GetTimeFromTimeServer: PROCEDURE RETURNS [serverTime:
System.GreenwichMeanTime, serverLTPs: System.LocalTimeParameters];
```

```
OthelloOps.TimeServerError: ERROR [error: OthelloOps.TimeServerErrorType];
```

```
OthelloOps.TimeServerErrorType: TYPE = {noCommunicationFacilities, noResponse};
```

The validity of the time in the processor clock can be ascertained by calling **IsTimeValid**. The processor clock can be explicitly set by calling **SetProcessorTime**. This is required of all UtilityPilot clients as their first action upon gaining control. The time servers on the network can be queried for their notion of the current time by calling **GetTimeFromTimeServer** which returns the time that the time servers believe it is, as well as the local time parameters that they are using. The error **TimeServerError** indicates that the attempt to access a time server failed; **noCommunicationFacilities** indicates the processor is not connected to the Ethernet; **noResponse** indicates that there was no response from any time server on the local network.

8.4 Communication initialization

Local networks are interconnected logically via machines executing an internetwork routing function. Physically, the interconnection of networks can be via a phone line link or via a processor with multiple ethernet boards. All Pilot processors contain a simple routing function, which is capable of requesting routing information from internetwork routers.

All machines running Pilot are automatically initialized to do routing. They discover their local network number(s) by broadcasting for routing information at initialization time or via routing update packets that are broadcast by internetwork routers.

There is a local network, thus network number, for every ethernet board in a Pilot processor. The network number is assigned via an administrative method that assigns unique 32-bit numbers.

When a Pilot processor is restarted, it does not know its network number. Until it is otherwise notified of a new number, it uses a default number, referred to as the unknown network number. A local network can operate correctly without an internetwork router; all the machines on the network use the same constant, unknown network number. If all machines on a network use the unknown network number in their network addresses, completely general communication is possible. If the default network number is used, there is no special communication initialization necessary to assign or discover the local network number.

8.5 Booting

TemporaryBooting: DEFINITIONS . . . ;

Pilot supports installing boot files on logical volumes, and booting from a specified file or logical volume. The operations providing this support are in the interface **TemporaryBooting**, the name of which reflects the fact that it is expected that these facilities will evolve somewhat before the final interface is frozen. Comments would be appreciated on this interface to help shape the final one.

A boot file is a client-on-Pilot configuration which has been converted by **MakeBoot** into a ready-to-run form. It is executed by loading it into a suitable processor with the Pilot boot loader, which is known as the germ. The boot file commences execution by first initializing Pilot and then invoking **PilotClient.Run**. Pilot associates a boot file with each logical, and with each physical, volume so that booting from that volume means loading the associated boot file. It is recommended that the boot file for a physical volume be the boot file for some logical volume on that physical volume, though this is not required. Pilot also provides an operation for booting directly from a file, which need not be the installed boot file of its volume.

Setting up a bootable file involves several steps. A file of the right size must be created, and its contents must be written with the boot file as produced by **MakeBoot**. Once the file is created, the operation **MakeBootable** must be applied to the file, modifying it in such a way that the germ can read it. Then the file may be booted using the operation **BootFromFile**. For this operation, installing the file is not necessary. If it is desired to associate this file with a particular logical volume, the file must be installed using the

operation **InstallVolumeBootFile**. A subsequent **BootFromVolume** operation applied to that volume (e.g., by Othello, or by a client program analogous to Othello) will cause the installed system to run. Similarly, for a physical volume, use **InstallPhysicalVolumeBootFile** to install a boot file, followed by a call on **BootFromPhysicalVolume** or **BootButton**, or by pushing the boot button.

8.5.1 Creating a boot file

A boot file is created in the normal fashion, using **File.Create**. The operations in **TemporaryBooting** are set up in such a way that a boot file may begin with a client-provided *leader* of one or more pages: in the relevant operations, a **firstPage** parameter specifies the page at which the "real" boot file (as output by **MakeBoot**) begins.

A boot file may have any file type. The interface item **TemporaryBooting.tBootFile** remains for compatibility (with a value of **FileTypes.tUntypedFile**). It will disappear in a later release.

As is always the case when creating a Pilot file, it is better to specify the actual size (i.e., the number of data pages output by **MakeBoot**, plus the number of leader pages to be prefixed) when creating it, rather than doing a series of **File.SetSize** operations. This gives Pilot the best opportunity to allocate the file in a small number of contiguous portions, which reduces both access times and storage overhead in Pilot's data structures. (See also the discussion under "Updating a boot file" below.)

8.5.2 Writing the contents of a boot file

The **Space** operations **Map**, **Unmap**, **CopyIn**, and **CopyOut** apply to bootable files just as to any other, allowing the contents to be written. Since a boot file is originally built by **MakeBoot**, which runs in a different environment, part of the process of installing a boot file is to copy it into the previously created Pilot boot file. This is typically accomplished via the Ethernet, e.g., Othello's fetch command.

8.5.3 Making a boot file bootable

Once a boot file has been created and written with the appropriate contents, it must be subjected to the operation

```
TemporaryBooting.MakeBootable: PROCEDURE [file: File.File,  
    firstPage: File.PageNumber ← 0];
```

The parameter **firstPage** specifies the first page containing the information produced by **MakeBoot**, e.g., the page following the client leader pages, or zero if no leader pages are present.

If the file doesn't contain a valid Pilot boot file starting at **firstPage**, the following error is raised:

```
TemporaryBooting.InvalidParameters: ERROR;
```

If the file being made bootable has an invalid version, the following error is raised:

TemporaryBooting.InvalidVersion: ERROR;

The file is made bootable before this error is raised so that boot files that are incompatible with Pilot 11.0 can be installed by Pilot 11.0.

8.5.4 Installing a boot file

To establish a file as the boot file of a particular logical volume, use the operation

**TemporaryBooting.InstallVolumeBootFile: PROCEDURE [file: File.File,
firstPage: File.PageNumber ← 0];**

The file should already have been made bootable. The parameter **firstPage** has the same significance as for **MakeBootable**. Note that **InstallVolumeBootFile** does not take an explicit volume parameter because a boot file may only be installed on the volume containing that file.

To associate a file as the boot file of a particular physical volume, use the operation

**TemporaryBooting.InstallPhysicalVolumeBootFile: PROCEDURE [file: File.File,
firstPage: File.PageNumber ← 0];**

8.5.5 Booting a boot file

Four operations are provided: booting a specified boot file, booting from the file installed on a specified logical volume, booting from the file installed on a specified physical volume, and simulation of the boot button. A program may boot from any Pilot-formatted volume, regardless of its type. These operations do not return. Control passes irrevocably to the new boot file.

**TemporaryBooting.BootFromFile: PROCEDURE [file: File.File,
firstPage: File.PageNumber ← 0,
switches: System.Switches ← System.defaultSwitches];**

Note: Pilot 11.0 will not successfully complete the **TemporaryBooting.BootFromFile** operation if the **file** is temporary and **firstPage** is zero.

**TemporaryBooting.BootFromVolume: PROCEDURE [volume: Volume.ID,
switches: System.Switches ← System.defaultSwitches];**

**TemporaryBooting.BootFromPhysicalVolume: PROCEDURE [volume: Volume.ID,
switches: System.Switches ← System.defaultSwitches];**

Note that the parameter to **BootFromPhysicalVolume** is not a physical volume identifier, but the identifier of any logical volume on that physical volume.

**TemporaryBooting.BootButton: PROCEDURE [
switches: System.Switches ← System.defaultSwitches];**

The value of the **defaultSwitches** parameter represents all switches as being up. Errors resulting in improper arguments to these booting operations typically result in a maintenance panel code and a crash.

8.5.6 Updating a boot file

From time to time it is necessary to install a new version of a boot file onto a volume. Several approaches are possible:

1. a new file can be created, written, made bootable, and installed; then the old boot file may be deleted.
2. an existing boot file may be overwritten with new contents; then **MakeBootable** must be applied again. **InstallVolumeBootFile** need not be reapplied.

The first approach has the advantage that it never leaves the volume in an inconsistent state. It has the disadvantage of requiring extra disk space during the time the old and new boot files exist. If the second approach is used, then before rewriting the old boot file's contents, it must be made unbootable using the operation

```
TemporaryBooting.MakeUnbootable: PROCEDURE [file: File.File,
    firstPage: File.PageNumber<-0];
```

To understand the purpose of this operation, a little background is helpful. **MakeBootable** writes an absolute disk address (called a *link*) in otherwise unused words of the label of some boot file pages. The germ uses this information rather than the ordinary Pilot volume file map structure to read the file. If the size of the boot file changes when it is being updated, new physical disk pages may be allocated, invalidating some of the old links. Thus **MakeUnbootable** is provided to remove the old links from a boot file about to be updated. Afterward, **MakeBootable** must be used to put in the correct new links.

8.5.7 Atomic saving and restoring of Pilot instances

```
TemporaryBooting.BootLocation: TYPE = RECORD [
    body: SELECT bootLocation:* FROM
        bootButton, none => NULL,
        physicalVolume => [pvLocation: TemporaryBooting.PVLocation],
        logicalVolume => [volumeLocation: TemporaryBooting.VolumeLocation],
        file => [fileLocation: TemporaryBooting.FileLocation],
    ENDCASE];
```

```
TemporaryBooting.PVLocation: TYPE [11];
```

```
TemporaryBooting.VolumeLocation: TYPE [11];
```

```
TemporaryBooting.FileLocation: TYPE [11];
```

A **BootLocation** describes a place that the state of a running Pilot may be saved in or restored from. A **bootButton BootLocation** and a **none BootLocation** are always valid; the other variants are only valid for limited periods of time as described below. The conservative approach is never to store these other variants in a permanent location but to

recreate them just before using them (as parameters to **OutloadInload**). Currently, it is only possible to save state in a file **BootLocation**.

The following procedures return a **BootLocation** for the specified location. For each operation, the circumstances under which the returned information becomes invalid are noted.

```
TemporaryBooting.GetFileLocation: PROCEDURE [file: File.File, firstPage: File.PageNumber ← 0]
    RETURNS [bootLocation: file TemporaryBooting.BootLocation];
```

The returned **BootLocation** is valid so long as the specified file is neither deleted nor has any of its attributes changed (including size and permanency). Scavenging may invalidate the returned **BootLocation** if the file was damaged and the client scavenger repaired the damage. The returned **BootLocation** is also only valid if the specified file has been made bootable (via **TemporaryBooting.MakeBootable**) and is not subsequently made unbootable. **GetFileLocation** raises **TemporaryBooting.InvalidParameters** if the specified file page is beyond the end of the file. It may also raise **File.MissingPages**, **File.Unknown**, **Volume.NotOnline**, **Volume.NotOpen**, **Volume.Unknown**.

```
TemporaryBooting.GetVolumeLocation: PROCEDURE [volume: Volume.ID]
    RETURNS [bootLocation: logicalVolume TemporaryBooting.BootLocation];
```

The returned **BootLocation** refers to the boot file installed on the logical volume. It is valid as long as the boot file on the specified volume is not deleted. The comments for the validity of returned **BootLocations** in **GetFileLocation** also apply here. **TemporaryBooting.InvalidParameters** will be raised if the specified volume does not have a Pilot boot file installed on it. **Volume.Unknown**, **Volume.NeedsScavenging**, and **Volume.NotOnline** may also be raised.

```
TemporaryBooting.GetPVLocation: PROCEDURE [volume: PhysicalVolume.ID]
    RETURNS [bootLocation: physicalVolume TemporaryBooting.BootLocation];
```

The returned **BootLocation** refers to the boot file installed on the physical volume. It is valid as long as the boot file on the specified physical volume is not deleted. The comments for the validity of returned **BootLocations** in **GetFileLocation** also apply here. **TemporaryBooting.InvalidParameters** will be raised if the specified volume does not have a Pilot boot file installed on it. **GetPVLocation** may also raise **PhysicalVolume.Error[physicalVolumeUnknown]**.

```
TemporaryBooting.OutLoadInLoad: PROCEDURE [
    outloadLocation: file TemporaryBooting.BootLocation,
    inloadLocation: TemporaryBooting.BootLocation,
    pMicrocode, pGerm: LONGPOINTER ← NIL,
    countGerm: Environment.PageCount ← 0,
    switches: System.Switches ← System.defaultSwitches];
```

TemporaryBooting.OutLoadInLoad is an atomic operation; that is, nothing happens between the outload and inload. The state of the currently running system is saved on **outloadLocation**. The system represented by **inloadLocation** is restored to a running state. The microcode and/or germ may be changed by passing the appropriate information in **pMicrocode**, **pGerm** and **countGerm**. If **pMicrocode** is defaulted, the microcode is not changed. If **pGerm** is defaulted, the germ is not changed. The switches are available to the

inloaded Pilot. These are typically examined only when the system being booted is not an outload file (e.g., it was made by MakeBoot). Note that the switches may be ignored if **inloadLocation** is a **bootButton BootLocation**. Upon return, the following sequence has occurred: (1) Pilot has successfully performed the outload and has executed the inloaded system; (2) at a later time a client, (possibly a different one), has inloaded the state of the original system (the one outloaded in (1)).

(

(

(

The Backstop

A *backstop* is a system for recording information about sick software and hardware. For product systems, it is installed instead of a debugger, and receives control in the same way and at the same times that a debugger would. When the backstop is invoked, it records the error and a restart message in a backstop log file and reboots the debuggee system. The debuggee system may then read the restart message from the backstop log and inform the user as to what has happened. The interface **Backstop** supplies facilities for implementing a backstop. The interface **BackstopNub** supplies facilities for reading entries from a log file written by a backstop.

The implementation modules are **BackstopImpl.bcd** and **BackstopNubImpl.bcd**. When these modules are used, the object files **VMMLogImpl.bcd**, **MemCacheNub.bcd**, and **BSMemCache.bcd** must also be bound in. In the following description, the term *backstop core* refers to the facilities provided by these interfaces. The term *backstop control* refers to the software built on top of it to implement a complete backstop system. Where the meaning is unambiguous, the term *backstop* may be used for either or both.

9.1 Implementing a backstop

The facilities in the **Backstop** interface are used to implement a backstop. The backstop core uses the Pilot logging facilities (**Log**) for recording the error information and the restart message in the backstop log file.

The implementation module **BackstopNubImpl.bcd** exports the interface **BackstopNub** and can be used for reading backstop logs. It uses the facilities of the **Log** and **LogFile** interfaces, so clients of **BackstopNub** must ensure that these interfaces are exported to **BackstopNubImpl**.

The following kinds of errors are reported to a backstop:

- Address faults
- Write protect faults
- Uncaught signals and errors
- Direct calls: **Runtime.CallDebugger** and **Runtime.Interrupt**

Operations are provided to determine the type of error and to record sufficient information in the log to later identify the source line in the procedure and module which caused the error. Parameters accompanying signals, errors, and direct calls are also recorded.

Additional information about the currently running processes and their call stacks can also be recorded.

9.1.1 Initializing a backstop log file

The following procedure is used to initialize a backstop log file.

```
Backstop.CreateBackstopLog: PROCEDURE [size: CARDINAL, file:File.File,  
firstPageNumber: File.PageNumber ← 0];
```

file will be initialized as a backstop log. **firstPageNumber** indicates the number of pages over which the backstop should skip before it starts writing its data.

9.1.2 Control flow

A backstop receives control when its volume is booted or when its client tries to go to the debugger. The backstop may be booted to create a new log file, read an existing log file, or perform some other maintenance task. A boot switch should be used when booting a backstop to perform the maintenance tasks so that the backstop control software can determine why it received control. Whenever the backstop is booted, control enters the backstop when Pilot calls **PilotClient.Run**.

When the backstop is installed, it may raise the signals **volume.InsufficientSpace** or **Volume.RootDirectoryError** in the process of creating its outload file.

A backstop must pass control to the debugger system by calling **Backstop.Proceed**.

```
Backstop.Proceed: PROCEDURE [boot: Volume.ID];
```

boot specifies the volume to be restarted. If **boot** is **volume.nullID**, the physical volume will be booted.

```
Backstop.VersionMismatch: SIGNAL;
```

VersionMismatch indicates that the version of Pilot in the backstop is different from that in the product system, and that the backstop may not record meaningful error information. This could occur if a new version of a debugger system was installed without also installing a compatible version of the backstop. **VersionMismatch** will be raised by the first **Backstop** procedure called that examines the client.

9.1.3 Logging errors

Procedures in this section are used to write information into the current backstop log file about the state of the product system when an error occurs. These are the only procedures that may be used to write entries into a backstop log file (do not use **Log.PutBlock**, etc.). The backstop control software may use **LogFile.Restart** to communicate with the debugger system. It may also use **LogFile.GetLost**, etc., to determine the state of the current backstop log file. These procedures may raise the signal **VersionMismatch**.

```
Backstop.LogError: PROCEDURE [];
```

```
Backstop.GetError: PROCEDURE RETURNS [BackstopNub.ErrorType];  
  
BackstopNub.ErrorType: TYPE = MACHINE DEPENDENT {  
    addressfault, writeprotectfault, signal, call, unused, interrupt, other, bug};  
  
Backstop.NotLoggingError: ERROR;
```

LogError records the type of error that caused the backstop to be invoked, along with the information necessary to locate the error in the source code and any parameters of the error. It also does a **Log.SetRestart**, recording the index of the log entry it wrote and the current time. **GetError** returns the type of the current error. These operations and all operations in this section can only be used when the backstop is invoked to process an error. If they are called when not processing an error, they will raise **NotLoggingError**.

The following procedures can be used to enumerate all of the debugger's active processes and log the state of each one. The current process can also be identified.

```
Backstop.GetNextProcess: PROCEDURE [process: Backstop.Process]  
    RETURNS [next: Backstop.Process];  
  
Backstop.GetCurrentProcess: PROCEDURE RETURNS [process: Backstop.Process];  
  
Backstop.GetFaultedProcess: PROCEDURE RETURNS [process: Backstop.Process];  
  
Backstop.LogProcess: PROCEDURE [process: Backstop.Process];  
  
Backstop.nullProcess: READONLY Backstop.Process;  
  
Backstop.Process: TYPE [1];  
  
Backstop.NotAFault: ERROR;
```

GetNextProcess is a stateless enumerator that begins and ends with **nullProcess**. Processes are returned in order beginning with the handle of the process that caused the error. **GetCurrentProcess** returns the handle of the process that caused the error. **GetFaultedProcess** returns the handle of the process that took the fault when the error type is **addressfault** or **writeprotectfault**. If **GetFaultedProcess** is called for some other error type, the signal **NotAFault** is raised. **LogProcess** records information about the state of its argument process in the current backstop log file.

Once a process is obtained, the following procedures can be used to enumerate the frames in its call stack, starting with the most recently called procedure, and log the state of each one.

```
Backstop.GetNextFrame: PROCEDURE [process: Backstop.Process, frame: Backstop.Frame]  
    RETURNS [next: Backstop.Frame];  
  
Backstop.LogFrame: PROCEDURE [frame: Backstop.Frame];  
  
Backstop.nullFrame: READONLY Backstop.Frame;  
  
Backstop.Frame: TYPE [1];
```

Passing **nullFrame** to **GetNextFrame** will return a handle for the local frame of the most recently called procedure of the process. Passing a handle so obtained will return a handle for the local frame of the next-most-recently-called procedure, and passing the handle of the root frame of the process will return **nullFrame**. **LogFrame** records information about the state of its argument frame in the current backstop log file.

9.2 Reading backstop log files

Facilities provided by the **BackstopNub** interface are used to enumerate the entries of a backstop log file and to read the information there. This might be done either by backstop control or by the debugger system. The backstop log file is implemented using the Pilot logging facilities. **Log.GetLost**, etc., may be used to determine the state of the backstop log file.

```

BackstopNub.GetNext: PROCEDURE [log: File.File, current: Log.Index,
    firstPageNumber: File.PageNumber ← 0]
    RETURNS [next: Log.Index];

BackstopNub.GetSize: PROCEDURE [log: File.File, current: Log.Index,
    firstPageNumber: File.PageNumber ← 0]
    RETURNS [size: CARDINAL];

BackstopNub.GetLogEntry: PROCEDURE [log: File.File, current: Log.Index,
    place: BackstopNub.Handle, firstPageNumber: File.PageNumber ← 0];

BackstopNub.NoErrorEntry: ERROR;

BackstopNub.Handle: LONG POINTER TO BackstopNub.ErrorEntry;

BackstopNub.ErrorEntry: TYPE = MACHINE DEPENDENT RECORD{
    globalFrame(0): BackstopNub.GlobalFrame,
    pc(1): BackstopNub.PC,
    time(2): System.GreenwichMeanTime,
    options(4): SELECT error(4): BackstopNub.ErrorType FROM
        signal => [signal(5): BackstopNub.Signal,
            msg(6): BackstopNub.SignalMsg,
            stk(7): ARRAY [0..stackSize] OF UNSPECIFIED],
        call => [msg(5): StringBody],
        unused => [],
        interrupt => [],
        addressfault => [faultedProcess(5): BackstopNub.PSBIndex],
        writeprotectfault => [faultedProcess(5): BackstopNub.PSBIndex],
        other => [reason(5): BackstopNub.SwapReason],
        bug => [bugType(5): CARDINAL],
    ENDCASE};

BackstopNub.GlobalFrame: TYPE [1];

BackstopNub.PC: TYPE [1];

BackstopNub.PSBIndex: TYPE [1];

```

BackstopNub.Signal: TYPE [2];

BackstopNub.SignalMsg: TYPE [1];

BackstopNub.SwapReason: TYPE [1];

GetNext is a stateless enumerator that begins and ends with **Log.nullIndex**. Values are returned in the order that they were written to the file. **GetSize** returns the number of words of the current entry. An entry of type **ErrorEntry** is copied into the storage provided to **GetLogEntry**. **firstPageNumber** is the number of pages over which the backstop should skip before it starts reading the data. If any of these procedures are called with an index that does not correspond to a valid backstop log entry, they raise **NotErrorEntry**.

No facilities are provided for reading process or frame entries.

(

(

(



Online Diagnostics

10.1 Communication Diagnostics

CommOnlineDiagnostics: DEFINITIONS . . . ;

This interface is used by clients of communications online diagnostics. It includes procedures for gathering ethernet statistics, running echo tests and testing RS232C and dialer facilities. All tests may be run on any host machine exporting the communications online diagnostics server.

CommOnlineDiagnostics.ServerOn: PROC;

Calling **ServerOn** causes the local machine to export the communications online diagnostics. Any of the following diagnostics can then be run on the local machine from any other machine.

CommOnlineDiagnostics.ServerOff: PROC;

Calling **ServerOff** causes the local machine to unexport the communications online diagnostics. If a client attempts to run a diagnostic on a machine that is not exporting communications online diagnostics, the error **CommError** will be raised with a reason of **noSuchDiagnostic**.

CommOnlineDiagnostics.CommError: ERROR [reason: CommErrorCode];

CommError is raised by any of the diagnostics when an error occurs in the communications used to call the diagnostics.

CommOnlineDiagnostics.CommErrorCode: TYPE = MACHINE DEPENDENT {
 transmissionMediumProblem,
 noAnswerOrBusy,
 noRouteToSystemElement,
 transportTimeout,
 remoteSystemElementNotResponding,
 noCourierAtRemoteSite,
 tooManyConnections,
 invalidMessage
 noSuchDiagnostic,

```

returnTimedOut,
callerAborted,
unknownErrorInRemoteProcedure,
streamNotYours,
truncatedTransfer,
parameterInconsistency,
invalidArguments,
protocolMismatch,
duplicateProgramExport,
noSuchProgramExport,
invalidHandle,
noError};

```

CommErrorCode defines the type of fatal error that occurred.

transmissionMediumProblem

transmissionMediumProblem indicates some sort of problem with the physical device.

noAnswerOrBusy

This error applies to circuit oriented media only and indicates that the remote end did not answer or was already busy.

noRouteToSystemElement

noRouteToSystemElement indicates the network on which the diagnostic is to be run is not reachable at this time.

remoteSystemElementNotResponding

This error code indicates the machine specified in the host parameter of the diagnostic is not responding.

tooManyConnections

This error code indicates that the maximum number of courier connections has been reached.

noSuchDiagnostic

The remote service does not export the diagnostic specified.

The rest of the error codes are translations of the Courier error codes that define Courier communication errors. See the Courier section for more details.

10.1.1 Ethernet echo testing

```

EchoDiagHandle:TYPE = LONG POINTER TO echoDiagObject;
EchoDiagObject: TYPE;

CommOnlineDiagnostics.StartEchoUser: PROC [
    targetSystemElement: System.NetworkAddress,
    echoParams: EchoParams,
    eventReporter: EventReporter ← NIL,
    host: System.NetworkAddress ← System.nullNetworkAddress]      RETURNS
    [dH:EchoDiagHandle]

```

StartEchoUser starts the echo test. Multiple echo tests may be run on the same host. The dH returned from **StartEchoUser** is the handle to be used to retrieve the echo test results.

targetSystemElement	targetSystemElement is the machine that is to be the echo server.
echoParams	echoParams are the client specified parameters for the test to be run.
eventReporter	eventReporter is the client-supplied procedure that will be called whenever an interesting event occurs. An interesting event may be when an echo response is received, or when some kind of error occurs. If the client does not wish the kind of feedback provided by the event reporter, he should set the eventReporter to NIL or let it default to NIL .
host	host is the network address of the machine that is to be used as the echo user.

```
CommOnlineDiagnostics.GetEchoResults: PROC[
  dH: EchoDiagHandle;
  host: System.NetworkAddress,
  stopIt: BOOLEAN]
RETURNS [totalsSinceStart: EchoResults,
        hist: CommOnlineDiagnostics.Histogram];
```

After starting the echo user, the client obtains the results of the test by calling **GetEchoResults**. The test is implemented with a "dead man's switch"--the client must call **GetEchoResults** within the **safetyTOInMsecs** that was passed in **StartEchoUser** for the test to actively continue. Every echo test that was started with the **StartEchoUser** proc *must* eventually be terminated by a call to **GetEchoResults** with **stopIt** set to **TRUE**, regardless of whether the test is actually sending.

dH	dH is the handle that identifies the test to retrieve the results from.
stopIt	If the procedure is called with stopIt equal to TRUE , the test will return the results and then stop. If the client wishes to obtain intermediate results of an echo test, he may call GetEchoResults with stopIt equal to FALSE , the current counters will be returned, and the test will continue to run. This is useful for real-time feedback at time intervals chosen by the client.
host	host is the network address of the machine that is the echo user.
totalsSinceStart	totalsSinceStart are the actual results of the echo user test.
hist	hist is a histogram of the timing between the sending of the echo request and the receiving of the echo reply.

```
CommOnlineDiagnostics.EchoEvent: TYPE =
{success, late, timeout, badDataGoodCRC, sizeChange, unexpected};
```

Used with **EventReporter** for client feedback, **EchoEvent** defines the type of event that has just occurred in the echo test.

success	success indicates that the echo request/response exchange was successfully completed.
late	late indicates the response to the echo request arrived late.

timeout	This event occurs when no response is received for the echo request sent. The test will timeout and send the next echo request.
badDataGoodCRC	badDataGoodCRC indicates the echo response was received without a CRC error, but some data bytes of the packet do not match the expected pattern.
sizeChange	If the test is varying the length of the data in the echo request, an event of sizeChange will occur when the size goes from the maximum back to the minimum.
unexpected	unexpected indicates that unsolicited packets were received on the echo socket before the echo test was actually started.

```
CommOnlineDiagnostics.EchoParams: TYPE = MACHINE DEPENDENT RECORD [
    totalCount(0): CARDINAL ← LAST[CARDINAL],
    safetyTOInMsecs(1): LONG CARDINAL ← 60000,
    minPacketSizeInBytes(3): CARDINAL ← 2,
    maxPacketSizeInBytes(4): CARDINAL ← 512,
    wordContents(5): WordsInPacket ← incrWords,
    constant(6): CARDINAL ← 1252528,
    waitForResponse(7): BOOLEAN ← TRUE,
    minMsecsBetweenPackets(8): CARDINAL ← 0,
    checkContents(9): BOOLEAN ← TRUE,
    showMpCode(10): BOOLEAN ← FALSE];
```

EchoParams is used by the client to define the parameters desired for the echo test.

totalCount	totalCount indicates the number of echo request/response exchanges the client wishes the test to execute. After totalCount packets have been echoed, the test will wait in an idle state for the client to terminate it and to retrieve the results via GetEchoResults . Of course, the test may be terminated at any time (i.e., before totalCount packets have been echoed) via GetEchoResults . If this number is set to 0, the test will run until stopped by GetEchoResults or by the "dead man's" switch.
safetyTOInMsecs	safetyTOInMsecs is the timeout used in the test's "dead-man's switch." After starting the echo test, GetEchoResults must be called within this time, to either reset the timeout and continue echoing or to stop the test and collect the results. If GetEchoResults is not called within this time, the test will enter an idle state. It must still be terminated via GetEchoResults .
minPacketSizeInBytes	minPacketSizeInBytes is used to specify the minimum number of data bytes to send in the echo request.
maxPacketSizeInBytes	maxPacketSizeInBytes is used to specify the maximum number of data bytes to send in the echo request. If the specified size is larger than the maximum data bytes allowed in an echo packet, it will be truncated to the maximum allowed. If maxPacketSizeInBytes is equal to minPacketSizeInBytes , the test will send constant length

	echo packets, otherwise the size will range from the minimum specified to the maximum.
wordContents	This parameter specifies what the data words in the packet will contain.
constant	The data word constant is set using the constant parameter. This parameter is used by the test only if the wordContents parameter is allConstant .
waitForResponse	If waitForResponse is TRUE , the test will not send an echo request until the reply to the previous request is received or a timeout occurs.
minMsecsBetweenPackets	The client can set the approximate interval between echo requests by specifying minMsecsBetweenPackets .
checkContents	Client may have the test verify each word in the echo response packet by specifying checkContents .
showMpCode	This parameter is currently unimplemented.

```
CommOnlineDiagnostics.EchoResults: TYPE = MACHINE DEPENDENT RECORD [
    totalAttempts, successes, timeouts, late, unexpected: LONG CARDINAL,
    avgDelayInMsecs: LONG CARDINAL,
    okButDribble, badAlignmentButOkCrc, packetTooLong, overrun, idleInput,
    tooManyCollisions, lateCollisions, underrun, stuckOutput: LONG CARDINAL];
```

Returned by the **GetEchoResults** procedure, **EchoResults** is the results of the ethernet echo test. It includes statistics obtained from the ethernet during the test.

totalAttempts	totalAttempts is the total number of echo packets that the echo user attempted to send, regardless of the number of valid responses received.
successes	successes is the total number of successful echo request/response exchanges.
timeouts	timeouts is the number of times the test sent an echo request, and did not receive the response before timing out and sending the next request.
late	late is the number of echo responses that arrived at the echo user late.
unexpected	unexpected is the number of unexpected packets that were received on the echo socket.
avgDelayInMsecs	The avgDelayInMsecs is the average time between successful echo request/response exchanges.
okButDribble, badAlignmentButOkCrc, packetTooLong, overrun, idleInput, tooManyCollisions, lateCollisions, underrun, stuckOutput	

These ethernet statistics are the number of packets found with the specified problem. Note: These statistics are only valid for echo tests using ethernets, and should be ignored for other mediums.

```
CommOnlineDiagnostics.EtherDiagError: ERROR [reason: EtherErrorReason];
```

Raised by the ethernet diagnostics, **EtherDiagError** indicates an error has occurred which prohibits the test from starting or continuing. The **reason** parameter indicates what type of fatal error has occurred.

```
CommOnlineDiagnostics.EtherErrorReason: TYPE = MACHINE DEPENDENT {
    echoUserNotThere,
    noMoreNets,
    invalidHandle};
```

EtherErrorReason defines the fatal errors that can occur in the ethernet echo test, retrieving echo counters and the gathering of ethernet statistics.

echoUserNotThere	If GetEchoResults is called when there is no echo test running on the host machine, an error will be raised with a reason of echoUserNotThere .
noMoreNets	Raised by GetEthernetStats , noMoreNets indicates that there is no existing net with the physicalOrder specified.
invalidHandle	Raised by GetEchoResults , invalidHandle indicates the client attempted to retrieve results with an already active handle.

```
CommOnlineDiagnostics.EventReporter: TYPE = PROCEDURE [event: EchoEvent];
```

Clients who wish to be notified at every echo event can implement a **EventReporter** procedure. This procedure is passed to **StartEchoUser**, and is called whenever an interesting event occurs, usually at the successful or unsuccessful completion of a echo request/response exchange.

```
CommOnlineDiagnostics.Histogram: TYPE = LONG DESCRIPTOR FOR ARRAY CARDINAL OF Detail;
Detail: TYPE = RECORD[msec, count: CARDINAL];
```

A **Histogram** is used for the data of the histogram that the echo test builds. Each element of the histogram is a **Detail**.

msec	msec is chosen by the echo test. msec for the current element of the histogram and msec for the previous element specifies an interval in which echo packets complete a round trip.
count	The count is the number of packets that were sent and returned in the interval defined by the value of msec and the value of msec for the previous element of the histogram.

```
CommOnlineDiagnostics.WordsInPacket: TYPE = MACHINE DEPENDENT {
    all0s(0), all1s(1), incrWords(2), allConstant(3), dontCare(4)};
```

The data content of the echo request is defined by **WordsInPacket**.

all0s	all0s means the words in the packet will contain zeros.
all1s	all1s means the words in the packet will contain ones.
incrWords	incrWords means each word of the packet will be incremented, starting with the first word equal to one.

allConstant	allConstant means the words in the packet will be a client specified constant.
dontCare	dontCare means the client does not care what the data content of the packet is.

10.1.2 Gathering Ethernet statistics

CommOnlineDiagnostics.EtherStatsInfo: TYPE = ARRAY StatsIndices OF LONG CARDINAL;

EtherStatsInfo is the statistics collected for the ethernet since the last system restart.

CommOnlineDiagnostics.StatsIndices: TYPE = {echoServerPkts, EchoServerBytes, packetsRecv, wordsRecv, packetsMissed, badRecvStatus, okButDribble, badCrc, badAlignmentButOkCrc, crcAndBadAlignment, packetTooLong, overrun, idleInput, packetsSent, wordsSent, badSendStatus, tooManyCollisions, lateCollisions, underrun, stuckOutput, coll0, coll1, coll2, coll3, coll4, coll5, coll6, coll7, coll8, coll9, coll10, coll11, coll12, coll13, coll14, coll15, spare};

Each item in the **StatsIndices** represents the specified ethernet statistic.

echoServerPkts	The number of packets that the machine has echoed is indicated by echoServerPkts .
EchoServerBytes	The number of bytes that the machine has echoed is indicated by EchoServerBytes .
packetsRecv	packetsRecv indicates the total number of packets that have been successfully received, including echo packets.
wordsRecv	wordsRecv indicates the total number of words that have been successfully received, including words in echo packets.
packetsMissed	packetsMissed is the number of packets that have been dropped for lack of buffering.
badRecvStatus	badRecvStatus is indicates the total number of packets that were not successfully received.
okButDribble	okButDribble indicates the number of packets that were successfully received, but had extra bits at the end.
badCrc	badCrc indicates the number of packets that were received with bad CRCs.
badAlignmentButOkCrc	badAlignmentButOkCrc indicates the number of packets that were received with correct CRCs, but did not end on byte boundaries.
crcAndBadAlignment	crcAndBadAlignment indicates the number of packets that were received with bad CRCs and did not end on byte boundaries.
packetTooLong	packetTooLong indicates the number of packets received that were long than the maximum internet size of 576 bytes.
overrun	overrun occurs when the microcode cannot take bits out of the input silo fast enough to keep up with the bits coming in of the wire.
idleInput	idleInput indicates the number of times the machine did not receive input from the ethernet for at least 40 seconds.
packetsSent	packetsSent indicates the total number of packets that have been successfully sent, including echo packets.

wordsSent	wordsSent is the total number of words sent, including those in echo packets.
badSendStatus	badSendStatus indicates the total number of packets that were not successfully sent.
tooManyCollisions	tooManyCollisions indicates the number of packets that were never sent after sixteen attempts failed because of collisions.
lateCollisions	lateCollisions indicates the number of packets which have had collisions occur in the later part of the packet (after bit 512).
underrun	underrun occurs when the microcode cannot put bits into the output silo fast enough to maintain the 10Mbit rate.
stuckOutput	stuckOutput indicates the number of times the machine was unable to send a packet in 2.5 seconds.
coll0, coll1, coll2, coll3, coll4, coll5, coll6, coll7, coll8, coll9, coll10, coll11, coll12, coll13, coll14, coll15	Each of these items indicates the number of packets that were sent after the specified number of collisions.

```
CommOnlineDiagnostics.GetEthernetStats: PROC [
    physicalOrder: CARDINAL ← 1,
    host: System.NetworkAddress ← System.nullNetworkAddress]
RETURNS [info: CommOnlineDiagnostics.EtherStatsInfo,
time: System.GreenwichMeanTime];
```

Calling **GetEthernetStats** obtains the ethernet statistics since the last system restart from the machine.

physicalOrder	physicalOrder is the number of the device on the device chain. The primary network has a physical order of one.
host	host is the machine from which to obtain the statistics.
stats	The current ethernet statistics are returned in stats .
time	time is the time the snapshot of the stats was taken. The client may make multiple calls to GetEthernetStats and use the times returned to calculate the number of echoed packets in a certain time interval.

```
CommOnlineDiagnostics.GetEchoCounters: PROC [
    host: System.NetworkAddress ← System.nullNetworkAddress]
RETURNS [packets, bytes: LONG CARDINAL, time: System.GreenwichMeanTime];
```

To obtain the number of packets which the echo server on the specified machine has echoed since the last system restart, clients may call **GetEchoCounters**. The additional parameter **host** in the **RemoteCommDiags** procedure is the network address of the machine from which to collect the echo counters.

host	host is the network address of the machine from which to collect the echo counters.
packets	packets is the number of echoed packets.
bytes	bytes is the total number of bytes the server has echoed.
time	time is the time the statistics were collected. The client may make multiple calls to GetEchoCounters and use the times

returned to calculate the number of echoed packets within a certain time interval.

10.1.3 RS232C testing

RS232C testing consists of running a loopback test that exercises and verifies the data-transmission/reception features of the RS232C channel. As the client is required to set some of the channel characteristics, he should be familiar with the EIA RS232C standard.

```
CommOnlineDiagnostics.StartRS232CTest: PROC [
    rs232cParams: RS232CParams,
    setDiagnosticLine: SetDiagnosticLine ← NIL,
    writeMsg: WriteMsg ← NIL,
    modemChange: ModemChange ← NIL,
    host: System.NetworkAddressSystem.nullNetworkAddress];
```

The test is run by calling **StartRS232CTest** and requires that a loopback plug be installed on the RS232C cable. The parameters specified by the client in the **StartRS232CTest** test are concerned with defining the transmission medium usage and the session characteristics.

Multiple RS232C tests may be run on the same machine, but only one per port. Calling **StartRS232CTest** on an already active port will result in the error **RS232CDiagError** with the code **channelInUse**.

setDiagnosticLine

setDiagnosticLine is used only by CIU diagnostic implementors for resetting the port for running the loopback test. Other clients should set it to **NIL**.

writeMsg

writeMsg is a client-supplied procedure for realtime feedback, called after a frame has been sent and received through the loopback. Clients who are not interested in this kind of feedback should set this parameter to **NIL**.

modemChange

modemChange is a client-supplied procedure for realtime feedback, called whenever any of the **ModemSignals** changes state. Clients who are not interested in this state change should set this parameter to **NIL**.

host

host is the network address of the machine on which to run the diagnostic.

```
CommOnlineDiagnostics.GetRS232CResults: PROC [
```

stopIt: BOOLEAN,

host: System.NetworkAddress ← System.nullNetworkAddress]

RETURNS [**counters:** CountType];

After starting the loopback test, the client obtains the results of the test by calling **GetRS232CResults**. The test is implemented with a "dead mans switch" - the client must call **GetRS232CResults** with **safetyTOInMsecs** that was passed to the **StartRS232CTest** procedure in order for the test to continue. Clients *must* eventually terminate the loopback test by calling **GetRS232CResults** with **stopIt** equal to true.

stopIt

If the procedure is called with **stopIt** equal to **TRUE**, the test will return the results and then terminate. If the client

wishes to obtain intermediate results of an echo test, he may call **GetRS232CResults** with **stopIt** equal to **FALSE**, the current counters will be returned, and the loopback test will continue to run.

host	host is the network address of the machine on which to run the loopback test.
counters	counters is the current results of the loopback test.

CommOnlineDiagnostics.RS232CDiagError: ERROR [reason: RS232CErrorReason];

The error **RS232CDiagError** is raised whenever a fatal error occurs during the test. The client should do the necessary clean up and end the test process.

RS232CErrorReason: TYPE = {aborted, noHardware, noSuchLine, channelInUse, unimplementedFeature, invalidParameter, invalidHandle};

The reason in the **RS232CDiagError** is defined by **RS232CErrorReason**.

aborted	aborted indicates the channel has been aborted.
noHardware	This error reason will occur if there is no RS232C hardware present or if the RS232C channel code has not been started.
noSuchLine	noSuchLine indicates a bad RS232C line number has been specified by the client.
channelInUse	If some other process is already using the RS232C port when the client attempts to start the RS232C test, the error will be raised with a reason of channelInUse .
unimplementedFeature	This error reason is used internally and should never be observed by the client.
invalidParameter	If an invalid parameter is passed to the RS232C test, the error will be raised with a reason of invalidParameter .
invalidHandle	This error indicates that the client called GetRS232CResults with a handle that had previously been deleted.

CommOnlineDiagnostics.CountType: TYPE = MACHINE DEPENDENT RECORD [sendOk, bytesSent, recOk, bytesRec, deviceError, dataLost, xmitErrors, badSeq, missing, sendErrors, recErrors: LONG CARDINAL];

CountType contains the counters used in the **RS232CLoopback** test. At the end of the test these counters are the results. The client may also check these counters during the test by calling **GetRS232CResults** with **stopIt** equal to **FALSE**.

sendOk	sendOk is a counter that reflects the number of successfully sent frames.
bytesSent	bytesSent is the current number of bytes that have been sent.
recOk	recOk reflects the number of successfully received frames.
bytesRec	bytesRec is the current number of bytes that have been sent.
deviceError	deviceError indicates the number of times data was received when no receive operation was outstanding.
dataLost	dataLost indicates the number of times that a incoming frame was too large to fit in the input buffer.

xmitErrors	xmitErrors indicates the number of frames that have been received with some sort of transmission error (e.g., checksum error, parity error, etc).
badSeq	badSeq indicates the number of times the receiver detected a frame with an unrecognizable sequence number. Generally this means that a frame has been lost or garbled during transmission.
missing	missing indicates the number of times the receiver has detected a missing frame from looking at the sequence numbers.
sendErrors	sendErrors indicates the total number of frames that have not been successfully sent.
recErrors	recErrors indicates the total number of frames that have not been successfully received.

CommOnlineDiagnostics.LengthRange: TYPE = RECORD [low, high: [0..maxData)];

The range of data length (in bytes) in the frames is defined by **LengthRange**.

CommOnlineDiagnostics.maxData: CARDINAL = 1000;

maxData is the maximum number of bytes of data in a frame.

CommOnlineDiagnostics.ModemChange: TYPE = PROC [
 modemSignal: ModemSignal; **state:** BOOLEAN];

ModemChange is a procedure type that is used by the client when he wishes to be notified when a change occurs in the state of the signals defined in **ModemSignals**.

modemSignal	modemSignal is the signal of interest.
state	state is the state of the signal.

CommOnlineDiagnostics.ModemSignal: TYPE = {dataSetReady, clearToSend,
 carrierDetect, ringIndicator, ringHeard};

ModemSignal contain the state of the corresponding circuits described in EIA Standard RS232C. They are passed to the client through the procedure **modemChange**.

CommOnlineDiagnostics.PatternType: TYPE = {zero, ones, oneZeroes, constant, byteIncr};

The **PatternType** defines the contents of the data in the frames being sent.

zero	zero indicates the contents will be all zeros.
ones	ones indicates the contents will be all ones.
constant	constant indicates the test will use a client-supplied constant in each byte of data.
byteIncr	byteIncr indicates the test will increment each byte of data in the frame, starting with a value of one.

CommOnlineDiagnostics.RS232CParams: TYPE = MACHINE DEPENDENT RECORD [
 testCount(0): CARDINAL ← LAST[LONG CARDINAL],
 safetyTOInMsecs(1): LONG CARDINAL ← 6000,

```

lineSpeed(3): RS232C.LineSpeed,
correspondent(4): RS232C.Correspondent,
lineType(5): RS232C.LineType,
lineNumber(6): CARDINAL,
parity(7): RS232C.Parity,
charLength(8): RS232C.CharLength,
pattern(9): PatternType,
constant(10): CARDINAL ← 0,
dataLengths(11): LengthRange
clockSource(13): RS232C.clockSource,
waitForDSR(14): BOOLEAN ← TRUE];

```

The parameters passed to the RS232C test are defined by the **RS232CParams**.

testCount	testCount specifies the number of frames to send/receive. If this number is set to 0, the test will run actively loopback until stopped by the GetRS232CResults or by the "dead man's" switch.
safetyTOInMsecs	safetyTOInMsecs is the timeout used in the test's "deadman's switch." After starting the RS232C test by calling StartRS232CTest , GetRS232CResults must be called within this time, to either reset the timeout and continue echoing or stop the test and collect the results.
lineSpeed	The lineSpeed is the speed of the line and should agree with the setting of the modem.
correspondent	correspondent specifies a type of system the test is to "correspond" with. The line type determines what the correspondent should be. For a line type of asynchronous, ttyHost should be used. For bit synchronous, nsSystemElement should be used, and for byte synchronous, system6 .
lineType	lineType is the type of line the channel will use.
lineNumber	lineNumber is the number of the RS232C line and should normally be set to 0. Other values apply only to processors with multiple RS232C lines.
parity	parity is the parity to use during the test.
charLength	The character data length, (excluding parity, stop and start bits), is specified by charLength and should agree with the setting on the modem.
pattern	The contents for each byte of data is specified by pattern .
constant	If the client has specified a pattern of constant, the constant parameter is used to specify what the data constant should be.
clockSource	clockSource determines whether the clock will be provided by the DTE (internal) or by the modem (external).
dataLengths	dataLengths specifies the range of data lengths to send in the frames. If the low and high are equal, the test will send constant length data. If they are not equal, the test will first send the frame of the right length, decrementing the length with each subsequent send.

waitForDSR

There are some modems in the field that do not raise DSR. This parameter enables users of such modems to tell the test to start even if DSR has not come up.

CommOnlineDiagnostics.RS232CTestMessage: TYPE = {looped,sendError,recError};

The **RS232CTestMessage** is passed to the client-supplied procedure that is called every time a frame is sent/received. The message indicates the status of the transfer.

sendError

sendError indicates there has been some sort of transmission error.

recError

If errors occurred when the frame was received, the message will indicate **recError**.

CommOnlineDiagnostics.SetDiagnosticLine: TYPE = PROC [lineNumber: CARDINAL]

RETURNS [lineSet: BOOLEAN];

SetDiagnosticLine is a type used by the CIU implementors to reset the port when diagnostics are started. **lineNumber** is the line to set; **lineSet** indicates whether the reset was successful.

CommOnlineDiagnostics.WriteMsg: TYPE = PROC [msg: RS232CTestMessage];

WriteMsg is a procedure type that is used by the client when he wishes real-time feedback during the RS232C test. The **msg** parameter indicates the type of event that just occurred.

10.1.4 Dialer testing

This test is used to verify correct operation of the RS366 hardware and an external auto-dialer. The RS366 cable must be connected to the auto-dialer.

CommOnlineDiagnostics.DialupTest: PROC[
rs232ClineNumber: CARDINAL,
phoneNumber: LONG POINTER TO Dialup.Number,**dialerType:** dialup.DialerType,
host: System.NetworkAddress ← System.nullNetworkAddress]
RETURNS [outcome: DialupOutcome];

DialupTest is called to test the dialer. The test will retry the dial a maximum of three times before returning to the client. The additional parameter **host** in the **RemoteCommDiags** procedure specifies the network address of the machine on which to run the test.

rs232ClineNumber

rs232ClineNumber specifies the line number to be used and should be set to 0. Other values apply only to processors with multiple RS232C lines.

phoneNumber

phoneNumber is the number to be used to call the foreign device. Note: The dialup implementation attaches no semantics to any of the bit patterns specified in **phoneNumber**, simply passing them to the dial hardware. Clients and/or their users must determine what the special characters (such as EON and SEP) are for their particular hardware and pass those characters to the dialup test.

dialerType **dialerType** is the type of dialing equipment being used.
outcome **outcome** is the result of the dialup test.

```
CommOnlineDiagnostics.DialupOutcome: TYPE = {
    success, failure, aborted, formatError, transmissionError, dataLineOccupied,
    dialerNotPresent, dialingTimeout, transferTimeout, otherError, noHardware,
    noSuchLine, channelInUse, unimplementedFeature, invalidParamater};
```

DialupOutcome defines the result of the **DialupTest**.

success	success indicates the dialing operation was successful. This means all the digits in the number were dialed, and control was successfully transferred to the modem.
failure	If the dialing operation resulted in no answer, a busy signal, or the telephone was answered by something other than a compatible modem, the outcome will be failure .
aborted	aborted is currently not implemented.
formatError	formatError indicates the parameter phoneNumber was formatted incorrectly.
transmissionError	transmissionError indicates the transfer of the dialing information to the dialing hardware did not succeed. This outcome indicates a hardware problem.
dataLineOccupied	dataLineOccupied indicates the telephone line to which the dialing hardware is connected is off-hook.
dialerNotPresent	This outcome indicates the lack of working dialer hardware.
dialingTimeout	An outcome of dialingTimeout indicates a hardware problem - the dialer did not respond to a request during dialing.
transferTimeout	transferTimeout indicates that no meaningful reply was received from the dialer following dialing the last digit.
otherError	This outcome indicates a hardware problem.
noHardware, noSuchLine, channelInUse, unimplementedFeature, invalidParamater	otherError means an unknown, unexpected error occurred. noHardware, noSuchLine, channelInUse, unimplementedFeature, invalidParamater These errors are used internally and should never be observed by the client.

10.2 Bitmap Display, Keyboard, and Mouse Diagnostics

OnlineDiagnostics: DEFINITIONS...;

This interface is used by clients of the bitmap Display, Keyboard, and Mouse Online Diagnostics. It includes procedures for running the bitmap Display Diagnostics, the Keyboard Diagnostics and the Mouse Diagnostics.

OnlineDiagnostics.Background: TYPE = {white, black};

Defines the background on the bitmap Display.

OnlineDiagnostics.CursorArray: TYPE = ARRAY [0..16] OF WORD;

Defines the size and bit pattern of the cursor for display on the bitmap Display.

OnlineDiagnostics.Coordinate: TYPE = MACHINE DEPENDENT RECORD [x, y: INTEGER];

The bitmap Display is addressed by x-y coordinates. The coordinate origin (0, 0) is the uppermost, leftmost pixel of the display; x increases to the right and y increases downward.

OnlineDiagnostics.KeyboardType: TYPE = {american, european, japanese};

Defines the type of keyboard being used.

OnlineDiagnostics.KeyboardAndMouseTest: PROCEDURE [
 keyboardType: OnlineDiagnostics.KeyboardType,
 screenHeight: CARDINAL [0..32767],
 screenWidth: CARDINAL [0..32767],
 SetBackground: PROC [background: OnlineDiagnostics.Background],
 SetBorder: PROC [oddPairs, evenPairs: [0..377B]],
 GetMousePosition: PROC RETURNS [OnlineDiagnostics.Coordinate],
 SetMousePosition: PROC [newMousePosition: OnlineDiagnostics.Coordinate],
 SetCursorPattern: PROC [cursorArray: OnlineDiagnostics.CursorArray],
 SetCursorPosition: PROC [newCursorPosition: OnlineDiagnostics.Coordinate],
 keyboard: LONG POINTER,
 Beep: PROC [duration: CARDINAL],
 ClearDisplay: PROC,
 BlackenScreen: PROC [x, y, width, height: CARDINAL],
 InvertScreen: PROC [x, y, width, height: CARDINAL],
 WaitForKeyTransition: PROC];

The **KeyboardAndMouseTest** procedure is used to run keyboard and mouse diagnostics using a bitmap display.

screenHeight defines the number of horizontal lines on the bitmap Display. Equivalent to **UserTerminal.screenHeight**.

screenWidth Defines the number of horizontal dots across the bitmap Display. Equivalent to **UserTerminal.screenWidth**.

SetBackground [background: ...] Lets the bitmap Display background to either white or black. Equivalent to **UserTerminal.SetBackground**.

SetBorder If the display has a border, then clients may set the pattern to be displayed in the border by calling this procedure. Equivalent to **UserTerminal.SetBorder**.

GetMousePosition[] gets the x and y values of the mouse position.

SetMousePosition modifies the x and y values of the mouse position. Equivalent to **UserTerminal.SetMousePosition**.

SetCursorPattern sets up the bit pattern of the cursor for display on the bitmap Display. Equivalent to **UserTerminal.SetCursorPattern**.

SetCursorPosition	sets the position of the cursor on the bitmap Display. Equivalent to UserTerminal.SetCursorPosition .
keyboard	Equivalent to UserTerminal.keyboard .
Beep	emits a tone from the speaker for the given duration of time. Duration is in milliseconds. Equivalent to UserTerminal.Beep .
ClearDisplay	turns the entire screen white
BlackenScreen	turns the screen black for the given width and height starting at the x/y coordinates.
InvertScreen	inverts the screen for the given width and height starting at the x/y coordinates.

WaitForKeyTransition waits for an entry from the keyboard before returning (not presently used in Star).

OnlineDiagnostics.NextAction: TYPE = {nextPattern, invertPattern, quit}; defines the next action to be taken. Used with the bitmap Display alignment pattern.

```
OnlineDiagnostics.LFDisplayTest:PROCEDURE [
  screenHeight: CARDINAL [0..32767],
  screenWidth: CARDINAL [0..32767],
  SetBackground: PROC [background: OnlineDiagnostics.Background],
  SetBorder: PROC [oddPairs, evenPairs: [0..377B]],
  GetNextAction: PROC RETURNS [OnlineDiagnostics.NextAction],
  ClearDisplay: PROC,
  BlackenScreen: PROC [x, y, width, height: CARDINAL],
  FillScreenWithObject: PROC [p: LONG POINTER TO ARRAY [0..16) OF WORD]];
```

The **LFDisplayTest** procedure displays test patterns on the display. It can be used as a bitmap Display alignment tool.

screenHeight	defines the number of horizontal lines on the bitmap Display. Equivalent to UserTerminal.screenHeight .
screenWidth	defines the number of horizontal dots across the bitmap Display. Equivalent to UserTerminal.screenWidth .
SetBackground	sets the bitmap Display background to either white or black. Equivalent to UserTerminal.SetBackground .
SetBorder	if the display has a border, then clients may set the pattern to be displayed in the border by calling this procedure. Equivalent to UserTerminal.SetBorder .
GetNextAction	gets the next action through keyboard input from the user. See OnlineDiagnostics.NextAction above.

ClearDisplay	erases the entire display to white.
BlackenScreen	turns the screen black for the given width and height starting at the x/y coordinates.
FillScreenWithObject	fills the entire screen with the bit pattern in the 16 word array.

10.3 Lear Siegler Diagnostics

OnlineDiagnostics: DEFINITIONS . . . ;

This interface is used by clients of the Lear Siegler Online Diagnostics. It includes procedures for running the Lear Siegler Diagnostic.

```
OnlineDiagnostics.LSMessage: TYPE = {kTermAdj, kTypeCharFill, kCTL, kFillScreen,
kTypeXHair, kEndAdj, kTermTest, kTestKey, kCTLStop, kLineFeed, kReturnKey, kLetter,
kAndCTL, kEscape, kSpBar, kAndShift, kShColon, kShSemiColon, kTypeComma,
kHyphen, kTypePeriod, kVirgule, kNumeral, kKey, kLearColon, kSemiColon, kShComma,
kShHyphen, kShPeriod, kShVirgule, kAtSign, kLeftBracket, kBackSlash, kRightBracket,
kCaret, kBreak, kShAt, kShLeftBracket, kShBackSlash, kShRightBracket, kShCaret,
kShBreak, kUnknown};
```

Defines the message displayed on the screen when the given character is entered from the keyboard.

```
OnlineDiagnostics.LSAdjust: PROCEDURE [
  cancelSignal: SIGNAL,
  GetMesaChar: PROC RETURNS [CHARACTER],
  PutCR: PROC,
  PutMessage: PROC [message: OnlineDiagnostics.LSMessage, char: CHARACTER ← 0C],
  PutMesaChar: PROC [char: CHARACTER]];
```

The LSAdjust procedure allows the user to adjust the Lear Siegler Display.

cancelSignal	is raised when the user enters a 'Control C' on the keyboard. Equivalent to NSCommand.Cancel .
GetMesaChar	gets the character entered on the keyboard by the user. Equivalent to NSCommand.GetMesaChar .
PutCR	outputs a carriage return to the Lear Siegler Display. Equivalent to NSCommand.PutCR[TRUE] .
PutMessage	displays the given message on the Lear Siegler Display. Equivalent to NSCommand.PutLine . Note: The default for char is used for the Lear Siegler diagnostic.
PutMesaChar	outputs a character to the Lear Siegler Display. Equivalent to NSCommand.PutMesaChar .

```
OnlineDiagnostics.LSTest: PROCEDURE [
  cancelSignal: SIGNAL,
  GetMesaChar: PROC RETURNS [CHARACTER],
  PutMessage: PROC [message: OnlineDiagnostics. LSMessage, char: CHARACTER ← 0C]];
```

The **LSTest** procedure allows the user to test the Lear Siegler Display or equivalent.

cancelSignal	is raised when the user enters a 'Control C' on the keyboard. Equivalent to NSCommand.Cancel .
GetMesaChar	gets the character entered on the keyboard by the user. Equivalent to NSCommand.GetMesaChar .
PutMessage	displays the given message on the Lear Siegler Display. Equivalent to NSCommand.PutLine . Note: For the diagnostic the default for char is taken.

10.4 Floppy Diagnostics

```
OnlineDiagnostics: DEFINITIONS . . . ;
```

This interface is used by clients of the Floppy Online Diagnostics. It includes procedures for running the Floppy Diagnostic.

```
OnlineDiagnostics.FloppyMessage: TYPE = {
  cFirst, cCallCSC, cCloseWn, cEnsureReady, cExit, cInsDiffCleanDisk, cInsertCleanDisk,
  cInsertDiagDisk, cInsertWriteable, cNBNotReady, cOtherDiskErr, cRemoveCleanDisk,
  cRemoveDiskette, cLast,

  hFirst, hBusy, hExpec1, hExpec2m, hCRC1, hCRC2, hCRCerr, hDelSector, hDiskChng,
  hErrDetc, hGoodComp, hHead, hHeadAddr, hIlliStat, hIncrLngth, hObser1, hObser2,
  hReadHead, hReadSector, hReadStat, hReady, hRecal, hRecalErr, hSector, hSectorAddr,
  hSectorCntErr, hSectorLngth, hSeekErr, hTimeExc, hTrack, hTrack0, hTrackAddr,
  hTwoSide, hWriteDelSector, hWritePro, hWriteSector, hLast,

  iFirst, iBadContext, iBadLabel, iBadSector, iBadTrack0, iCheckPanel, iCIERec, iCleanDone,
  iCleanProgress, iErrDet, iErrNoCRCER, iExerWarning, iFormDone, iFormProgress,
  iFormWarning, iHardErr, iHeadDataErr, iInsertDiagDisk, iInsertFormDisk, iOneSided,
  iRunStdTest, iSoftErr, iTnx, iTwoSided, iUnitNotReady, iVerDataErr, iLast,

  tFirst, tByteCnt, tCIERH, tCIERS, tCIEVer, tCIEWDS, tCIEWS, tHeadDataErr, tHeadDisp,
  tHeadErrDisp, tSectorDisp, tStatDisp, tSummErrLog, tVerDataErr, tLast,

  yFirst, yDispSects, yDispExpObsData, yDoorJustOpened, yDoorOpenNow,
  yDoorOpenShut, yIsItDiagDisk, yIsItWrProt, yStillContinue, yStillSure, yLast};
```

Define the message keys used by the Floppy Diagnostic.

```
OnlineDiagnostics.FloppyReturn: TYPE = {
  deviceNotReady, notDiagDiskette, floppyFailure,noErrorFound};
```

Defines the type of returns from some of the Floppy Diagnostics tests.

deviceNotReady is returned when the floppy drive is not ready and therefore cannot be tested.

notDiagDiskette is returned when the floppy diskette is not a Diagnostics Diskette and therefore cannot be tested because it cannot be written on.

floppyFailure is returned when a floppy hardware error is detected.

noErrorFound is returned when the test runs successfully.

OnlineDiagnostics.Field: TYPE = RECORD [

fieldName: OnlineDiagnostics.FloppyMessage, fieldValue: UNSPECIFIED];

Used for Floppy Diagnostics status display.

OnlineDiagnostics.FieldDataType: TYPE = {

boolean, cardinal, character, hexadecimal, hexbyte, integer, octal, string};

Defines the various types of data displayed by the Floppy Diagnostics .

OnlineDiagnostics.FloppyWhatToDoNext: TYPE = {

continueToNextError, loopOnThisError, displayStuff, exit};

Defines the operator options for running Floppy Diagnostics command files.

OnlineDiagnostics.SingleDouble: TYPE = {single, double};

Defines the number of sides and data density of a floppy diskette.

OnlineDiagnostics.SectorLength: TYPE = {one28, two56, five12, one024};

Defines the number of bytes in a sector of a floppy diskette. Used in Floppy Diagnostics command files.

OnlineDiagnostics.ErrorHandling: TYPE = {

noChecking, stopOnError, loopOnError, continueOnError};

Defines the operator options for the handling of floppy errors in the Floppy Diagnostics command files.

OnlineDiagnostics.DisplayFieldsProc: PROCEDURE [

fields: DESCRIPTOR FOR ARRAY OF Field,

title: OnlineDiagnostics.FloppyMessage ← tFirst,

fieldType: OnlineDiagnostics.FieldDataType,

numberOfColumns: CARDINAL ← 3];

DisplayFieldsProc displays Floppy Diagnostics status.

fields defines the names of the status bits and their boolean values.

title defines the title of the display.

fieldType defines the type of data being displayed.

numberOfColumns defines the number of columns in which to display the data.

OnlineDiagnostics.DisplayTableProc: PROCEDURE [

headers: DESCRIPTOR FOR ARRAY OF OnlineDiagnostics.FloppyMessage,

rowNames: DESCRIPTOR FOR ARRAY OF OnlineDiagnostics.FloppyMessage ,

values: DESCRIPTOR FOR ARRAY OF DESCRIPTOR FOR ARRAY OF UNSPECIFIED,
title: OnlineDiagnostics.FloppyMessage← tFirst,
fieldType: OnlineDiagnostics.FieldDataType];

DisplayTableProc displays an error/summary log.

headers defines the name of each column in the error/summary log.
rowNames defines the name of each entry in the error/summary log.
title defines the title of the error/summary log.
fieldType defines the type of data being displayed.

OnlineDiagnostics.DisplayNumberedTableProc: PROCEDURE [
values: LONG DESCRIPTOR FOR ARRAY OF UNSPECIFIED,
rowNameHeader: OnlineDiagnostics.FloppyMessage← tFirst,
title: OnlineDiagnostics.FloppyMessage← tFirst,
numOfColumns: CARDINAL,
startNum: INTEGER,
fieldType: OnlineDiagnostics.FieldDataType];

DisplayNumberedTableProc displays a table of numbers plus the number of entries displayed.

values defines the actual numbers to be displayed.
rowNameHeader defines the name of the entries displayed (Example: "Byte Count").
title defines the title of the table.
numOfColumns defines the number of columns displayed.
startNum defines the first of the number of entries displayed.
fieldType defines the type of numbers being displayed.

OnlineDiagnostics.PutMessageProc:PROCEDURE [msg: OnlineDiagnostics.FloppyMessage];

PutMessageProc displays the given message to the operator.

OnlineDiagnostics.GetConfirmationProc:PROCEDURE [
msg: OnlineDiagnostics.FloppyMessage];

GetConfirmationProc displays the given message to the operator and requests confirmation.

OnlineDiagnostics.YesNo: TYPE = {yes, no};

OnlineDiagnostics.GetYesOrNoProc:PROCEDURE [
msg: OnlineDiagnostics.FloppyMessage] RETURNS [OnlineDiagnostics.YesNo];

GetYesOrNoProc displays a message to the operator and requests a yes or no response.

OnlineDiagnostics.GetFloppyChoiceProc:PROCEDURE
 RETURNS [OnlineDiagnostics.FloppyWhatToDoNext];

GetFloppyChoiceProc gets an answer from the operator on how to proceed after an error has occurred in the command file.

OnlineDiagnostics.FloppyExerciser:PROCEDURE [
displayFields: OnlineDiagnostics.DisplayFieldsProc,
displayTable: OnlineDiagnostics.DisplayTableProc,

```
displayNumberedTable: OnlineDiagnostics.DisplayNumberedTableProc,  
putMessage: OnlineDiagnostics.PutMessageProc,  
getConfirmation: OnlineDiagnostics.GetConfirmationProc,  
getYesOrNo: OnlineDiagnostics.GetYesOrNoProc,  
getFloppyChoice: OnlineDiagnostics.GetFloppyChoiceProc];
```

FloppyExerciser thoroughly exercises the floppy disk hardware. See arguments described above.

```
OnlineDiagnostics.FloppyStandardTest:PROCEDURE [  
    displayFields: OnlineDiagnostics.DisplayFieldsProc,  
    displayTable: OnlineDiagnostics.DisplayTableProc,  
    displayNumberedTable: OnlineDiagnostics.DisplayNumberedTableProc,  
    putMessage: OnlineDiagnostics.PutMessageProc,  
    getConfirmation: OnlineDiagnostics.GetConfirmationProc,  
    getYesOrNo: OnlineDiagnostics.GetYesOrNoProc,  
    getFloppyChoice: OnlineDiagnostics.GetFloppyChoiceProc]  
    RETURNS [floppyReturn: OnlineDiagnostics.FloppyReturn];
```

FloppyStandardTest runs a nondestructive floppy disk diagnostic. See arguments described above.

```
OnlineDiagnostics.FloppyCleanReadWriteHeads:PROCEDURE [  
    displayFields: OnlineDiagnostics.DisplayFieldsProc,  
    displayTable: OnlineDiagnostics.DisplayTableProc,  
    displayNumberedTable: OnlineDiagnostics.DisplayNumberedTableProc,  
    putMessage: OnlineDiagnostics.PutMessageProc,  
    getConfirmation: OnlineDiagnostics.GetConfirmationProc,  
    getYesOrNo: OnlineDiagnostics.GetYesOrNoProc,  
    getFloppyChoice: OnlineDiagnostics.GetFloppyChoiceProc]  
    RETURNS [floppyReturn: OnlineDiagnostics.FloppyReturn];
```

FloppyCleanReadWriteHeads cleans the read/write heads of the floppy disk drive. See arguments described above.

```
OnlineDiagnostics.FloppyFormatDiskette:PROCEDURE [  
    displayFields: OnlineDiagnostics.DisplayFieldsProc,  
    displayTable: OnlineDiagnostics.DisplayTableProc,  
    displayNumberedTable: OnlineDiagnostics.DisplayNumberedTableProc,  
    putMessage: OnlineDiagnostics.PutMessageProc,  
    getConfirmation: OnlineDiagnostics.GetConfirmationProc,  
    getYesOrNo: OnlineDiagnostics.GetYesOrNoProc,  
    getFloppyChoice: OnlineDiagnostics.GetFloppyChoiceProc];
```

FloppyFormatDiskette formats a diskette using the IBM format. See arguments described above.

```
OnlineDiagnostics.FloppyCommandFileTest:PROCEDURE [  
    density: OnlineDiagnostics.SingleDouble,  
    sides: OnlineDiagnostics.SingleDouble,  
    sectorsPerTrack: CARDINAL [8..26],  
    sectorLength: OnlineDiagnostics.SectorLength,
```

```
errorHandling: OnlineDiagnostics.ErrorHandling,
cmdFile: LONG STRING,
displayFields: OnlineDiagnostics.DisplayFieldsProc,
displayTable: OnlineDiagnostics.DisplayTableProc,
displayNumberedTable: OnlineDiagnostics.DisplayNumberedTableProc,
putMessage: OnlineDiagnostics.PutMessageProc,
getConfirmation: OnlineDiagnostics.GetConfirmationProc,
getYesOrNo: OnlineDiagnostics.GetYesOrNoProc,
getFloppyChoice: OnlineDiagnostics.GetFloppyChoiceProc];
```

FloppyCommandFileTest executes an operator-generated floppy command file. **sectorsPerTrack** indicates the number of sectors per track that are to be used. **cmdFile** are the Floppy commands that are to be executed. For the remaining arguments, see the descriptions above.

```
OnlineDiagnostics.FloppyDisplayErrorLog:PROCEDURE [
displayFields: OnlineDiagnostics.DisplayFieldsProc,
displayTable: OnlineDiagnostics.DisplayTableProc,
displayNumberedTable: OnlineDiagnostics.DisplayNumberedTableProc,
putMessage: OnlineDiagnostics.PutMessageProc,
getConfirmation: OnlineDiagnostics.GetConfirmationProc,
getYesOrNo: OnlineDiagnostics.GetYesOrNoProc,
getFloppyChoice: OnlineDiagnostics.GetFloppyChoiceProc];
```

FloppyDisplayErrorLog displays a summary/error log of the prior executed tests. See arguments described above.

Performance Criteria

This appendix contains quantitative information about the observed performance of Pilot and information about how client programs are expected to behave. Where machine dependencies are a factor, it is assumed that the machine is a Dandelion. Some effort has been expended in describing the source of and confidence in the figures presented. These figures are presented to convey the flavor of the system rather than as hard performance guarantees. In general, crisp and quantitative performance requirements for Pilot are not available for comparison with the figures presented here.

A.1 Physical memory requirements of Pilot

The resident part of Pilot, the part that is ineligible for swapping, is 113 pages (28,928 words). It is allocated as follows: code - 51, data - 36, the Mesa runtime data structures - 19, and global frames - 7. As far as memory usage is concerned, this is the only machine dependent part of Pilot.

Most Pilot functions will require additional code and data to be swapped in. The memory requirements for Pilot functions are given in terms of working sets. A working set for a function is defined to be those virtual pages (code and data) which, if they are all in memory, provide a local minimum of page faults to service the function.

Because there is a significant overlap of code and data between one Pilot function and another, it is not possible to simply add up the sizes of all the working sets one anticipates using to get the total amount of memory required for a task.

Working set sizes are given in pages. They do not include the resident.

Pilot Function	Working Set Size	Notes
Communication		
Idle	15	
First Connection	13	Does not include Idle
Subsequent Connections	2 - 17	Does not include first connection

Pilot Function	Working Set Size
File	
Create	26
Delete	25
SetSize	28
Floppy Channel	8
Heap	
MakeNode	4
FreeNode	4
Signals	7
Space	
Allocate	9
Deallocate	14
MapAt	21
UnmapAt	6
Streams	1

A.2 Execution speed and client program profile

This section enumerates some typical characteristics which Pilot expects or will support in its clients. These estimates are intended to assist the client programmer in designing his use of Pilot facilities. They provide guidelines about which facilities are expensive and thus to be used sparingly and which facilities are inexpensive and can be exercised heavily. *None of these estimates are binding on either Pilot or client programs. Pilot 11.0 may deviate from these figures.*

These estimates apply to the cumulative load imposed by all clients operating on a single system element. A particular client program or system which does not exercise any of the resources very heavily may share the system element with other client programs, provided that the sum of their requirements remains within the estimates set out below.

A.2.1 Memory management

The following figures indicate the dynamic cost of virtual memory in terms of disk accesses, CPU time, and real time for a particular disk unit.

Facility	Minimum	Typical	Maximum
disk accesses to create or delete a space	0	2	>4
number of disk accesses to handle a page fault	0	1	>2
cpu time to handle a page fault	4-5 msec	6-8 msec	-
real time to handle a page fault ^{1,2}	5-7 msec	45-55 msec	>0.1 sec

¹Paging from the local Shugart 4008 disk. Real time per disk access = 1 · 200 milliseconds.

²No guarantee as to the maximum time to service a page fault will ever be given. In the case that the disk is occupied with real time processing, page fault handling times of several seconds or more can occur. The maximum time stated is the max time exclusive of such situations.

A.2.2 File management

The following figures indicate the typical characteristics of the Pilot file system. In this table, the term "active file" means a file which has been referenced recently so that its location and description are still present in the Pilot's caches.

Facility	Typical	Maximum
total drives (i.e., active physical volumes)	1	16
total existing files per volume	-	1/disk page
rate of file creation and deletion (long term average)	4	-
size of files (in pages)	8	$8 * 10^6$
number of volume pages allocated as a unit	-	$8 * 10^6$
number of file pages accessed in a sequence ¹	-	$8 * 10^6$

¹Limited by the amount of real memory for the access sequence.

A.2.3 Communication via the Ethernet

The following figures indicate the expected performance of communication between system elements connected to the same Ethernet.

Facility	Maximum
memory-to-memory transfer through the Stream interface	$7.5 * 10^5$ bits/sec

A.2.4 Processes

The following table provides data about the expected processing time on the Dandelion of each of the process structuring facilities.

Facility	Minimum	Typical	Maximum
Monitor entry or exit	3 μ sec.	< 4 μ sec.	5 μ sec.
Process switch time	25 μ sec.	30 μ sec.	40 μ sec.
Fork or Join ²	0.7 msec.	1 msec.	1.5 msec.

A**Performance Criteria**

Facility	Minimum	Typical	Maximum
Wait ^{1,2}	10 μ sec.	<60 μ sec.	100 μ sec.
Notify ¹	10 μ sec.	15 μ sec.	20 μ sec.

¹Exclusive of process switching time.

²The wide range on this facility reflects a current lack of data about its operating time rather than a dynamic variation in the final product.

Managing and Assigning File Types

In Pilot, every file must be assigned a type code at the time it is created. This code is of type **FileType** and is constant for the life of the file. It provides a means for Pilot, various scavenging programs, and clients to recognize the purpose for which each file was intended. This is especially important because files on Pilot disks do not inherently have meaningful strings for names, making it difficult for a human user or programmer to recognize which file is which. To make this principle work effectively, each different kind of file should be assigned its own unique type. This appendix describes how the type codes are assigned.

The center of this scheme is the **FileTypes** interface, maintained by the Pilot group. In this file are defined all subranges of **FileType** assigned to individual client and application groups. This module is designed so that it can be recompiled whenever a new type is assigned without invalidating any old version. Thus, within certain limits, a program may include any version of **FileTypes** which contains the type codes of interest to it without building in an unnecessary or awkward compilation dependency.

The basic structure of **FileTypes** is a set of subrange and constant definitions of the following form:

PilotFileType: TYPE = CARDINAL [0..256];

MesaFileType: TYPE = CARDINAL [256..512];

DCSFileType: TYPE = CARDINAL [512..768];

. . . -- *Subranges assigned to other clients and subsystems*

The subranges are designed to allow individual client organizations to administer their own file type assignment for their own purposes. Each group should maintain a module of the same form as **FileTypes** and include **FileTypes** in its **DIRECTORY** clause. Such a module would be used to assign types within the subrange allocated to that group while still providing a measure of protection against conflicting assignment by independent groups. The structure of this module should be similar to that of **FileTypes** in order that the assignment of a new type code does not trigger a universal recompilation of the subsystem.

For example, the Mesa Development Environment group is assigned the subrange of file types [9280..9344) to allocate as they see fit. This allocation is managed by the module **MesaDEFileType**, of the following form:

```
DIRECTORY

File: USING [Type],
FileTypes: USING [MesaDEFileType];
MesaDEFileType: DEFINITIONS =
BEGIN
MesaDEFileType: TYPE = FileTypes.MesaDEFileType;
-- MesaDE File Types
tUnassigned: File.Type = [MesaDEFileType-FIRST[MesaDEFileType]];
tRootDirectory: File.Type = [tUnassigned + 1];
tDirectory: File.Type = [tRootDirectory + 1];
. . . -- Other file types for use by Mesa
END.
```

This module can be recompiled independently of the module **FileTypes**, for example each time a new type code is added by the Mesa Development Environment group. All of mesa environment would derive the type codes for its files from this module.

In a similar manner, types within the subrange **PilotFileType**, for file types used by Pilot itself, are found in a private Pilot definitions module.

It is possible for two different program modules or configurations which include two different versions of **FileTypes.bcd** (or any of its derivatives, such as **MesaDEFileType.bcd**) to be bound together without error or conflict. This situation can arise, for example, because one configuration was compiled prior to the assignment of a new file type while the other was compiled afterwards. A problem occurs, however, if a module includes (either directly or indirectly) two different files defining file types. In this case, the compiler will refuse to compile the module unless the same version is used in both cases. For example, if a program includes both **FileTypes** and **MesaDEFileType**, and if **FileTypes.mesa** was updated after **MesaDEFileType.bcd** was created, then the Mesa compiler would generate an error message about **FileTypes** being used in differing versions. This error would also be generated if the program included **FileTypes** indirectly, say, by including another definitions module which itself had included a different version of **FileTypes**.

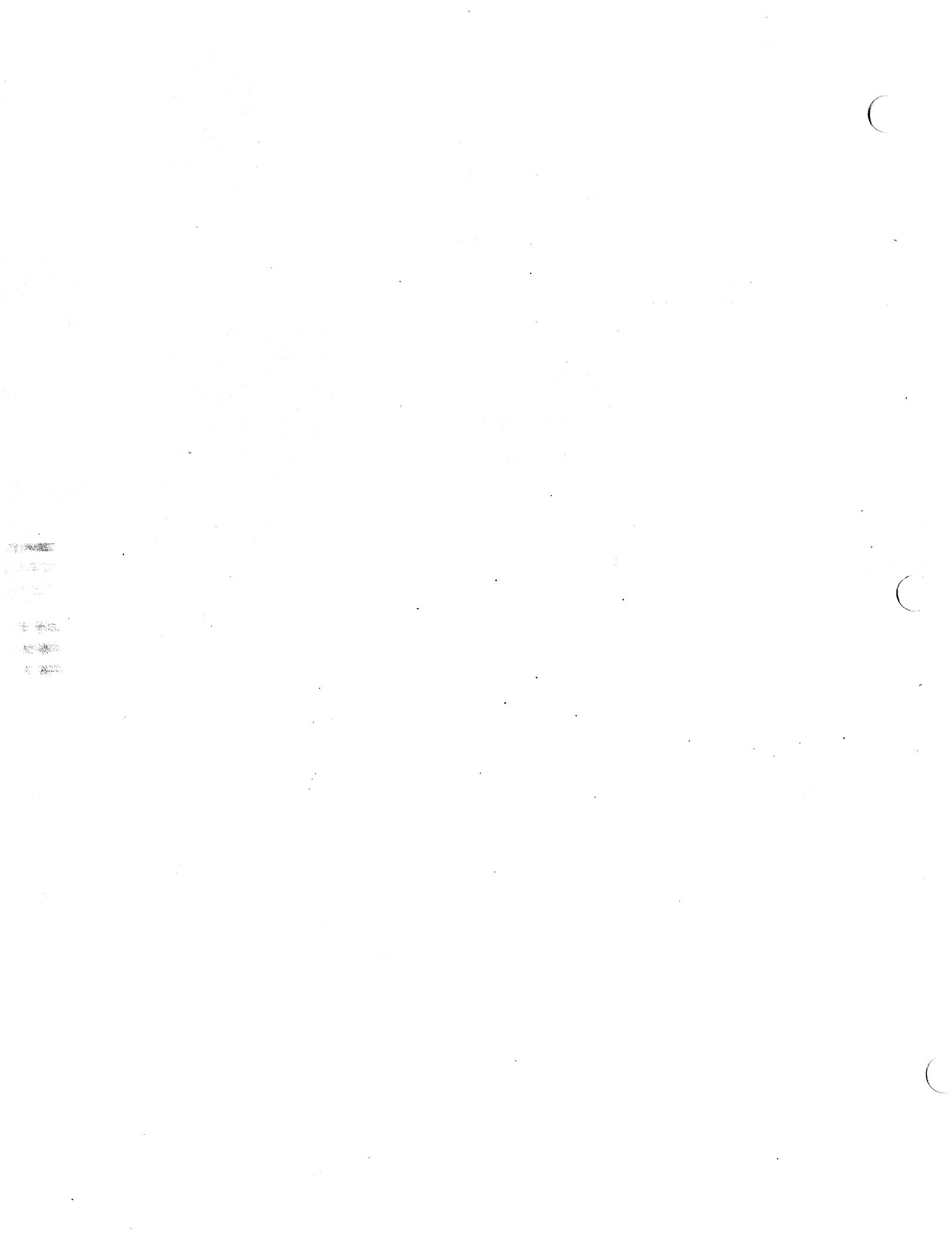
This problem should not, however, occur in a well-structured system design. For example, a file of type **tWidget** is perceived as such only by the module or modules which actually implement *widget* objects. All other modules use only a well-defined interface and deal in widgets, not widget implementations; i.e., the underlying file and its type are hidden. Since a single module will not be involved in the implementation of abstractions from two widely separated parts of the NS world, it need not see two different modules both defining separate ranges of type codes for files.

Therefore, the following style rules are recommended:

- a. **FileTypes.bcd** and its derivatives should be included only in program modules, not in definitions modules.

- b. Only one module defining the type codes for files should be included in any program (e.g., do not include both **FileTypes** and **MesaDEFileTypes**).
- c. The Pilot group will keep **FileTypes.mesa** and **FileTypes.bcd** up-to-date in conspicuous places, on the release directory between releases of Pilot.
- d. All programs, including Pilot, Common Software, and applications, should use type codes only symbolically from modules in which they are assigned. No program should fabricate a value of type **File.Type** from a numeric constant.

If all clients of Pilot observe these rules and the style of using Mesa definitions modules of the form of **FileTypes**, the job of administering the assignment of type codes for Pilot files can be kept manageable. In return, the Pilot group can react immediately to requests for a new type code or subrange of type codes. If this style is not observed, the administration of global constants such as these will become a complicated, time-consuming task with a corresponding difficulty in reacting quickly to requests.



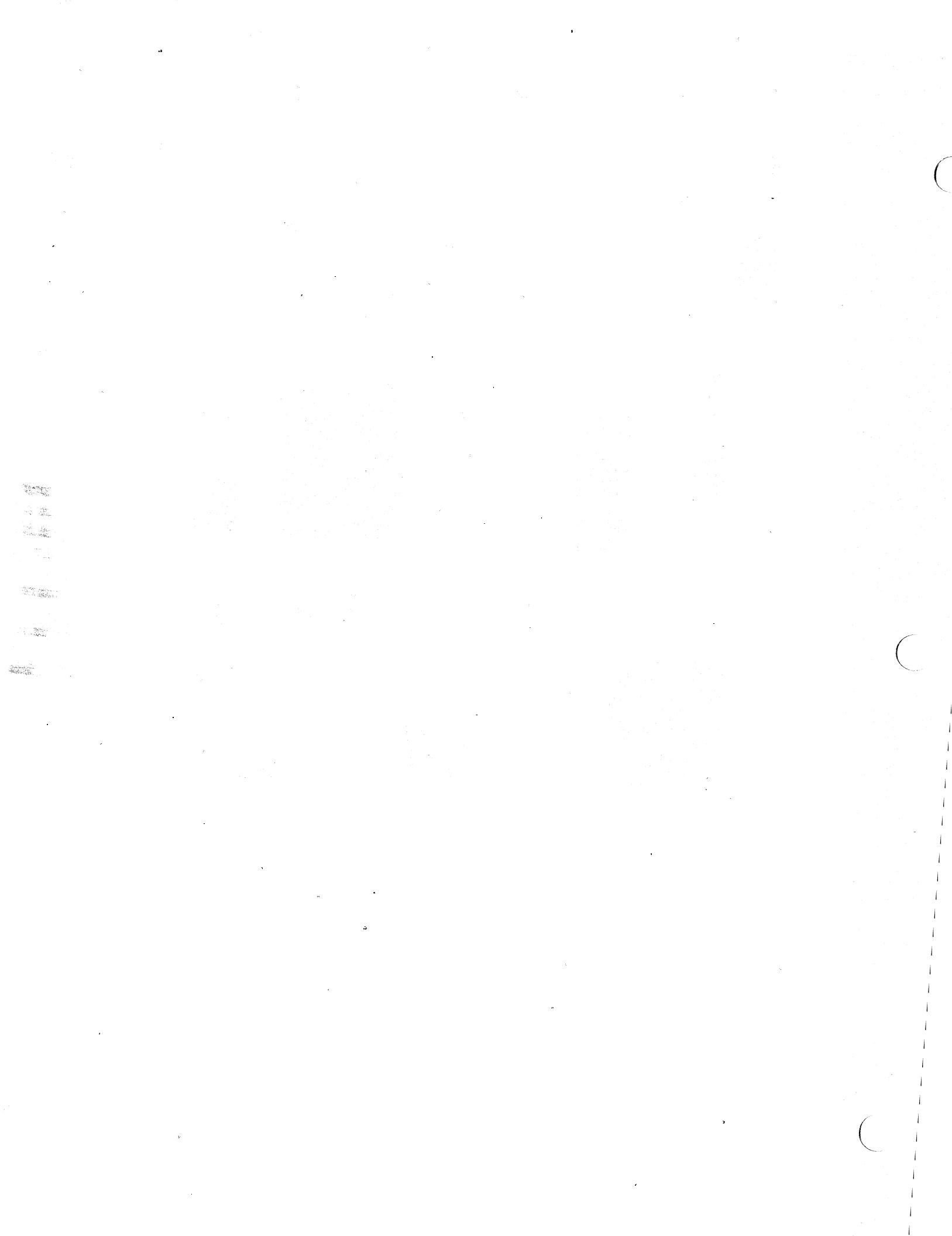


Pilot's Interrupt Key Watcher

This appendix describes the operation of the interrupt key watcher that can be enabled by users or clients at boot time, via boot switch 8.

If one goes to the debugger and then does an interpret call, the interpret call is executed in the process that went to the debugger, and consequently runs at that process's priority. If this is a priority at which the taking of faults is restricted, the interpret call may fault and block trying to allocate state vectors.

If Pilot is booted with the 8 boot switch, pressing **LOCK-LeftSHIFT-RightSHIFT-STOP** will cause Pilot to call the debugger with the message "Pilot Emergency Interrupt". This is done at a priority level that precludes doing any interpret calls from the debugger.





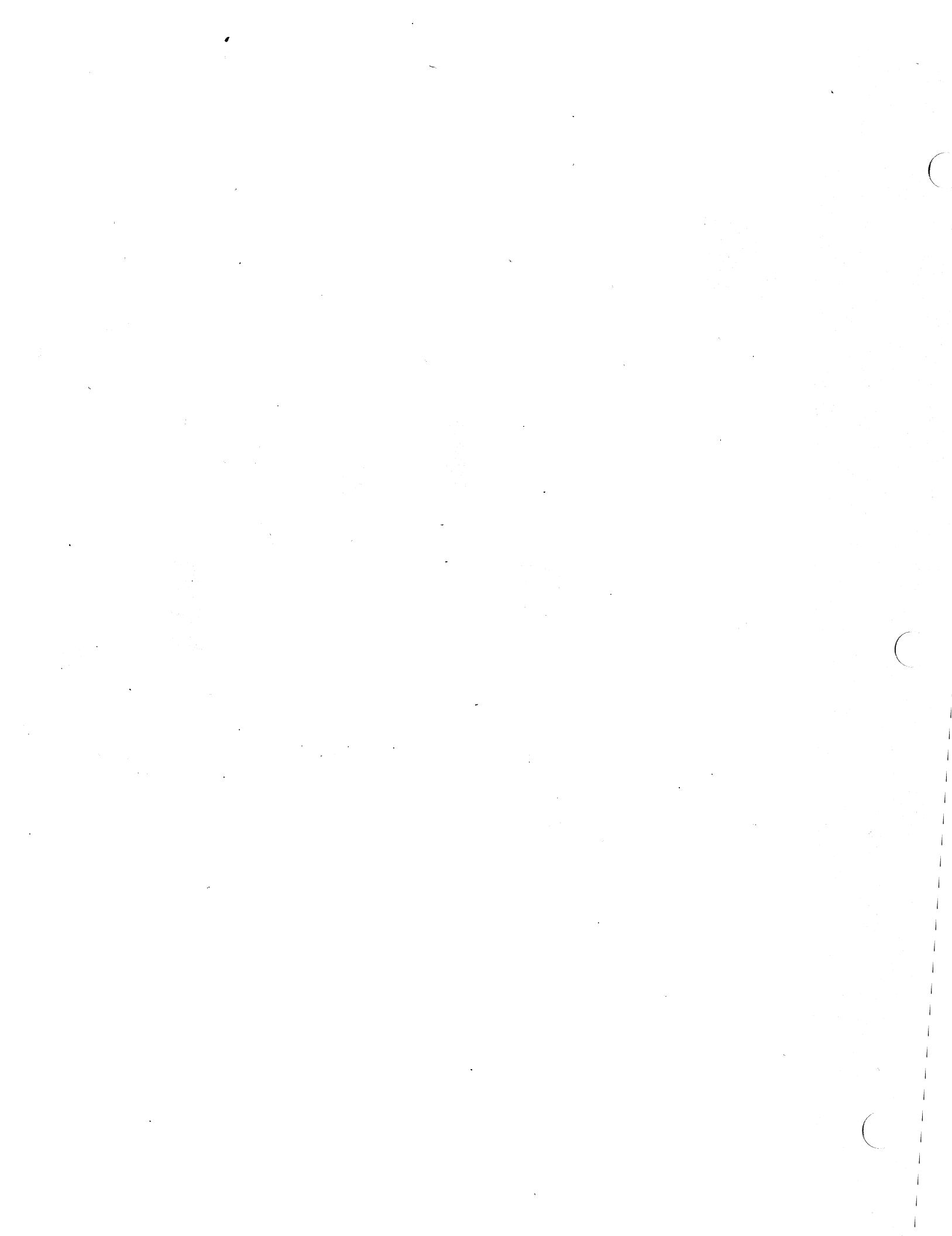
UtilityPilot

Systems that are based on **PilotKernel.bcd** require that a disk be present on the machine. The boot file containing the system must be installed on the disk, from which it is loaded into the processor memory when the system is booted. The disk contains the system physical and logical volumes for the system (i.e., those on which the boot file is located).

Systems that are based on **UtilityPilotKernel.bcd** do not require that a disk be present on the machine. The boot file containing the system may be loaded from any source, e.g., ethernet, floppy disk. UtilityPilot provides the same facilities as regular Pilot, with the following exceptions:

- There are no system physical and logical volumes.
- No volumes are brought online as part of Pilot initialization.
- The entire system and its working data must fit into the real memory of the processor. (Backing storage provided by Space.ScratchMap and the system heaps come from real memory)
- Clients must validate/set local time parameters before calling any pilot facility that needs them.
- Map logging is disabled.
- Run-time loading is not supported.

UtilityPilot is commonly used to build special utility systems, such as, disk initializers and diagnostics.



Multi-national Considerations

The hardware and software described in this manual support serial communication via the RS-232-C controller in accordance with EIA standard RS-232-C. No support is provided for CCITT Recommendations V.24 and V.27, the equivalent prevailing standard in most of Europe.

(

(

(

References

F.1 Mandatory references

The following documents should be studied before or in conjunction with this document:

- *Courier: The Remote Procedure Call Protocol*, XSIS 038112
- *Mesa Language Manual*--610E00170
- *XDE User's Guide*--610E00140
- *Mesa Programmer's Manual*--610E00150

In addition, the release documentation accompanying each release of Pilot should be consulted before writing programs that use Pilot.

F.2 Informational references

The following documents provide useful additional information:

- *The Ethernet, A Local Area Network, Data Link Layer, and Physical Layer Specifications, Version 1.0.* [September 30, 1980]
- *Xerox Internet Transport Protocols.* [February 1982]



Index

a1, 4-48
a16, 4-48
a2, 4-48
a4, 4-48
a8, 4-48
Abort, 2-18, 2-21, 5-16
abort
 canceling, 2-22
 key, 5-33
AbortCall, 6-44
ABORTED, 2-22
aborted, 5-29, 5-30, 10-9
abortedByDelete, 5-29, 5-30
AbortPending, 2-22
Access, 4-31
 access permissions, 4-31
Activate, 4-37, 4-38, 4-39
ActivateProc, 4-38
AddDependency, 2-32
 address fault, 2-28, 2-36, 4-30, 4-35,
 4-45, 5-23, 9-1
addressfault, 9-3
AddSegment, 4-47
AdjustGreenwichMeanTime, 2-13
 agent procedure, 2-30, 2-31
AgentProcedure, 2-31
 alarm clock, 2-15
Alignment, 4-48
 alignment, 4-48, 5-3
 alignment
 byte, 5-3
 page, 5-3, 5-23
 word, 2-35, 4-48, 5-3
alive, 4-31
Allocate, 2-35, 4-40, 4-41
 allocation
 of objects, 2-34
AllocationPool, 2-34, 2-35
AllocFree, 2-34
AllocPoolDesc, 2-35
alreadyAllocated, 4-40, 4-41
alreadyAsserted, 4-3, 4-5, 4-6, 4-7,
 8-7
alreadyDeallocated, 4-41
AlreadyFormatted, 5-25
AlreadyFreed, 2-35, 2-36
Alto, 7-10
Alto
 ADL keyboard, 5-6
 Microswitch keyboard, 5-6
 time standard, 2-13
american, 10-14
ANSI, 7-10
anyEthernet, 2-4
anyPilotDisk, 2-4, 2-5
Append, 7-11
AppendChar, 7-6
AppendCharAndGrow, 7-9
AppendCurrent, 7-11
AppendDecimal, 7-8
AppendExtensionIfNeeded, 7-9
AppendLongDecimal, 7-8
AppendLongNumber, 7-8
AppendNumber, 7-8
AppendOctal, 7-8
AppendString, 7-6
AppendStringAndGrow, 7-9
AppendSubString, 7-6
Applications, 1-2
ApproveConnection, 6-14, 6-17
Arguments, 6-48, 6-51
 arguments, 6-50, 6-51, 6-52, 6-53
ARRAY, 6-46
Ascii
 DEFINITIONS, 7-1
asciiByteSync, 6-28
AssertNotAPilotVolume, 4-6

AssertPilotVolume, 4-5, 4-6
AssignAddress, 6-24
AssignDestinationRelativeAddress
 6-25
AssignLocalAddress, 2-11
AssignNetworkAddress, 6-11,
 6-12, 6-18
asynchFramingError, 5-29, 5-30
asynchronous, 6-31, 6-32
 asynchronous operation
 definition of, 1-8
 atomic
 restoring, 8-16
 saving, 8-16
Attention, 3-5, 3-8, 6-20
attention, 3-4
 attention, 6-20, 6-21
 attention flag, 3-2, 3-8
Attributes, 4-53, 5-17
AutoRecognitionOutcome, 6-28
AutoRecognitionWait, 6-28
AwaitStateChange, 4-3
Background, 5-13, 10-14
backing file, 1-5, 2-17, 5-33
backing storage, 4-30, 4-32
backing stream, 5-34
BackingStream, 5-34
Backstop
 DEFINITIONS, 9-1
backstop, 1-10, 2-28, 9-1
 control, 9-1
 core, 9-1
 implementing, 9-1
 initializing log file, 9-2
 log file, 9-1, 9-2
 logging errors, 9-2
 reading log file, 9-1, 9-4
BackstopImpl.bcd, 9-1
BackstopNub, 9-4
 DEFINITIONS, 9-1
BackstopNubImpl.bcd, 9-1
bad pages, 4-9, 4-10, 4-27, 8-5, 8-6
bad sector, 5-23
badCode, 2-24, 2-25
badDisk, 4-3, 4-6, 4-7, 5-25
BadPage, 8-5, 8-7
badPageList, 4-7, 4-8
badSectors, 5-27
badSpotTableFull, 4-3, 4-10
BadSwitches, 8-11
Base, 2-3, 4-44
BASE POINTER, 4-44
basic machine, 1-2
 facilities, 1-3
BBTable, 5-13, 5-14
Beep, 5-16, 10-15
BEL, 7-1
Billing and Accounting Functions,
1-3
binding, 6-46
BitAddress, 2-3
BITAND, 2-9
BitBlt, 2-3, 2-6, 5-13
 table, 5-13
BitmapIsDisconnected, 5-14
BITNOT, 2-9
BitOp, 2-8
BITOR, 2-9
BITROTATE, 2-9
BITSHIFT, 2-9
bitsPerByte, 2-1
bitsPerCharacter, 2-1
bitsPerWord, 2-1
bitSync, 6-28
bitSynchronous, 6-31
BITXOR, 2-9
black, 5-13, 10-14
BlackenScreen, 10-15, 10-16
Blank, 7-3
Blanks, 7-3
BlinkDisplay, 5-13, 5-34
Block, 2-2, 3-3, 3-4, 3-12, 3-13, 6-6
 6-9, 6-17, 6-20, 6-21, 6-65, 7-3
block, 4-58
blockPointer, 2-2
BlockSize, 4-44
boolean, 10-18
BooleanDefaultFalse, 4-14
boot button, 8-13, 8-15
boot file, 2-20, 2-26, 4-8, 4-14, 5-28,
 8-2, 8-4, 8-7, 8-8, 8-9, 8-13, 8-14
booting, 8-15
creation, 8-13
default, 8-8
installation, 8-7, 8-14, 8-15
leader, 8-13, 8-14
local, 8-3
making, 8-14
universal, 8-3
updating, 8-15
writing, 8-14
boot loader, 2-17, 8-8
boot switch, 2-16, C-1
 default, 8-11
 assignments, 2-16
bootable floppies, 5-27
BootButton, 8-13, 8-15
BootDevice, 2-16
bootFile, 4-7, 4-8
BootFileType, 4-24, 8-7
BootFromFile, 8-13, 8-15

- BootFromPhysicalVolume**, 8-13,
 8-15
BootFromVolume, 8-13, 8-15
booting
 Pilot's state after, 2-16
 preparation, 8-4
booting agent, 2-16
BootLocation, 8-16
bootServerSocket, 2-11
BoundsFault, 2-27
break, 5-32
breakDetected, 5-29, 5-30, 5-31
broadcastHostNumber, 2-10
BS, 5-36, 7-1
BSPMemCache.bcd, 9-1
bug, 9-3
bulk data transfer, 6-47, 6-50, 6-53,
 6-55, 6-58
Byte, 2-1, 3-2
byte alignment, 5-3
ByteBit
 DEFINITIONS, 2-6
bytesPerPage, 2-2
bytesPerWord, 2-1
byteSynchronous, 6-31
CADFileType, 4-19
Call, 6-49, 6-59
call, 9-3
CallDebugger, 2-28, 9-1
CancelAborts, 2-22
cancelSignal, 10-16, 10-17
cannotWriteLog, 4-23
cantFindStartListHeader, 8-9
CanInstallUCodeOnThisDevice,
 8-5 8-7
cantWriteBootFile, 8-9
cardinal, 10-18
catch phrase, 2-22, 3-5
Caution, 1-9
cCallCSC, 10-17
CCITT Recommendations, E-1
cCloseWn, 10-17
cdc9730, 2-5
CedarFileType, 4-19
cEnsureReady, 10-17
Century Data Systems, 2-5
cExit, 10-17
cFirst, 10-17
change count, 4-3, 4-4
ChangeLabelString, 4-16
ChangeName, 4-9
channel, 1-7, 6-1
ChannelAlreadyExists, 5-28
ChannelHandle, 5-28
ChannelInUse, 6-37
channelInUse, 10-9
ChannelQuiesced, 5-29
ChannelSuspended, 6-38
Char, 7-2
character, 10-18
character terminal, 5-28, 5-32
CharacterLength, 5-30, 5-31
CharLength, 6-29, 6-32, 6-34
CharsAvailable, 5-30, 5-34
charsPerPage, 2-2
charsPerWord, 2-1
CharStatus, 5-35
checkOnly, 4-7, 4-8, 4-23
CheckOwner, 4-51, 4-54
CheckOwnerMDS, 4-54
Checksum, 2-6
cnsDiffCleanDisk, 10-17
cnsInsertCleanDisk, 10-17
cnsInsertDiagDisk, 10-17
cnsInsertWriteable, 10-17
Class, 4-33
ClassOfService, 6-11, 6-47, 6-48
cLast, 10-17
ClearDisplay, 10-15, 10-16
clearinghouse, 2-11
clearingHouseSocket, 2-11
client, 6-10
client program profile, A-2
client programs, 1-1
clients, 2-33
ClientsImpls, 2-33
clock ticks
 conversion of, 2-19
Close, 4-15, 4-56, 4-58, 5-22, 6-18,
 6-19
close protocol, 6-18
closedAndConsistent, 4-14
closedAndInconsistent, 4-14
CloseReply, 6-19
closeReplySST, 6-18, 6-19
closeSST, 6-18, 6-19
CloseStatus, 6-18, 6-19
cmcll, 6-28
cNBNotReady, 10-17
code links, 2-24, 2-26
CommError, 10-1
Common Software, 1-2, 5-28, 5-32,
 7-1, 7-2, 7-5, 7-10, B-3
CommonSoftwareEventIndex, 2-31
CommonSoftwareFileType,
 4-19, 4-20
CommonSoftwareFileTypes
 DEFINITIONS, 4-17
CommParamHandle, 6-29, 6-32
CommParamObject, 6-29, 6-32
communication
 errors, 6-14

initialization, 8-12
link, 1-7
performance, A-3
system, 1-6
Communication package, 2-17, 8-1
Communication.bcd, 6-4, 6-9, 6-23
communicationError, 10-1
Compact, 5-26, 5-27
Compare, 7-7
Compiler option, 2-27
CompletionCode, 3-4
CompletionHandle, 6-30, 6-32
complex services, 1-4
condition variable, 1-4, 1-10, 2-19, 2-22
 timeout, 2-20
ConfigError, 2-24, 2-25
ConfigErrorType, 2-24
configuration, 2-24, 2-26
connection, 6-10
ConnectionFailed, 6-13, 6-14, 6-15
ConnectionID, 6-11
connectionless protocol, 6-4
ConnectionSuspended, 6-14, 6-19
containsOpenVolumes, 4-3, 4-5, 4-6
Context, 5-17, 5-19
continueOnError, 10-19
continueToNextError, 10-18
Control, 7-1
control characters, 7-1
control codes, 3-2
Control Data Corporation, 2-5
control link
 null, 2-27
ControlFault, 2-27
ControlLink, 2-23
Coordinate, 5-12, 10-14
CoPilot, 8-2
COPY, 2-7
Copy, 7-7
CopyFromPilotFile, 5-23
CopyIn, 4-35, 4-36, 5-23, 8-14
CopyOut, 4-35, 4-36, 5-23, 8-14
CopyToString, 7-9
CopyToPilotFile, 5-23
Correspondent, 6-30, 6-32, 6-34
cOtherDiskErr, 10-17
CountType, 10-10
Courier data types, 6-59
courierSocket, 2-11
CR, 7-1, 7-3
Create, 3-10, 4-12, 4-21, 4-44, 4-45, 4-50, 4-52, 4-53, 5-28, 5-32, 5-33, 6-11, 6-13, 6-15, 6-38, 6-47, 6-61, 7-12, 8-13
CreateBackstopLog, 9-2
CreateFile, 5-26, 5-27
CreateFloppyFromImage, 5-24
CreateListener, 6-11, 6-13, 6-17
CreateMDS, 4-50, 4-52
CreatePhysicalVolume, 4-5, 4-6, 8-5
CreateReplier, 6-7, 6-9
CreateRequestor, 6-7
CreateScrollWindow, 5-14
CreateSubsystem, 2-32
CreateTransducer, 6-11, 6-12, 6-13, 6-14, 6-15, 6-18
CreateUniform, 4-50
RemoveCleanDisk, 10-17
RemoveDiskette, 10-17
Current, 7-11
current date, 7-10
current time, 7-10
currentLogVersion, 4-24
cursor, 5-14
CursorArray, 5-14, 10-14
CyclicSubsystem, 2-32
Dakuon, 5-9
damaged, 4-7, 4-8
DamageStatus, 4-7
Dandelion, 2-17, 5-16, A-1, A-3
dangling reference, 1-9, 2-18, 2-24, 2-24, 2-25, 3-3
data blocks, 8-5
data space, 2-17
data window, 4-34, 4-35
DataError, 5-22, 5-23, 5-25
dataLost, 5-29, 5-30
dataTerminalReady, 6-34
Date, 7-4, 7-5
date, 2-12
DateFormat, 5-36, 7-4
dateOnly, 5-37, 7-4
dateTime, 5-37, 7-4
Daylight Saving Time, 2-13
DBITAND, 2-9
DBITNOT, 2-9
DBitOp, 2-9
DBITOR, 2-9
DBITSHIFT, 2-9
DBITXOR, 2-9
DCSFileType, 4-19, B-1
Deactivate, 4-37, 4-38, 4-39
DeactivateProc, 4-38
dead, 4-31
Deallocate, 4-41
debuggeDebugger, 4-14
debugger, 4-13, 4-14, 8-10
debugger, 2-17, 2-28, 8-4, 8-9, 9-1, C-1
 debugger, 8-4
remote, 2-17

debuggerDebugger, 4-13
debuggerVolumeID, 4-15
Decimal, 7-4
DecimalFormat, 7-3
DecodeSwitches, 8-11
default, 5-25
default stream, 6-58
default volume, 4-11
defaultBase, 4-40, 4-41
defaultInputOptions, 3-4, 6-20, 6-58
defaultObject, 3-18
defaultPageCount, 5-23
defaultRetransmissionInterval, 6-5
defaultSwapUnitOption, 4-33
defaultSwitches, 2-16, 8-12
defaultTime, 7-11
defaultWaitTime, 6-5, 6-10
DEL, 7-1
DEL, 5-33
Delete, 3-3, 3-10, 4-21, 4-51, 5-29,
 5-30, 5-32, 6-7, 6-40, 6-47
delete, 3-17
DeleteFile, 5-26
DeleteListener, 6-13, 6-14, 6-17
DeleteLog, 4-24, 4-26
DeleteMDS, 4-51
DeleteOrphanPage, 4-28
DeleteProcedure, 3-17
DeleteScrollWindow, 5-15
DeleteSubString, 7-7
DeleteSubsystem, 2-32
DeleteTempFiles, 8-10
Density, 5-25
dependency relationship, 2-32
DependsOn, 2-31, 2-33
Description, 6-49, 6-59
description, 6-53
description routine, 6-59, 6-64, 6-66
DescriptorForArray, 6-65
deserialization, 6-61
DeserializeParameters, 6-66, 7-12
Destroy, 5-33, 7-12
Detach, 2-18, 2-21, 2-29
Detail, 10-6
Development Common Software, 1-9
development tools, 8-2
Device
 DEFINITIONS, 2-4
device driver, 1-7, 5-1
device faces, 8-1
device interfaces
 model of, 1-10
device numbers, 2-4
device types, 2-4
deviceNotReady, 10-18
DeviceStatus, 5-31, 6-32
DeviceTypes
 DEFINITIONS, 2-4
Diablo 630 character printer, 5-28
diagnostics, 10-1
DiagnosticsFileType, 4-19
diagnosticsServerSocket, 2-11
Dial, 6-43
Dialer testing, 10-12
DialMode, 6-30, 6-33
DialupOutcome, 10-13
DialupTest, 10-13
DifferentType, 4-58, 4-59
directoryFull, 4-16
Disable, 4-57
disable, 4-57
DisableAborts, 2-22
DisableTimeout, 2-20
disconnected, 5-12
disjoint data, 6-46
disjoint data, 6-52, 6-53, 6-65
disjoint data types, 6-64
DisjointData, 6-61, 6-65
disk diagnostic, 8-4
disk drive, 4-3, 4-4
 change state, 4-3
 direct access, 4-4
 inactive state, 4-5, 4-6
 non-Pilot access, 4-4, 4-5
 Pilot access, 4-4
 read-only, 4-5
 ready, 4-3, 4-4
 state, 4-4
disk formatting, 8-4
DiskAddress, 5-19
diskette, 5-17
 bad pages, 5-27
 compaction, 5-25
 free pages, 5-25, 5-27
 IBM format, 5-17, 5-20
 label, 5-25
 malformed, 5-27
 Troy format, 5-17
 write enable sticker, 5-23
 Xerox 850 format, 5-17
diskette hardware error
 read or write, 5-23
diskHardwareError, 4-27, 4-28
diskNotReady, 4-27, 4-28
DiskPageNumber, 8-5, 8-6
diskReadError, 4-3, 4-6
Dispatch, 6-48
dispatch, 6-48
Dispatcher, 6-52, 6-57
dispatcher, 6-52
display, 5-12
 blink, 5-13

border, 5-13
cursor, 5-12, 5-14
cursor coordinates, 5-14
cursor pattern, 5-14
image, 5-12
DisplayFieldsProc, 10-19
DisplayNumberedTableProc, 10-19
displayStuff, 10-18
DisplayTableProc, 10-19
DivideCheck, 2-8, 2-28
DIVMOD, 2-8
DocProcFileType, 4-19
double, 10-18, 5-17, 5-25
down, 2-16
Drive, 5-20
Duplicity, 6-29, 6-33
duplicate, 4-25
duplicate page, 4-25
duplicate suppression, 6-4, 6-9
duplicateRootFile, 4-16
east, 2-13
ebcdicByteSync, 6-28
echo testing, 10-2
EchoClass, 5-34
echoerSocket, 2-11
EchoEvent, 10-3
echoing, 6-35
EchoParams, 10-3, 10-4
EchoResults, 10-4
echoUserNotThere, 10-5
EIA Standard RS-232-C, E-1
EIDisk, 8-4
electronicMailFirstSocket, 2-11
electronicMailLastSocket, 2-11
Empty, 7-7
empty, 4-9
emptyFile, 8-7
EnableAborts, 2-22
end of time, 2-12
end-of-stream
 implementation, 3-5
endEnumeration, 6-23
endOfFile, 5-22, 5-23
EndOfStream, 3-5, 3-7, 6-19
endOfStream, 3-4
EndRecord, 3-6, 3-12, 3-14
endRecord, 3-4, 3-6, 3-7, 3-12
EntryType, 4-25
EnumerateExports, 6-66
EnumerateRoutingTable, 6-25
EnumerationAborted, 2-34
Environment
 DEFINITIONS, 2-1
envoySocket, 2-11
Equal, 7-7
EqualSubString, 7-7
Equivalent, 7-7
EquivalentSubString, 7-7
Erase, 4-13
Error, 2-25, 2-35, 4-2, 4-3, 4-8, 4-9,
 4-10, 4-12, 4-20, 4-23, 4-31, 4-35,
 4-36, 4-41, 4-42, 4-50, 4-55, 5-14,
 5-17, 5-21, 6-6, 6-8, 6-9, 6-47,
 6-53, 8-5, 8-7, 8-9, 8-10
error, 4-56
error
 protocol, 6-2
 uncaught, 9-1
error-free, 6-9
Error[alreadyAsserted], 8-5
ErrorCode, 6-47, 6-48, 6-49, 6-50,
 6-51, 6-53, 6-59
ErrorEntry, 9-4, 9-5
ErrorHandling, 10-19
ErrorReason, 6-6
errorSocket, 2-11
ErrorType, 2-35, 4-3, 4-4, 4-5, 4-6,
 4-12, 4-13, 4-20, 4-23, 4-27, 4-28,
 4-31, 4-33, 4-38, 4-40, 4-41, 4-50,
 4-51, 4-54, 4-55, 4-56, 4-57, 5-14,
 5-15, 5-17, 5-20, 5-21, 5-22, 5-23,
 5-24, 5-25, 5-26, 5-27, 5-28, 9-3
ESC, 7-1
ESC, 5-37
etherBooteeFirstSocket, 2-11
etherBooteeLastSocket, 2-11
etherBootGermSocket, 2-11
EtherDiagError, 10-5
EtherErrorReason, 10-5
Ethernet, 2-4
ethernet, 2-4
ethernet, 1-7, 6-1, 2-17, 2-29, 8-2,
 8-11, 8-14
 performance, A-3
 statistics, 10-6
Ethernet 1, 2-17
ethernetOne, 2-4
EtherStatsInfo, 10-6
european, 10-14
even, 5-31
Event, 2-31
eventData, 2-31, 2-33
EventIndex, 2-31
EventReporter, 10-5
ExchangeClientType, 6-4, 6-9
ExchangeHandle, 6-5, 6-9
ExchangeID, 6-5
exit, 10-18
Expand, 4-53
ExpandAllocation, 2-35
ExpandMDS, 4-53
ExpandString, 7-10

expiration date, 8-11
exportedTypeClash, 2-24, 2-25
ExportItem, 6-66
ExportRemoteProgram, 6-48, 6-52, 6-56, 6-61, 6-66
Exports, 6-66
face, 1-1, 8-1
failure, 6-28
FailureReason, 6-15
FailureType, 8-7
fetch, 6-61
fetch, 6-66
FF, 7-1
Field, 10-18
FieldDataType, 10-18
File, 4-17
file, 1-5, 4-1, 4-17
 absence of pages at end, 4-25
 access, 1-5
 addressing, 4-18
 attributes, 4-21
 create, 4-21
 creation performance, 4-21
 delete, 4-21
 extension, 4-22
 id, 1-6
 identifier, 4-17
 list, 5-21, 5-25, 5-26
 location of, 1-2
 management, 1-10
 performance, A-3
 manager, 2-17
 maximum size, 4-18
 name, 4-17
 permanent read-only, 4-21
 temporary, 4-22, 4-26
 type, 5-28
 type code, 4-18
 allocation, 4-18
 windows, 4-21, 4-34
File
 DEFINITIONS, 4-17
FileCount, 5-21
FileEntry, 4-24, 4-25
FileHandle, 5-22
FileID, 5-21, 5-28
fileListFull, 5-26
fileListLengthTooShort, 5-24
FileLocation, 8-16
fileNotFound, 5-22, 5-23, 5-26
FileServiceFileType, 4-19
FileTypes, B-1, B-2
FileTypes, 4-19
 DEFINITIONS, 4-17
 FileTypes.bcd, B-2, B-3
 FileTypes.mesa, B-3
FillRoutingTable, 6-25
FillScreenWithObject, 10-16
filter, 1-7, 3-1, 3-2, 3-5, 3-9, 3-11, 3-13, 3-14, 3-18
FindAddresses, 6-18
FindDestinationRelativeNetID, 6-26
FindMyHostID, 6-26
FinishWithNonPilotVolume, 4-6
first64K, 2-3
firstPageBad, 8-7
firstPageCount, 2-2, 4-2, 4-11, 4-18, 4-29
firstPageNumber, 2-2, 4-2, 4-11, 4-18, 4-29
firstPageOffset, 2-3, 4-29
FirstSA1000PageForPilot, 8-6
Firstr300PageForPilot, 8-6
Firstr80PageForPilot, 8-6
five12, 10-18
flakeyPageFound, 8-7
Floppy
 DEFINITIONS, 5-21
floppy
 enumeration of bad sectors, 5-27
 enumeration of files, 5-26
 Pilot supported standard, 5-22
 snapshotting and replication, 5-24
floppy disk, 5-16, 8-2
 drive characteristics, 5-17
 multiple sector transfers, 5-19
Floppy file system, 4-20, 5-21
FloppyChannel
 DEFINITIONS, 5-16
FloppyCleanReadWriteHeads, 10-20, 10-21
FloppyCommandFileTest, 10-21
FloppyDisplayErrorLog, 10-21
FloppyExerciser, 10-20
floppyFailure, 10-18
FloppyFormatDiskette, 10-21
floppyImageInvalid, 5-24
FloppyImpl.bcd, 5-21
FloppyMessage, 10-17
FloppyReturn, 10-18
floppySpaceTooSmall, 5-24
FloppyStandardTest, 10-20
FloppyWhatToDoNext, 10-18
flow-controlled, 6-9
FlowControl, 6-30, 6-33
flowControl, 6-35
Flush, 4-53
FlushMDS, 4-53
ForceOut, 4-22, 4-37, 4-38, 4-39
FORK, 2-18, 2-20
Format, 5-25, 8-5
Format

DEFINITIONS, 7-2
Format package, 7-2
FormatBootMicrocodeArea, 8-5
FormatImpl.bcd, 7-2
FormatPilotDisk, 8-5, 8-7
 DEFINITIONS, 8-3
FormatPilotDiskImpl.bcd, 8-5
formatted, 8-5
FormattingMustBeTrackAligned, 8-5
Frame, 9-3
frame links, 2-25, 2-26
frameTimeout, 6-35
FREE, 4-43, 4-49, 4-51, 4-52, 4-54
Free, 2-35, 6-52
free, 6-61
 free storage package, 4-43
FreeEnumeration, 6-66
FreeMDSNode, 4-54
FreeMDSString, 7-9
FreeNode, 4-49, 4-54
FreeString, 7-9
full, 5-37, 7-4
full-duplex, 5-31
garbage collection, 4-44
GenericProgram, 2-23
germ, 4-7, 4-8, 8-7, 8-8
germ, 8-1, 8-4, 8-7, 8-8, 8-13, 8-16
Get, 5-29, 6-40
GetAttributes, 4-9, 4-15, 4-22, 4-46, 4-53, 4-58, 4-59, 5-25
GetAttributesMDS, 4-53
GetBackground, 5-13
GetBcdTime, 2-26
GetBitBltTable, 5-13, 5-14
GetBlock, 3-4, 3-5, 3-6, 3-7, 3-10, 3-11, 3-13, 4-58, 4-59
GetBootFilePointer, 5-27
GetBootFiles, 5-27
GetBuildTime, 2-26
GetByte, 3-6, 3-7
GetByteProcedure, 3-15
GetCaller, 2-26
GetChar, 3-6, 3-7, 5-35
GetClockPulses, 2-14
GetConfirmationProc, 10-20
GetContainingPhysicalVolume, 4-9
GetContext, 5-17
GetCount, 4-57, 4-58
GetCurrent, 2-21
GetCurrentProcess, 9-3
GetCursorPattern, 5-14
GetDecimal, 5-38
GetDelayToNet, 6-26
GetDeviceAttributes, 5-17
GetDialerCount, 6-45
GetDriveSize, 8-11
GetEcho, 5-34
GetEchoCounters, 10-8
GetEchoResults, 10-2
GetEditedString, 5-33, 5-35, 5-36, 5-37
GetError, 9-3
GetEthernetStats, 10-7
GetExpirationDate, 8-11
GetExpirationDateSuccess, 8-11
GetFaultedProcess, 9-3
GetFileAttributes, 5-26
GetFileLocation, 8-16
GetFloppyChoiceProc, 10-20
GetGreenwich mean time, 7-11
GetGreenwichMeanTime, 2-12
GetHandle, 4-4, 5-20
GetHints, 4-4, 4-5
GetID, 5-36
GetImageAttributes, 5-24, 5-25
GetIndex, 4-57, 7-12
GetLabelString, 4-16
GetLine, 5-36
GetLocalTimeParameters, 2-14
GetLog, 4-26
GetLogEntry, 9-4, 9-5
GetLongDecimal, 5-38
GetLongNumber, 5-37
GetLongOctal, 5-38
GetLost, 4-57, 4-58, 9-2, 9-4
GetMapUnitAttributes, 4-42
GetMesaChar, 10-16, 10-17
GetMousePosition, 10-15
GetNetworkID, 6-26
GetNext, 4-9, 4-14, 4-58, 9-4, 9-5
GetNextAction, 10-16
GetNextBadPage, 4-10
GetNextBadSector, 5-27
GetNextDrive, 4-3, 5-20
GetNextFile, 5-26
GetNextFrame, 9-3
GetNextLine, 6-42
GetNextLogicalVolume, 4-8
GetNextProcess, 9-3
GetNextRootFile, 4-17
GetNextSubVolume, 8-10
GetNumber, 5-37
GetOctal, 5-38
GetPassword, 5-36
GetPhysicalVolumeBootFile, 8-9
GetPosition, 3-9
GetPositionProcedure, 3-17
GetPriority, 2-21
GetProcedure, 3-15
GetPVLocation, 8-17
GetRestart, 4-59

GetRootNode, 4-46
GetRouterFunction, 6-26
GetRS232CResults, 10-9
GetSegmentAttributes, 4-47
GetSize, 4-21, 9-4, 9-5
GetSSTProcedure, 3-16
GetState, 4-57, 5-13
GetStatus, 4-13, 5-31, 6-42
GetString, 4-58, 4-59, 5-36
GetSwapUnitAttributes, 4-42
GetSwitches, 8-11, 8-12
GetTableBase, 2-26
GetTimeFromTimeServer, 8-12
getTimeout, 3-17
GetTimeoutProcedure, 3-17
GetType, 4-15
GetUniqueConnectionID, 6-11, 6-12,
 6-18
GetUniversalID, 2-10
GetUpdate, 4-57
GetVolumeBootFile, 8-8
GetVolumeLocation, 8-16
GetWord, 3-6, 3-7
GetWordProcedure, 3-15
GetYesOrNoProc, 10-20
 global frame, 1-6, 2-19
 validation, 2-23
 global frame space, 4-43
GlobalFrame, 2-23, 9-4
gmtEpoch, 2-12, 4-59
granularity, 5-3
 Greenwich mean time, 2-12
 comparison, 2-12
GreenwichMeanTime, 2-12, 7-11
Handakuon, 5-9
Handle, 3-1, 3-3, 3-5, 3-9, 3-10, 3-11,
 3-13, 3-14, 3-19, 4-4, 4-44, 4-46,
 5-16, 5-20, 5-32, 5-33, 5-34, 5-35,
 6-11, 6-12, 6-13, 6-47, 6-48, 9-4
handle, 1-9
hardMicrocode, 4-7, 4-8, 8-7, 8-8
 hardware devices
 control of, 1-10
hardwareError, 4-3, 4-13, 5-22, 5-23
hasBorder, 5-13
hasPilotVolume, 4-3, 4-6
hBusy, 10-17
hCRC1, 10-17
hCRC2, 10-17
hCRCerr, 10-17
hDelSector, 10-17
hDiskChng, 10-17
 head, 1-2, 8-1, 8-2
Header, 4-24, 4-25
 headers, 8-5
Heap
DEFINITIONS, 4-49
heap, 4-1, 4-43
MDS, 4-49
normal, 4-49
performance impact, 4-51
uniform, 4-49
hErrDetc, 10-18
hex, 7-5
hexadecimal, 10-18
hexbyte, 10-18
hExpec1, 10-17
hExpec2m, 10-17
hFirst, 10-17
hGoodComp, 10-18
hHead, 10-18
hHeadAddr, 10-18
HighByte, 2-7
HighHalf, 2-7
hIlliStat, 10-18
hIncrLngth, 10-18
Histogram, 10-6
hLast, 10-18
hObser1, 10-18
hObser2, 10-18
hop, 6-23
HostNumber, 2-10, 6-1, 7-5
hReadHead, 10-18
hReadSector, 10-18
hReadStat, 10-18
hReady, 10-18
hRecal, 10-18
hRecalErr, 10-18
hSector, 10-18
hSectorAddr, 10-18
hSectorCntErr, 10-18
hSectorLgth, 10-18
hSeekErr, 10-18
hTimeExc, 10-18
hTrack, 10-18
hTrack0, 10-18, 10-18
hTwoSide, 10-18
hWriteDelSector, 10-18
hWritePro, 10-18
hWriteSector, 10-18
iBadContext, 10-18
iBadLabel, 10-18
iBadSector, 10-18
iBadTrack0, 10-18
IBM, 5-17
ibm2770Host, 6-28
ibm3270Host, 6-28
ibm6670, 6-28
ibm6670Host, 6-28
iCheckPanel, 10-18
iCIERec, 10-18
iCleanDone, 10-18

iCleanProgress, 10-18
ID, 4-2, 4-10, 4-11, 4-17
idle line probes, 6-15
iErrDet, 10-18
iErrNoCRCErr, 10-18
iExerWarning, 10-18
iFirst, 10-18
iFormDone, 10-18
iFormProgress, 10-18
iFormWarning, 10-18
ignore, 5-35
iHardErr, 10-18
iHeadDataErr, 10-18
iInsertDiagDisk, 10-18
iInsertFormDisk, 10-18
iLast, 10-18
IllegalEnumerate, 4-58
illegalLog, 4-55
immediate timeout, 6-5
implementation module
 BackstopImpl.bcd, 9-1
 BackstopNubImpl.bcd, 9-1
 BSMemCache.bcd, 9-1
 Communication, 6-23
 Communication.bcd, 6-4, 6-9
 FloppyImpl.bcd, 5-21
 FormatImpl.bcd, 7-2
 FormatPilotDiskImpl.bcd, 8-5
 Loader.bcd, 2-24
 LogFileImpl.bcd, 4-55
 LogImpl.bcd, 4-55
 MemCacheNub.bcd, 9-1
 OthelloOpsImpl.bcd, 8-5
 PilotKernel.bcd, 1-2, 8-1, 8-2
 RS232CIO.bcd, 6-27
 RuntimeLoader.bcd, 8-2
 StringsImplA.bcd, 7-5
 StringsImplB.bcd, 7-5
 SupervisorImpl.bcd, 2-30
 TimeImpl.bcd, 7-10
 TTYPortChannel.bcd, 5-28
 UtilityPilotKernel.bcd, 1-2, 8-1,
 8-3
 VMMMapLogImpl.bcd, 9-1
implementors, 2-33
imports
 unbound, 2-26
in-band
 attention, 6-22
 signal, 3-8
incompatibleSizes, 5-23
incompleteSwapUnits, 4-33, 4-35
Inconsistent, 4-58
Index, 4-57
IndexOutOfRangeException, 7-12
infinite wait time, 6-5
infiniteWaitTime, 6-11, 6-14, 6-17
infinity, 6-23
initial microcode, 5-27, 8-4
InitializeCondition, 2-19, 2-20
InitializeMonitor, 2-19
InitializePool, 2-36
initialMicrocodeSpaceNotAvailable,
 5-27
Inline
 DEFINITIONS, 2-6
inload, 8-17
input streams
 alternate, 5-35
InputOptions, 3-4, 3-8, 3-14
InsertRootFile, 4-16
Install, 4-55
InstallBootMicrocode, 8-7
InstallPhysicalVolumeBootFile,
 8-13, 8-15
InstallVolumeBootFile, 8-13, 8-14,
 8-15
instance data, 2-32
instanceData, 2-32
InsufficientSpace, 4-12, 4-21, 4-33,
 4-34, 4-40, 4-41, 9-2
insufficientSpace, 2-35, 4-3, 4-13,
 4-51, 5-26
integer, 10-18
inter-processor communication, 6-1
interesting event, 2-30
interface
 volume, 8-4
internal buffering, 3-3, 3-9
internalStructures, 4-7, 4-8
Internet Datagram Protocol, 6-1
Internet Transport Protocols, 6-1
Internetwork routers, 6-23
internetwork topology, 6-23
internetworking, 8-12
InterpretHandle, 4-4, 5-20
Interrupt, 2-28, 9-1
interrupt, 9-3
interrupt key, 2-17, 2-28, C-1
Interval, 2-35, 4-30
interval, 1-5, 1-6, 4-32, 4-33
interval timing, 2-14
intra-processor communication, 6-1
Invalid, 7-11
InvalidArguments, 6-56, 6-57
invalidConfig, 2-24, 2-25
invalidDrive, 5-20
InvalidFile, 4-58
invalidFile, 4-55, 4-56
invalidFormat, 5-22
InvalidFrame, 2-23
InvalidGlobalFrame, 2-23, 2-26

invalidHandle, 4-3, 4-4, 4-5, 4-6, 4-7,
 5-17
invalidHeap, 4-51, 4-54
InvalidLineNumber, 5-29, 6-38
invalidNode, 4-47, 4-49, 4-54
InvalidNumber, 5-37, 7-7
InvalidOperation, 3-18, 6-13, 6-58
invalidOwner, 4-54
invalidPageNumber, 5-28
InvalidParameter, 6-38
invalidParameter, 10-9
InvalidParameters, 8-14, 8-16, 8-17
invalidParameters, 2-35, 4-20, 4-21,
 4-40, 4-41, 4-42, 4-51
invalidProcedure, 4-38
InvalidProcess, 2-18
invalidRootFileType, 4-16
invalidSegment, 4-47, 4-48
invalidSize, 4-51
InvalidSubsystem, 2-32, 2-33
invalidSwapUnitSize, 4-33, 4-35
InvalidVersion, 8-8, 8-14
invalidVolumeHandle, 5-21, 5-22
invalidWindow, 4-33, 4-34, 4-36
invalidZone, 4-47, 4-48, 4-54
invertPattern, 10-15
InvertScreen, 10-15
IOError, 4-32, 4-35, 4-38, 4-39
IOError, 4-36
iOneSided, 10-18
irregular, 4-33
iRunStdTest, 10-18
IsBound, 2-26
iSoftErr, 10-18
 isolated page zero, 4-25
isPilot, 4-5
IsReady, 4-4
IsTimeValid, 8-12
isUtilityPilot, 2-16
 italics
 as metasymbols, 1-9
ItemCount, 2-35
ItemIndex, 2-35
iTxn, 10-18
iTwoSided, 10-18
iUnitNotReady, 10-18
iVerDataErr, 10-18
 January 1 1968, 2-12
japanese, 10-14
 Japanese keyboard, 5-6
JLevelIVKeys
 DEFINITIONS, 5-6
 job control facilities, 1-3
JOIN, 2-18
kAndCTL, 10-16
kAndShift, 10-16

kAtSign, 10-16
kBackSlash, 10-16
kBreak, 10-16
kCaret, 10-16
kCTLC, 10-16
kCTLStop, 10-16
kEndAdj, 10-16
kEscape, 10-16
keyboard, 5-6, 5-16, 10-15
keyboard, 5-12, 5-13, 5-16
KeyboardAndMouseTest, 10-14
KeyboardType, 10-14
KeyboardType, 10-14
Keyname, 5-6
Keys
 DEFINITIONS, 5-6
keyset, 5-12, 5-16
KeyStations
 DEFINITIONS, 5-6
kFillScreen, 10-16
kHyphen, 10-16
Kill, 4-37, 4-38, 4-39
kKey, 10-16
kLearColon, 10-16
kLeftBracket, 10-16
kLetter, 10-16
kLineFeed, 10-16
kNumeral, 10-16
kReturnKey, 10-16
kRightBracket, 10-16
kSemiColon, 10-16
kShAt, 10-16
kShBackSlash, 10-16
kShBreak, 10-16
kShCaret, 10-16
kShColon, 10-16
kShComma, 10-16
kShHyphen, 10-16
kShLeftBracket, 10-16
kShPeriod, 10-16
kShRightBracket, 10-16
kShSemiColon, 10-16
kShVirgule, 10-16
kSpBar, 10-16
kTermAdj, 10-16
kTermTest, 10-16
kTestKey, 10-16
kTypeCharFill, 10-16
kTypeComma, 10-16
kTypeHair, 10-16
kTypePeriod, 10-16
kUnknown, 10-16
kVirgule, 10-16
 labels, 8-5
lastPageCount, 2-2, 4-2, 4-11, 4-18,
 4-29

lastPageNumber, 2-2, 4-2, 4-11,
 4-18, 4-29
lastPageOffset, 2-3, 4-29
latch bit, 5-31, 5-32
latchBitClear, 6-35
LatchBitClearMask, 6-33
Layout, 4-9
LDIVMOD, 2-8
Lear Siegler ADM-3 display, 5-28
length, 7-7
lengthIs5bits, 5-31
lengthIs6bits, 5-31
lengthIs7bits, 5-31
lengthIs8bits, 5-31
LengthRange, 10-10
Level, 4-56
level 0, 6-1
level 1, 6-1
level 2, 6-2, 6-4, 6-9
LevelIVKeys
 DEFINITIONS, 5-6
LF, 7-1
LFDisplayTest, 10-15
Life, 4-31
lifetime, 6-10
Line, 7-3
lineCountError, 5-15
LineOverflow, 5-33, 5-35
LineSpeed, 5-30, 5-31, 6-30, 6-34
lineSpeed, 6-35
LineType, 6-30, 6-34
LinkageFault, 2-27
Listen, 6-13, 6-14, 6-17
listen, 6-10
listener, 6-11
ListenerHandle, 6-13
ListenerHandle, 6-11
ListenError, 6-14, 6-17
ListenErrorReason, 6-17
ListenTimeout, 6-13, 6-17
LoadConfig, 2-24
loader
 bootstrap, 8-1
Loader.bcd, 2-24
loading an object file, 8-2
loadstate, 2-17
local frame, 1-6, 2-19, 2-27
 validation, 2-23
local frame space, 4-43
local network number, 8-12
 default, 8-13
local networks, 8-12
local time parameters, 2-13
localHostNumber, 2-11
LocalSystemElement, 6-66
LocalTimeParameters, 2-13
LocalTimeParametersUnknown,
 2-14, 7-11
Log, 4-26
Log, 9-1
Log
 DEFINITIONS, 4-55
log entries
 enumeration of, 4-58
log file, 4-23, 4-24, 4-27, 4-55
 backstop, 9-1, 9-2
 current, 4-55, 4-57, 4-58, 4-59
 enumeration of, 9-5
 initializing, 4-55
 minimum size, 4-55
 opening, 4-56
 properties, 4-57
 reading, 4-58
 resetting, 4-59
 restart entry, 4-56, 4-59
 writing entries, 4-56
logBitsPerByte, 2-1
logBitsPerChar, 2-1
logBitsPerWord, 2-1
logBytesPerPage, 2-2
logBytesPerWord, 2-2
logCap, 4-55
logCharsPerPage, 2-2
logCharsPerWord, 2-2
.LogError, 9-2
LogFile, 4-58
LogFile, 9-1
LogFile
 DEFINITIONS, 4-55
LogFileImpl.bcd, 4-55
LogFormat, 4-24
LogFrame, 9-3, 9-4
logging
 controlling, 4-56
logical operations, 2-8
logical record, 6-21
logical volume, 2-30, 4-8, 4-22, 8-4
 attributes, 4-15
 close, 4-14
 consistant state, 4-6
 create, 4-12
 enumeration of, 4-8, 4-10, 4-14
 erase, 4-12
 errors, 4-12
 ID, 4-10
 label, 4-16
 maximum number, 4-10
 maximum size, 4-11
 name, 4-10
 open, 4-14
 opening, 8-3
 root directory, 4-16

spanning physical volumes, 4-11
 status, 4-13
LogicalVolumePageNumber, 8-10
LogImpl.bcd, 4-55
logNoEntry, 4-57
logNotOpened, 4-56, 4-57
LogProcess, 9-3
LogSeal, 4-24, 4-25
logWordsPerPage, 2-2
Long, 2-3
LONGCARDINAL, 6-62
LONGDESCRIPTOR, 6-46, 6-63
LONG DESCRIPTOR FOR ARRAY, 6-61
LONG INTEGER, 6-62
LONGPOINTER, 6-46, 6-64
LONG STRING, 6-62
LongBlock, 3-13, 6-21
LongCOPY, 2-7
LongCOPYReverse, 2-7
LongDecimal, 7-4
LongDiv, 2-8
LongDivMod, 2-8
LongMult, 2-8
LongNumber, 2-3, 2-7, 7-4
LongOctal, 7-4
LongPointerFromPage, 2-3, 4-30
LongString, 7-3
LongSubString, 7-2
LongSubStringItem, 7-2, 7-3
LookUpRootFile, 4-17
loopOnError, 10-19
loopOnThisError, 10-18
lost, 4-7, 4-8
LowByte, 2-7
LowerCase, 7-6
LowHalf, 2-7
LSAdjust, 10-16
LSMessage, 10-16
LSTest, 10-17
LTP, 7-11
 machine, 1-2
 machine-independent environment, 1-2
mailDate, 5-37, 7-4
Main Data Space, 4-40
 main data space, 1-6
 maintenance panel, 2-16, 2-17, 8-12
MakeBoot, 2-20, 2-26, 8-2, 8-3, 8-10, 8-13, 8-14
MakeBootable, 8-8, 8-13, 8-14, 8-15, 8-16
MakeDLionBootFloppy, 5-28
MakeDLionBootFloppyTool, 8-7
MakeFileList, 4-26
MakelImage, 5-24
MakeMDSNode, 4-54
MakeMDSString, 7-9
MakeNode, 4-48, 4-54
MakePermanent, 4-22
MakeReadOnly, 4-39
MakeString, 7-8
MakeUnbootable, 8-8, 8-16
MakeWritable, 4-39
Map, 4-26, 4-32, 4-35, 8-14
 map logging, 2-17
 map unit, 1-5, 1-5, 4-30, 4-33, 4-34, 4-35
MapAt, 4-40, 4-41
 mapped spaces, 4-21
 mapping, 1-5, 4-30, 4-32
MarkPageBad, 4-9, 4-10, 8-6
 marshalling, 6-61
 master mode, 1-3
maxBlockLength, 6-5, 6-6, 6-8
maxCARDINAL, 2-3
maxCharactersInLabel, 5-25
maxData, 10-10
maxEntriesInRootDirectory, 4-16
 maximum internet packet, 6-10
 maximum internet packet size, 6-6
 maximum internetwork length, 6-1
 maximum packet lifetime, 6-5
maxINTEGER, 2-3
maxLONGCARDINAL, 2-3
maxLONGINTEGER, 2-3
maxLength, 4-6, 4-9, 4-13
maxPagesInMDS, 2-2
maxPagesInVM, 2-2
maxPagesPerFile, 4-18, 4-22
maxPagesPerVolume, 4-11
maxSizeExceeded, 4-51
maxSubvolumesOnPhysicalVolume, 4-13
maxWellKnownSocket, 2-12
MDS, 4-40, 4-49
MDS, 1-6
 MDS zone, 2-19
MDSZone, 4-49, 4-52
MemCacheNub.bcd, 9-1
 memory management
 performance, A-2
MemoryStream
 DEFINITIONS, 7-12
Mesa development environment, 8-2
Mesa emulation
 microcode, 8-8
Mesa Language Manual, 2-18
Mesa Processor Principles of Operation, 1-2, 2-1, 2-6, 5-13
Mesa to Courier mapping, 6-59, 6-61
Mesa type-checking, 1-3
Mesa User's Guide, 4-10, 8-2, 8-7

Mesa variant record, 6-63
MesaDEFFileType, 4-19
MesaDEFileTypes, B-2
MesaEventIndex, 2-31
MesaFileType, 4-19, B-1
metasymbols, 1-9
microcode, 1-2, 8-1, 8-8
 initial, 8-7
microcode files, 8-7
 installing, 8-7
MicrocodeInstallFailure, 8-7
microcodeTooBig, 8-7
Microseconds, 2-15
MicrosecondsToPulses, 2-15
Milliseconds, 2-19
minimumNodeSize, 4-44, 4-50
minINTEGER, 2-3
minPagesPerVolume, 4-11
missing, 4-25, 4-27
missing page, 4-25
missingCode, 2-24, 2-25
MissingPages, 4-20, 4-34, 4-37, 8-16
ModemChange, 10-10
ModemSignal, 10-10
monitor, 1-4, 1-10, 2-18, 2-22, 2-30
monitor lock, 2-19
 uninitialized, 2-19
mouse, 5-16
mouse, 5-12, 5-13, 5-16
 coordinates, 5-16
move, 2-6
MsecToTicks, 2-19
multiple physical volumes, 8-10
multipleLogicalVolumes, 4-9
multipleWindows, 5-15
nameRequired, 4-3, 4-6, 4-9, 4-13,
 4-16
NARROW, 2-27
NarrowFault, 2-27
needsConversion, 4-3, 4-7, 4-23
needsRiskyRepair, 4-23
NeedsScavenging, 4-3, 4-5, 4-12,
 4-15, 4-16, 4-22, 8-9, 8-11, 8-17
needsScavenging, 5-22
NetAccess, 6-31, 6-34
NetFormat, 7-5
network address, 1-10, 2-10, 6-66
 editing, 7-5
 when connected to many
 networks, 2-11
Network stream, 1-7, 3-18
NetworkAddr, 6-11
NetworkAddress, 2-10, 6-1, 6-13,
 6-47, 7-5
NetworkAddresses, 6-9
NetworkNonExistent, 6-24
NetworkNumber, 2-10, 6-1, 7-5
NetworkStream, 1-7, 6-2
 DEFINITIONS, 6-9
NEW, 2-19, 2-29, 3-19, 4-43, 4-49,
 4-51, 4-52, 4-54
newClearinghouseSocket, 2-11
NewConfig, 2-24, 2-25
NewLine, 5-34
NextAction, 10-15
nextPattern, 10-15
nil, 4-46, 4-48
no, 10-20
 noAnswerOrBusy, 10-1
NoBackingFile, 5-34
noChecking, 10-19
noCommunicationFacilities, 8-12
NoCommunicationHardware, 6-43
node
 minimum size, 4-45, 4-51
noDebugger, 8-9
NoDefaultInstance, 5-32
nodeLoop, 4-47
NodeSize, 4-49
noErrorFound, 10-18
noHardware, 10-9
noMoreNets, 10-5
none, 5-31, 5-34
noneDeleted, 4-24
nonEmptySegment, 4-48
nonPilot, 4-13
Nop, 5-20
noProblems, 4-7, 4-8
noResponse, 8-12
noRetries, 8-5
normal, 3-4, 4-13
noRoomInZone, 4-48
noRouteToSystemElement, 10-1
NoRS232CHardware, 6-38
noScrollWindow, 5-15
noSeconds, 5-37, 7-4
noSuchDiagnostic, 10-1
noSuchDrive, 4-3, 4-4, 5-22
noSuchLine, 10-9
noSuchLogicalVolume, 4-3, 4-8
noSuchPage, 4-27
NoSuchProcedureNumber,
 6-48, 6-52, 6-57
NoSuchSubsystem, 2-32
NoTableEntryForNet, 6-24
NotAFault, 9-3
notAllocated, 4-41
NotAPilotDisk, 8-5, 8-7
notation, 1-8
notDiagDiskette, 10-18
Note, 1-9
NoteArrayDescriptor, 6-63

NoteBlock, 6-65
NoteChoice, 6-63
NoteDeadSpace, 6-65
NoteDisjointData, 6-63
NoteLongCardinal, 6-62
NoteLongInteger, 6-62
NoteParameters, 6-64
NotErrorEntry, 9-4, 9-5
Notes, 6-59
notes object, 6-59
NoteSize, 6-62
NotesObject, 6-59, 6-61, 6-62, 6-63, 6-64, 6-65
NoteSpace, 6-61, 6-64
NoteString, 6-62
NOTIFY, 2-15, 2-19, 2-22
NotifyAllSubsystems, 2-33
NotifyDirectSubsystems, 2-34
NotifyRelatedSubsystems, 2-33
notInitialBootFile, 8-9
NotLoggingError, 9-3
notMapped, 4-35, 4-36
NotOnline, 4-12, 4-13, 4-15, 4-16, 4-21, 4-26, 4-34, 4-36, 8-9, 8-11, 8-16, 8-17
NotOpen, 4-12, 4-16, 4-21, 4-26, 4-34, 4-37, 8-8, 8-9, 8-16
notPilot, 4-5
notReady, 4-3, 4-5, 5-22
NoTTYPortHardware, 5-29
noWindow, 2-25, 4-33, 4-34
NS Communication System, 1-3
NSConstants, 2-11
 DEFINITIONS, 2-9, 6-2
nsProtocol, 6-28
nsSystemElement, 6-28
nsSystemElementBSC, 6-28
NUL, 7-2
null, 2-5, 4-58
nullAgentProcedure, 2-31
nullBadPage, 4-10
nullBlock, 2-2, 6-8
nullBootFile, 8-9
nullBootFilePointer, 5-27
nullChannelHandle, 5-28
nullDeviceIndex, 4-3
nullDrive, 5-20
nullEvent, 2-31
nullExchangeHandle, 6-5
nullFile, 4-17, 4-26
nullFieldID, 5-26, 5-28
nullFrame, 9-3, 9-4
nullHandle, 4-45, 5-32
nullHostNumber, 2-10
nullID, 2-10, 4-2, 4-9, 4-10, 4-11, 4-12, 4-17
nullIndex, 4-57, 9-5
nullInterval, 4-30
nullLineNumber, 6-31, 6-34
nullNetworkAddress, 2-10
nullNetworkNumber, 2-10
nullParameters, 6-49, 6-50, 6-51, 6-53, 6-57, 6-58
nullProcess, 9-3
nullProgram, 2-23
nullSegment, 4-46
nullSocketNumber, 2-10
nullSubsystem, 2-32
nullSubVolume, 8-10
nullType, 2-4
nullVolumeHandle, 5-21
Number, 6-43, 7-3
NumberFormat, 5-38, 7-3, 7-4
NWords, 4-50
ObjAlloc
 DEFINITIONS, 2-34
Object, 3-9, 3-14, 3-18, 3-19, 6-11, 6-47
object allocation, 2-34
object file, 8-2, 8-2
Objects, 6-48
Octal, 7-4
octal, 10-18, 7-5
OctalFormat, 7-3
odd, 5-31
off, 4-56, 5-12
Offline, 4-5
ok, 5-35
okay, 4-7, 4-7, 4-8, 4-45, 4-49
on, 5-12
on-line, 8-3
one, 5-25, 5-31
one024, 10-18
one28, 10-18
oneAndHalf, 5-31
online, 4-5, 8-5
OnlineDiagnostics
 DEFINITIONS, 10-14, 10-16, 10-17
onlyEnumerateCurrentType, 4-14
onlyOneSide, 5-25
onlySingleDensity, 5-25
Open, 4-14, 4-22, 4-55, 4-59, 5-22
openRead, 4-14
openReadWrite, 4-14
OperationClass, 6-34
optional packages, 8-2
orphan, 4-25
orphan page, 4-25, 4-28
OrphanHandle, 4-25, 4-28
orphanNotFound, 4-28
Othello, 2-16, 4-10, 8-2, 8-4, 8-13
OthelloOps, 8-4, 8-7

DEFINITIONS, 8-3
OthelloOpsImpl.bcd, 8-5
other, 8-7, 8-9, 9-3
otherError, 4-51, 10-9
out-of-band
 attention, 6-21
 signal, 3-8
Outcome, 6-43
outload, 8-17
outload file, 9-2
OutLoadInLoad, 8-17
OutOfInstances, 5-32
outsideXeroxFirstSocket, 2-12
outsideXeroxLastSocket, 2-12
Overflow, 2-15, 4-57
OverLapOption, 2-6
owner checking, 2-17, 4-51
OwnerChecking, 4-54
OwnerCheckingMDS, 4-54
Pack, 7-11
packager, 1-6
Packed, 7-10
packet, 6-1, 6-1
Packet Exchange Protocol, 6-4
packet exchange protocol, 6-2
PacketExchange, 1-7, 6-2
 DEFINITIONS, 6-4
packets, 1-6
page alignment, 5-3
page fault service time, A-2
page number, 4-18
PageCount, 2-2, 4-2, 4-11, 4-18, 4-29
pageCountTooSmallForVolume,
 4-13
PageFromLongPointer, 2-4, 4-30
PageNumber, 2-2, 4-2, 4-11, 4-18,
 4-29, 4-30, 5-21
PageOffset, 2-3, 4-29, 4-30
PagesForImage, 5-24
PagesFromWords, 4-43
Parameter, 5-30, 6-34
parameter area, 6-46, 6-61, 6-62,
 6-64, 6-65
Parameters, 6-49, 6-52
ParameterType, 6-34
Parity, 5-30, 6-31, 6-34, 6-35
parityError, 5-29, 5-30
partialLogicalVolume, 4-9
partiallyOnLine, 4-14
PatternType, 10-11
Pause, 2-15, 2-22
PC, 9-4
PerformanceToolFileType, 4-19
permanent, 4-23, 4-25
permissions, 4-31
physical record, 6-20
physical volume, 2-13, 2-30, 4-5, 8-2,
 8-4
 consistant state, 4-6
 creation, 4-6
 enumeration of, 4-9
 errors, 4-2
 formatting, 8-5
 identifier, 8-15
 name, 4-2, 4-9
 size, 4-2
PhysicalMedium, 6-23
PhysicalRecord, 6-31, 6-36
PhysicalRecordHandle, 6-31, 6-36
PhysicalVolume, 8-5
 DEFINITIONS, 4-1
PhysicalVolumeID, 2-10, 4-2
physicalVolumeUnknown, 4-2, 4-3,
 4-5, 4-6, 4-9, 4-10, 4-13, 8-9, 8-10
physicalVolumeUnknown, 4-8
pilot, 8-7, 8-8
Pilot, 8-4, 8-5
 boot loader, 8-13
 disk utility, 8-2, 8-4
 execution speed, A-2
 initialization, 8-1, 8-2, 8-9
 microcode, 8-4
 performance requirements, A-1
 physical memory requirements,
 A-1
 program, 8-1
 released version of, 1-1
 restart, 8-3
 swapping, 1-5, 4-30, 4-31
 switches, 8-2, 8-11, 8-15
 System Components, 8-1
Pilot Emergency Interrupt, C-1
PilotClient, 8-3
 DEFINITIONS, 2-28
PilotDisk, 2-4, 4-3
PilotFileType, B-1, B-2
PilotKernel.bcd, 1-2, 1-9, 8-1, 8-2,
 D-1
pilotSnapshot, 8-7, 8-8
pipeline, 1-7, 3-1, 3-2, 3-5, 3-9, 3-10,
 3-11, 3-13
pixelsPerInch, 5-12
plain, 5-34
Pointer, 4-43
PointerFault, 2-28
PointerFromPage, 4-43
pointerPastEndOfVirtualMemory,
 4-31
PopAlternateInputStreams, 5-34
PORT, 3-18
port, 2-27
PortFault, 2-27

Position, 3-9
power off, 2-30
power on
 automatic, 2-30
PowerOff, 2-15
pre-emptive allocation, 6-38
primary storage, 4-47
priorities
 ranking of, 2-21
Priority, 2-21
priorityBackground, 2-21
priorityForeground, 2-21
priorityNormal, 2-21
probablyNotPilot, 4-5
probablyPilot, 4-5
Problem, 4-24
procedures
 activation and deactivation, 4-38
Proceed, 9-2
PROCESS, 2-18
Process, 9-3
Process
 DEFINITIONS, 2-18
process, 1-4, 1-6, 1-10, 2-18
 abort, 2-21
 active, enumeration of, 9-3
 awakening, 2-30
 dead, 2-18
 detached, 2-21
 fork, 2-20, 2-21
 lightweight, 1-4
 live, 2-18
 maximum number, 2-20
 performance, A-3
 priority, 2-21, C-1
 suspend, 2-22
 synchronization, 2-23
 validation, 2-18
processor
 ID, 8-3
 setting of clock, 8-12
 yielding control, 2-22
Product Common Software, 1-9,
 5-28, 5-32, 7-2, 7-5, 7-10
product system, 9-1
productSoftware, 7-5
PROGRAM, 2-25
program
 logical correctness of, 2-23
protection, 1-3
protocolCertificationControl, 2-12
protocolCertificationTest, 2-12
Prune, 4-53
PruneMDS, 4-53
PSBIndex, 9-4
pseudo-Mesa declarations, 1-8
pulse definition, 2-14
Pulses, 2-14
PulsesToMicroseconds, 2-15
pupAddressTranslation, 2-11
PushAlternateInputStream, 5-34
Put, 5-29, 6-40
PutBackChar, 5-34
PutBlank, 5-36
PutBlanks, 5-36
PutBlock, 3-6, 3-7, 3-10, 3-12, 4-56,
 5-34, 9-2
PutByte, 3-7
PutByteProcedure, 3-16
PutChar, 3-7, 5-36
PutCR, 5-36, 10-16
PutDate, 5-36
PutDecimal, 5-38
PutLine, 5-37
PutLongDecimal, 5-38
PutLongNumber, 5-38
PutLongOctal, 5-38
PutLongSubString, 5-37
PutMesaChar, 10-17
PutMessage, 10-17
PutMessageProc, 10-19
PutNumber, 5-38
PutOctal, 5-38
PutProcedure, 3-15
PutString, 3-7, 4-56, 5-37
PutSubString, 5-37
PutText, 5-37
PutWord, 3-7, 4-56
PutWordProcedure, 3-16
PVLocation, 8-16
q2000, 2-5
q2010, 2-5
q2020, 2-5
q2030, 2-5
q2040, 2-5
q2080, 2-5
quad-word alignment, 5-3
Quantum, 2-5
Quiesce, 5-29, 5-30, 5-32
quiescent state, 2-30
quit, 10-15
Read, 5-22, 5-23
ReadBadPage, 4-27
ReadID, 5-20
ReadOnly, 2-25, 4-12, 4-13, 4-15,
 4-16, 4-21, 4-26, 4-34, 4-36, 4-39,
 8-9, 8-11
readOnly, 4-31, 4-36
ReadOrphanPage, 4-28
ReadSectors, 5-19
readWrite, 4-31

Recalibrate, 5-20
recording information, 4-55
Recreate, 4-45
references
 informational, F-1
 mandatory, F-1
RejectRequest, 6-8, 6-9
ReleaseDataStream, 6-59
remark, 4-56
remote procedure calling, 6-46
remote program, 6-46
RemoteErrorSignalled, 6-51, 6-57
remoteSystemElementNot
 Responding, 10-1
removable medium, 1-6
RemoveCharacter, 5-37
RemoveCharacters, 5-37
RemoveRootFile, 4-17
RemoveSegment, 4-48
RemoveSubsystem, 2-32
repair, 4-23
repaired, 4-7
RepairStatus, 4-7
RepairType, 4-7, 4-23
Replace, 7-10
ReplaceBadPage, 4-27
ReplaceBadSector, 5-23, 5-24
replier, 6-4, 6-5
RequestHandle, 6-5
RequestObject, 6-5
requestor, 6-4, 6-5
requestToSend, 6-35
reservedType, 4-20, 4-21
ReserveType, 6-32, 6-37
Reset, 4-57, 4-59
reset, 4-57
ResetAutomaticPowerOn, 2-16
ResetUserAbort, 5-33
resource
 allocation, 1-3
 new, acquisition of, 2-30
 shared, acquisition and release,
 2-29
RESTART, 2-27
Restart, 4-59, 6-41, 9-2
restart
 file, 2-29
 message, 9-1
 system, 2-11
Results, 6-48, 6-52, 6-59
results, 6-50, 6-51, 6-52
retransmission, 6-4, 6-15
retransmissionInterval, 6-7
RETRY, 3-4, 3-9
RetryCount, 6-32, 6-43
RetryLimit, 8-5
retryLimit, 8-5
return, 4-35
ReturnWait, 4-35
RewritePage, 4-27
ripple, 2-6
riskyRepair, 4-7, 4-8, 4-23
root page, 2-13
RootDirectoryError, 4-16, 9-2
RootDirectoryErrorType, 4-16
rootFileUnknown, 4-16, 4-17
router, 1-6, 8-12
RoutersFunction, 6-24
routing delay, 6-23
routing protocol, 6-2
routing table, 6-23
routing table cache fault, 6-23
routingInformationSocket, 2-11
RPC, 6-46
RS232C
 DEFINITIONS, 6-32
RS232CCorrespondents
 DEFINITIONS, 6-27
RS232CDiagError, 10-9
RS232CErrorReason, 10-9
RS232CIO.bcd, 6-27
RS232CLoopback, 10-8
RS232CParams, 10-11
RS232CTestMessage, 10-12
Rubout, 5-33, 5-35
Run, 2-17, 2-29, 8-3, 8-13, 9-2
RunConfig, 2-24, 2-25
Runtime, 8-2
 DEFINITIONS, 2-23
RuntimeLoader.bcd, 8-2
sa1000, 2-4
SA1000lastPageOfMicrocode, 8-6
SA1000pagesPerTrack, 8-6
SA1000startOfMicrocode, 8-6
sa1004, 2-4
SA1004pagesPerCylinder, 8-6
sa4000, 2-4
SA4000lastPageOfMicrocode, 8-6
SA4000startOfMicrocode, 8-6
sa4008, 2-4
SA4008pagesPerCylinder, 8-6
sa800, 2-5
SA800, 8-7
safeRepair, 4-7, 4-8, 4-23
SBSOFileType, 4-19
Scan, 8-6
scan line zero, 5-13
Scavenge, 4-7, 4-23, 4-25, 4-26, 5-27
scavenge, 4-1, 4-23, 8-2, B-1
 physical volume, 4-6, 4-7
Scavenger, 8-4
DEFINITIONS, 4-22

ScavengerStatus, 4-7
ScratchMap, 4-33, 4-35
screenHeight, 5-12, 10-15, 10-16
screenWidth, 5-12, 5-13, 10-15,
 10-16
Scroll, 5-15
scroll window, 5-14, 5-15
scrollingInhibitsCursor, 5-15
scrollXQuantum, 5-14
scrollYQuantum, 5-14
SDDivMod, 2-7, 2-8
Seconds, 2-20
SecondsSinceEpoch, 2-12
SecondsToTicks, 2-20
SectorLength, 10-18
sectors, 8-5
segment, 4-44
 attributes, 4-47
SegmentHandle, 4-46
segmentTooSmall, 4-47
SelfDestruct, 2-24, 3-19
SendAttention, 3-8
SendAttentionProcedure, 3-17
SendBreak, 5-29, 6-42
SendBreakIllegal, 6-38
SendNow, 3-6, 3-7, 3-12, 3-18
SendNowProcedure, 3-17, 3-18
SendReply, 6-5, 6-8, 6-9
SendRequest, 6-7, 6-8
SEQUENCE, 4-54
sequence, 6-5
sequence packet protocol, 6-2
sequenced, 6-9
sequenced packet protocol, 6-9, 6-19
sequential
 access, 3-1
 data, 1-7
serialization, 6-61
SerializeParameters, 6-65, 7-12
server, 6-10, 6-46
ServerOff, 10-1
ServerOn, 10-1
ServicesFileType, 4-19
SetAccess, 4-39
SetAutomaticPowerOn, 2-16
SetBackground, 5-13, 10-15, 10-16
SetBackingSize, 5-33
SetBootFiles, 5-27, 5-28
SetBorder, 5-13, 10-15, 10-16
SetChecking, 4-47, 4-54
SetCheckingMDS, 4-54
SetContext, 5-18
SetCursorPosition, 5-14, 10-15
SetDebugger, 8-9
SetDebuggerSuccess, 8-9
SetDefaultOutputSink, 7-2
SetDiagnosticLine, 10-12
SetEcho, 5-34, 5-35
SetExpirationDate, 8-11
SetExpirationDateSuccess, 8-11
SetGetSwitchesSuccess, 8-11
SetIndex, 7-12
SetInputOptions, 3-4, 3-15
SetLineType, 6-42
SetLocalTimeParameters, 2-14
SetLocalTimeParameters, 7-11
SetMousePosition, 5-16, 10-15
SetNetworkID, 6-27
SetOverflow, 4-57
SetParameter, 5-30, 5-32, 6-39
SetPhysicalVolumeBootFile, 4-8, 8-8
SetPosition, 3-9
setPosition, 3-17
SetPositionProcedure, 3-17
SetPriority, 2-21, 2-29
SetProcessorTime, 8-12
SetRestart, 4-56, 4-59, 9-3
SetRootFile, 5-26
SetRootNode, 4-46
SetSize, 4-21, 8-14
SetSST, 3-7, 3-7, 3-12
SetSSTProcedure, 3-16
SetState, 4-56, 5-14
SetSwitches, 8-11, 8-12
SetTimeout, 2-15, 2-20
setTimeout, 3-17
SetTimeoutProcedure, 3-17
SetUserAbort, 5-33
SetVolumeBootFile, 8-8
SetWaitTime, 6-18
SetWaitTimes, 6-7, 6-8
shift operations, 2-9
ShortBlock, 3-13, 6-21
Shugart Associates, 2-5
Sides, 5-25
siemens9750, 6-28
Signal, 9-4, 9-5
signal, 9-3
signal
 in-band, 3-8
 out-of-band, 3-8
 uncaught, 9-1
signalAttention, 3-4, 3-7, 3-8
signalEndOfStream, 3-4, 3-5, 3-7,
 3-14
signalEndRecord, 3-4, 3-6, 3-7, 3-12
signalLongBlock, 3-4, 3-7, 3-13, 3-14
SignalMsg, 9-4, 9-5
SignalRemoteError, 6-58
signalShortBlock, 3-4, 3-7, 3-13, 3-14
signalsSSTChange, 3-4, 3-5, 3-7, 3-14

- signalTimeOut**, 3-14
signalTimeout, 3-4, 3-7
 simple routers, 6-23
single, 10-18, 5-17, 5-25
SingleDouble, 10-18
singleLogicalVolume, 4-9
sink, 7-2
 sixteen-word alignment, 5-3
 smooth scrolling, 5-12, 5-14
socket, 2-11, 6-1, 6-15
SocketkNumber, 6-1
SocketNumber, 2-10, 7-5
 sockets, 6-9
 - well-known, 2-11**softMicrocode**, 4-7, 4-8, 8-7, 8-8
 software channel, 5-1, 5-16
 - example of, 5-1**sound generator**, 5-16
SP, 7-2
space, 4-1
 - alive, 4-38
 - dead, 4-38**Space**
 - DEFINITIONS**, 4-29**space machinery**
 - storage, 4-43**SpaceUsage**
 - DEFINITIONS**, 4-29**SplitNode**, 4-49
SSTChange, 3-5, 3-7, 6-20
sstChange, 3-4
Star, 2-31
 Star System Keyboard Requirement Specification, 5-6
stars, 5-34
START, 2-19, 2-27, 2-29, 3-19
Start, 6-43
StartEchoUser, 10-2
startEnumeration, 6-24
StartFault, 2-27
startIndex, 2-2, 2-6, 3-4, 3-5
startIndexGreater Than StopIndexPlusOne, 2-6
startListHeaderHasBadVersion, 8-9, 8-10
State, 4-56, 5-12
 stateless enumerator of
 - active processes, 9-3
 - definition of, 1-8
 - floppy bad sectors, 5-27
 - floppy files, 5-26
 - log entries, 4-58
 - log files, 9-5
 - logical volumes, 4-8, 4-10
 - physical volumes, 4-9
 - subvolumes, 8-10**StatsIndices**, 10-6
Status, 4-13, 4-44, 4-45, 4-47, 4-48, 5-18
StatusWait, 5-31, 6-42
stillMapped, 4-41
STOP, 2-27
Stop, 6-43
stop, 5-35
StopBits, 5-30, 5-31, 6-32, 6-36
stopBits, 6-35
stopIndexPlusOne, 2-2, 2-6, 3-4
stopOnError, 10-19
 storage allocation
 - using heaps, 4-1
 - using zones, 4-1**storageOutOfRange**, 4-45, 4-47
store, 6-61
store, 6-66
Stream, 6-9
 - DEFINITIONS**, 3-1**stream**, 1-7, 1-10
 - component manager, 3-13, 3-18**creation**, 3-3, 3-9, 6-11, 6-13, 6-14
 delete instances of, 3-19
 - example of creating, 3-10**full duplex**, 3-3
half duplex, 3-3
implementation, 7-12
physical records, 3-2, 3-3, 3-11, 3-12
physical records, maximum, 3-12
positioning, 3-9
SubSequence Type, 6-56
timeouts, 3-9, 3-12, 6-18
STRING, 6-61
string, 10-18, 4-58
String
 - DEFINITIONS**, 7-5**string body**
 - allocating from a heap, 4-52**String package**, 7-2, 7-5, 7-10
StringBody, 6-46, 6-62
StringBoundsFault, 7-6
StringProc, 7-2
StringsImplA.bcd, 7-5, 7-10
StringsImplB.bcd, 7-5
StringToDecimal, 7-8
StringToLongNumber, 7-8
StringToNumber, 7-7
StringToOctal, 7-8
stringTooShort, 5-25
style rules, B-2
subscript out of range, 2-27
subsequence type, 3-2
subsequences, 3-2

SubSequenceType, 3-2, 3-4, 3-5, 3-7, 3-7, 3-12, 6-19, 6-56
SubString, 7-3, 7-6
SubStringDescriptor, 7-3, 7-6
SubsystemHandle, 2-32
subsystems, 2-29
 clients-first order, 2-29
 implementors-first order, 2-29
SubVolume, 8-10
subvolume, 8-10
 enumeration of, 8-10
subvolumeHasTooManyBadPages, 4-13
SubVolumeUnknown, 8-10
success, 5-29, 5-30, 8-9
Supervisor
 DEFINITIONS, 2-29
Supervisor error conditions
 recoverable, 2-34
SupervisorEventIndex, 2-31
 DEFINITIONS, 2-29
SupervisorImpl.bcd, 2-30
suppress duplicate, 6-5
Suspend, 6-40
SuspendReason, 6-14
swap unit, 1-5, 1-6, 4-30, 4-31, 4-32
 access, 4-39
 boundary, 4-31
 life, 4-31
 size, 4-32, 4-34, 4-35
swapping
 advice, 4-37
 controlled, 4-37
 demand, 4-37
SwapReason, 9-4, 9-5
SwapUnitOption, 4-33
SwapUnitType, 4-33
Switches, 2-16, 8-11
switches, 2-16
 switches
 boot, 2-16
SyncChar, 6-32
syncChar, 6-35
SyncCount, 6-32
syncCount, 6-35
synchronous, 5-22
synchronous operation, 1-5
 definition of, 1-8
 of physical devices, 1-5
synchronous procedures
 stream, 3-3
System
 DEFINITIONS, 2-9
system
 logical volume, 4-14
 physical volume, 4-14
 power, 2-15, 2-16
 restart, 2-11
 volume, 4-11, 8-2, 8-3
 zones, 2-17
system6, 6-28
systemBootDevice, 2-16
SystemElement, 6-47
systemID, 4-11
systemMDSZone, 4-49, 4-52
systemZone, 4-49, 4-52
t300, 2-5
t300lastPageOfMicrocode, 8-6
t300pagesPerCylinder, 8-6
t300pagesPerTrack, 8-6
t300startOfMicrocode, 8-6
t80, 2-5
t80lastPageOfMicrocode, 8-6
t80pagesPerCylinder, 8-6
t80pagesPerTrack, 8-6
t80startOfMicrocode, 8-6
TAB, 7-2
Table Compiler, 2-26
Tajo, 2-31, 8-2
tBackstopDebuggee, 4-20
tBackstopDebugger, 4-20
tBackstopLog, 4-20
tBootFile, 8-13
tByteCnt, 10-18
tCarryVolumeDirectory, 4-20
tCIERH, 10-18
tCIERS, 10-18
tCIEVer, 10-18
tCIEWDS, 10-18
tCIEWS, 10-18
tClearingHouseBackupFile, 4-20
tDirectory, 4-20
teleDebugSocket, 2-12
temporary, 4-22, 4-23
temporary file, 8-10
TemporaryBooting, 8-2, 8-7, 8-13
 DEFINITIONS, 8-13
terminateOnEndRecord, 3-4, 3-6, 3-7, 3-12, 6-20
TestFileType, 4-19
TextBit, 2-6
tFileList, 4-20
tFirst, 10-18
tHeadDataErr, 10-18
tHeadDisp, 10-18
tHeadErrDisp, 10-18
Ticks, 2-19, 2-20
ticks, 2-19
TicksToMsec, 2-19
Time
 DEFINITIONS, 7-10
time of day, 2-12

filed, 4-45, 4-46
recreating, 4-44
root node, 4-46
sizes, 4-44
wrong version, 4-45
zoneTooSmall, 4-45