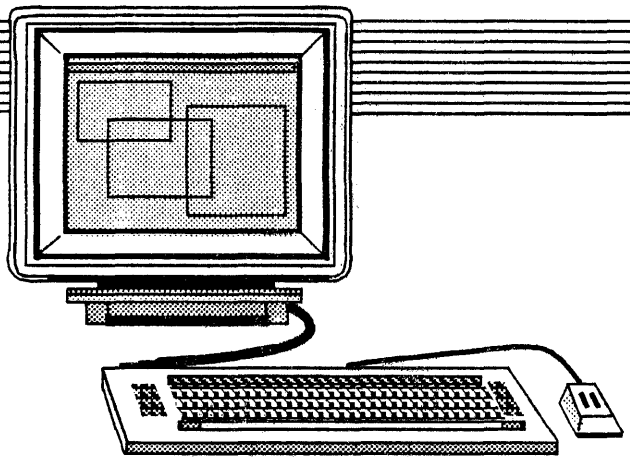


XDE 6.0 UPDATE

Pilot Programmer's Manual

To

Customer/Colleague

From

Holly Wanless

Technical Services and Support

Subject

Pilot Programmer's Manual

Date

February 1989

The enclosed manual is a complete replacement for your existing version. Please remove the old version from the binder, replace it with this new version, and discard the old version.

Pilot Programmer's Manual

XEROX

610E00161
September 1988

Xerox Corporation
XDE Technical Services
475 Oakmead Parkway
Sunnyvale, California 94086

Printed in the United States of America.

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 was prepared for this purpose. Please send your comments to:

Xerox Corporation
XDE Technical Documentation, SVHQ403
475 Oakmead Parkway
Sunnyvale, California 94086

Table of contents

5 I/O Devices

6 Communication

7 Editing and Formatting

8 System Generation and Initialization

9 The Backstop

1.

Introduction

1

Introduction

The Pilot Programmer's Manual 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; that is, applications, certain development and production tools, test programs, and so forth. The description provides sufficient information to allow the programmer to understand the available facilities and to write procedure calls in the Mesa language to invoke them. For each of the facilities of Pilot, the 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 tries to minimize, insofar as possible, 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 use 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. The major categories are as follows:

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 that 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 programs 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.

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`. In addition, a special version of Pilot, contained in the object file `UtilityPilotKernel.bcd`, supports 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 is mentioned in the section that describes the facility.

No explicit mention is made in this document of the location of files. That information is contained in the documentation 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 the clients 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.

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

Master Mode and Protection: No "ironclad" mechanism protects Pilot from errant or malicious client programs, or even 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, neither detailed billing nor accounting for the use of resources within a single system element is needed. In the 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 sharing 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 services are provided by client programs and are likely to vary across the product lines.

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 is described below in some detail.

1.3.1 Processes, monitors, and synchronization

Within a system element, several activities will almost always be 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. 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 instances 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 shareable 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 procedures 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 are 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, 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 about the following: 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 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 wishing 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, then 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, then 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 the volume is implemented on a removable medium, then 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. Clients 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)*. The MDS is a contiguous subset of virtual memory consisting of 2^{16} words (256 pages), any part of which may be addressed by a 16-bit Mesa **POINTER**.

An MDS contains the low-level data structures and mechanisms, such as local frames and trap handlers, necessary for executing Mesa processes. Global frames may also reside in the MDS if modules were compiled to run with pre-Pilot 14.0 ("old") versions or if they are packaged with pre-Pilot 14.0 ("old") versions. "Old" means that the modules either were not recompiled since the 12.0 compiler or were compiled with the 14.0 compiler /o switch.

Each process is associated with one and only one MDS. Although the Mesa processor supports multiple coexisting MDSs, 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, then 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 appropiate 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 definition 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

The methodologies which are used repeatedly in the design of the Pilot functions 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 usually has a name of the form **null***object*, 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 whereupon Pilot returns **null***object*, 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 some order is 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 called *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.variable:ModuleName.TypeName = . . .;

ModuleName.TypeName: TYPE = . . . ;

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

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

ModuleName.Error[error:ModuleName.ErrorType];

ModuleName.ErrorType: TYPE = {. . .}
```

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*; that is, 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; for example, 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; that is, 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 notes are included to help programmers design their programs to take best advantage of the Pilot facilities. Ignoring the 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, cautions 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 and later another client program requests Pilot to create a new one of the same type, then 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 **null***object* in the preceding section. A possible instance of a **null***object* 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. 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, and so forth 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. Chapter 2 also describes the run-time support needed to execute Mesa programs. The 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.

Chapter 3 introduces the general concept of a *stream*. 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 exercises 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*.

Chapter 10 describes online diagnostics for communication and I/O devices.

Appendices provide supplementary information, including performance criteria, file type management, Pilot interrupt key, UtilityPilot considerations, multi-national considerations, references, and a NetworkBinding example. Appendix H provides information about the TCP/IP interfaces (not Pilot-related).

2. Environment

2

Environment

Pilot programmers have available to them the constants, types, and procedures which describe the system elements and make available, at the client level, certain features of the abstract machine. This chapter describes these constants, types, and procedures and contains the basic levels of the system.

2.1 Processor environment

```
Environment: DEFINITIONS . . . ;
```

This section defines all of the basic constants describing the processor and peripherals. Section 2.1.1 describes the processor; section 2.1.2 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;
```

The maximum 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..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]. PageCount and PageOffset (below) are similar.

```
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;
```

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. BitBlt is the principal user.

```
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, then 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 . . . ;

DeviceTypesExtras: DEFINITIONS . . . ;

DeviceTypesExtraExtras: DEFINITIONS . . . ;

DeviceTypesExtras3: DEFINITIONS . . . ;

DeviceTypesExtras4: DEFINITIONS . . . ;

DeviceTypesExtras5: DEFINITIONS . . . ;
```

Definitions are provided for devices and classes of devices attached to the system element. These constants are defined in the interfaces **Device**, **DeviceTypes**, and **DeviceTypesExtras**. Definitions in the following **Device** interface 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);

DeviceTypesExtras.Floppy:TYPE = [17..24);

DeviceTypesExtras4.ExtendedFloppy: TYPE = CARDINAL [17..64);

DeviceTypesExtras4.FloppyTape: TYPE = ExtendedFloppy [50..64);

DeviceTypesExtras4.SCSIDisk: TYPE = Device..PilotDisk [896..1024);

DeviceTypesExtras4.OpticalDevice: TYPE = CARDINAL [1024..2048);

DeviceTypesExtras4.SingleBox: TYPE = DeviceTypesExtras4.OpticalDevice [1024..1040);

DeviceTypesExtras4.JukeBox: TYPE = DeviceTypesExtras4.OpticalDevice [1040..1048);

DeviceTypesExtras4.SCSITape: TYPE = CARDINAL [2048..2560);

DeviceTypesExtras4.SCSIProcessor: TYPE = CARDINAL [2560..2816);

DeviceTypesExtras4.SCSIReadOnly: TYPE = CARDINAL [2816..3072);
```

All Ethernet type devices have a value in the range defined by **Ethernet**.

All devices capable of containing a Pilot physical volume are in the range defined by **PilotDisk**.

All floppy drives and floppy tape drives have values defined in the range **ExtendedFloppy**. A specific subrange for floppies only is defined by **Floppy**; a specific subrange for floppy tapes is defined in **FloppyTape**.

High capability disks have their own subrange in **PilotDisk**, called **SCSIDisk**.

All optical devices have a value in the range defined by **OpticalDevice**. Subranges, defined by **SingleBox** and **JukeBox**, in the **OpticalDevice** category exist for single box and jukebox devices.

High capability tapes, processors, and read-only devices have values in the ranges defined by **SCSITape**, **SCSIProcessor**, and **SCSIReadOnly**, respectively.

Device types provide a means of classifying the different devices attachable to the system element.

Device types for Ethernet devices are listed below. The italicized column on the right indicates the specified device.

DeviceTypes.anyEthernet: Device.Type = ...;	*An Ethernet of unspecified type*
DeviceTypes.ethernet: Device.Type = ...;	*10 MB Ethernet*
DeviceTypes.ethernetOne: Device.Type = ...;	*3 MB Ethernet*

The following specific device types are assigned to Pilot disks. The italicized column on the right indicates the specified device.

DeviceTypes.anyPilotDisk: Device.Type = ...;	*A Pilot disk of unspecified type*
DeviceTypes.sa800: Device.Type = ...;	*Unspecified disk of the Shugart Associates SA800 family*
DeviceTypes.sa1000: Device.Type = ...;	*Unspecified disk of the Shugart Associates SA1000 family*
DeviceTypes.sa1004: Device.Type = ...;	*SA1004 disk*
DeviceTypes.sa4000: Device.Type = ...;	*Unspecified disk of the Shugart Associates SA4000 family*
DeviceTypes.sa4008: Device.Type = ...;	*SA4008 disk*
DeviceTypes.t300: Device.Type = ...;	*Century Data Systems T-300 disk*
DeviceTypes.t80: Device.Type = ...;	*Century Data Systems T-80 disk*
DeviceTypes.cdc9730: Device.Type = ...;	*Control Data Corporation CDC-9730 disk*
DeviceTypes.q2000: Device.Type = ...;	*Unspecified disk of the Quantum 2000 family*
DeviceTypes.q2010: Device.Type = ...;	*Quantum disk 2010*
DeviceTypes.q2020: Device.Type = ...;	*Quantum disk 2020*
DeviceTypes.q2030: Device.Type = ...;	*Quantum disk 2030*
DeviceTypes.q2040: Device.Type = ...;	*Quantum disk 2040*
DeviceTypes.q2080: Device.Type = ...;	*Quantum disk 2080*

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; for example, a Shugart SA4008 disk might be reported as DeviceTypes.anyPilotDisk, DeviceTypes.sa4000, or DeviceTypes.sa4008.

Another device type included in the interface is

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

The following Device Types interfaces contain various floppy and rigid disk drive types. In addition, DeviceTypesExtras4 defines types for floppy tapes, high capability devices, and

optical devices. Floppy tapes are streaming tape drives that are being incorporated in the current environment to appear much like floppy devices.

In the following list, the italicized column on the right indicates the device specified.

DeviceTypesExtras.anyFloppy: Device.Type = ...;	*a floppy drive of unspecified type*
DeviceTypesExtras.sa850: Device.Type = ...;	*Shugart SA850 floppy drive*
DeviceTypesExtras.sa455: Device.Type = ...;	*Shugart SA455 floppy drive*
DeviceTypesExtras.sa456: Device.Type = ...;	*Shugart SA456 floppy drive*
DeviceTypesExtraExtras.m2235: Device.Type = ...;	*Fujitsu 26 MB rigid disk drive*
DeviceTypesExtraExtras.m2242: Device.Type = ...;	*Fujitsu 50 MB rigid disk drive*
DeviceTypesExtraExtras.m2243: Device.Type = ...;	*Fujitsu 80 MB rigid disk drive*
DeviceTypesExtras3.sa475: Device.Type = ...;	*Shugart SA475 1.2Mb*
DeviceTypesExtras3.st212: Device.Type = ...;	*Seagate 10 Mb rigid disk*
DeviceTypesExtras3.st4026: Device.Type = ...;	*Seagate 20 Mb rigid disk*
DeviceTypesExtras3.tm702: Device.Type = ...;	*Tandon 20 Mb rigid disk*
DeviceTypesExtras3.tm703: Device.Type = ...;	*Tandon 40 Mb rigid disk*
DeviceTypesExtras3.mc1303: Device.Type = ...;	*Micropolis 80 Mb rigid disk*
DeviceTypesExtras3.mc1325: Device.Type = ...;	*Micropolis 80 Mb rigid disk*
DeviceTypesExtras3.q530: Device.Type = ...;	*Quantum 37 Mb rigid disk*
DeviceTypesExtras3.q540: Device.Type = ...;	*Quantum 40Mb rigid disk*
DeviceTypesExtras4.anyFloppyTape: Device.Type = ...;	*Floppy tape drive of unspecified type*
DeviceTypesExtras4.fad5000: Device.Type = ...;	*Wangtek floppy tape drive*
DeviceTypesExtras4.mr322: Device.Type = ...;	*Mitsubishi 26 MB rigid disk*
DeviceTypesExtras4.mr535: Device.Type = ...;	*Mitsubishi 51 MB rigid disk*
DeviceTypesExtras4.mk56fbx:Device.Type = ...;	*Toshiba 80 MB*
DeviceTypesExtras4.mk56fb: Device.Type = ...;	*Toshiba 86 MB rigid disk*
DeviceTypesExtras4.d3126: Device.Type = ...;	*NEC 26 MB rigid disk*
DeviceTypesExtras4.d5146h: Device.Type = ...;	*NEC 51 MB rigid disk*
DeviceTypesExtras4.m2243asx:Device.Type = ...;	*Fujitsu 80 MB*
DeviceTypesExtras4.m2243tx:Device.Type = ...;	*Fujitsu 80 MB*
DeviceTypesExtras4.m2243t:Device.Type = ...;	*Fujitsu 80 MB*
DeviceTypesExtras4.m2243as: Device.Type = ...;	*Fujitsu 86 MB rigid disk*
DeviceTypesExtras4.m2243x4: Device.Type = ...;	*344 MB rigid disk*
DeviceTypesExtras4.tm702x:Device.Type = ...;	*Tandon 20 MB*
DeviceTypesExtras4.tm703x:Device.Type =...;	*Tandon 2 0MB*
DeviceTypesExtras4.q530x:Device.Type = ...;	*Quantum 20 MB*
DeviceTypesExtras4.st4026x:Device.Type = ...;	*Seagate 20 MB*
DeviceTypesExtras4.st225x:Device.Type = ...;	*Seagate 20 MB*

```
DeviceTypesExtras4.st251x:Device.Type = ...;
DeviceTypesExtras4.st4051x:Device.Type = ...;
DeviceTypesExtras4.m2225ad:Device.Type = ...;
DeviceTypesExtras4.m2227d:Device.Type = ...;
DeviceTypesExtras4.m2227dx:Device.Type = ...;
DeviceTypesExtras4.d5146hx:Device.Type = ...;
DeviceTypesExtras4.st4096:Device.Type = ...;
DeviceTypesExtras4.st225:Device.Type = ...;
DeviceTypesExtras4.st4051:Device.Type = ...;
DeviceTypesExtras4.st251:Device.Type = ...;
DeviceTypesExtras4.st213:Device.Type = ...;
DeviceTypesExtras4.Maxtor1:Device.Type = ...;
DeviceTypesExtras4.Maxtor2:Device.Type = ...;
DeviceTypesExtras4.Maxtor3:Device.Type = ...;
DeviceTypesExtras4.sms2300:Device.Type = ...;
DeviceTypesExtras4.microp1578:Device.Type = ...;
DeviceTypesExtras4.cdcWrenIV:Device.Type = ...;
DeviceTypesExtras4.priam728:Device.Type = ...;
DeviceTypesExtras4.priam738:Device.Type = ...;
DeviceTypesExtras4.max3380:Device.Type = ...;
DeviceTypesExtras4.mo85:Device.Type = ...;
DeviceTypesExtras4.m2249s:Device.Type = ...;
DeviceTypesExtras4.m2452E:Device.Type = ...;
DeviceTypesExtras4.m2451A:Device.Type = ...;
DeviceTypesExtras4.anritsu2150C:Device.Type = ...;
DeviceTypesExtras4.LD1200JB:Device.Type = ...;
DeviceTypesExtras4.LD1200JBR1:Device.Type = ...;
DeviceTypesExtras4.LD1200JBR2:Device.Type = ...;
DeviceTypesExtras4.LD1200JBR3:Device.Type = ...;
DeviceTypesExtras4.Any12InchDisk1:Device.Type = ...;
DeviceTypesExtras4.Any12InchDisk2:Device.Type = ...;
DeviceTypesExtras4.Any12InchDisk3:Device.Type = ...;
DeviceTypesExtras4.Any12InchDisk4:Device.Type = ...;
DeviceTypesExtras4.LD500:Device.Type = ...;
DeviceTypesExtras4.LD500R1:Device.Type = ...;
DeviceTypesExtras4.LD500R2:Device.Type = ...;
DeviceTypesExtras4.LD500R3:Device.Type = ...;
```

Seagate 40 MB
Seagate 40 MB
Fujitsu 20 MB
Fujitsu 40 MB
Fujitsu 40 MB
NEC 40 MB
Seagate 80 MB
Seagate 20 MB
Seagate 40 MB
Seagate 40 MB
Seagate 10 MB
Maxtor 192 MB
Maxtor drive (placeholder)
Maxtor drive (placeholder)
Siemens 2300 310 MB
Micropolis 1578 380 MB
Control Data Wren IV 30 0MB
Priam 728 28 0MB
Priam 738 380 MB
Maxtor 3380 380 MB
12" Nikon optical 7.2 GB
Fujitsu 300 MB
Fujitsu 219 MB
Fujitsu 130 MB (w/m1008)
Anritsu Open reel MT
12" OSI Single Drive
Released OSI-Drive 2 GB
Released OSI-Drive 4 GB
Released OSI-Drive WORM/ROM
12" Hitachi Single Drive
Hitachi Released Drive
Toshiba Single Drive
Toshiba Released Drive
5 1/4" OSI Single Drive
Release of the OSI-Drive
Release of the OSI-Drive
Release of the OSI-Drive

DeviceTypesExtras4.Any5InchDisk1:Device.Type = ...;	*Hitachi Single Drive*
DeviceTypesExtras4.Any5InchDisk2:Device.Type = ...;	*Hitachi Released Drive*
DeviceTypesExtras4.Any5InchDisk3:Device.Type = ...;	*Toshiba Single Drive*
DeviceTypesExtras4.Any5InchDisk4:Device.Type = ...;	*Toshiba Released Drive*
DeviceTypesExtras4.ODSR1:Device.Type = ...;	*OSI jukebox 1 12" unit*
DeviceTypesExtras4.ODSR2:Device.Type = ...;	*OSI jukebox 2 12" units*
DeviceTypesExtras4.AnyJukeBox3:Device.Type = ...;	*Jukebox with 12" units*
DeviceTypesExtras4.AnyJukeBox4:Device.Type = ...;	*Jukebox with 12" units*
DeviceTypesExtras4.AnyJukeBox5:Device.Type = ...;	*Jukebox with 5 1/4" units*
DeviceTypesExtras4.AnyJukeBox6:Device.Type = ...;	*Jukebox with 5 1/4" units*
DeviceTypesExtras4.AnyJukeBox7:Device.Type = ...;	*Jukebox with 5 1/4" units*
DeviceTypesExtras4.AnyJukeBox8:Device.Type = ...;	*Jukebox with 5 1/4" units*
DeviceTypesExtras4.daylight:Device.Type = ...;	*Daylight*
DeviceTypesExtras4.loopBack:Device.Type = ...;	*Loop Back Tool*
DeviceTypesExtras4.xm2000a:Device.Type = ...;	*Toshiba CD-ROM*
DeviceTypesExtras5.mr533:Device.Type = ...;	*Mitsubishi 26MB*
DeviceTypesExtras5.m2225d:Device.Type = ...;	*Fujitsu 26MB*

2.2 Processor interface

Pilot provides interfaces that permit access to features provided by the underlying Mesa processor which are not provided by the Mesa language, as described in this section. These interfaces define pseudo-faces; that is, 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 Bit block transfer

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.

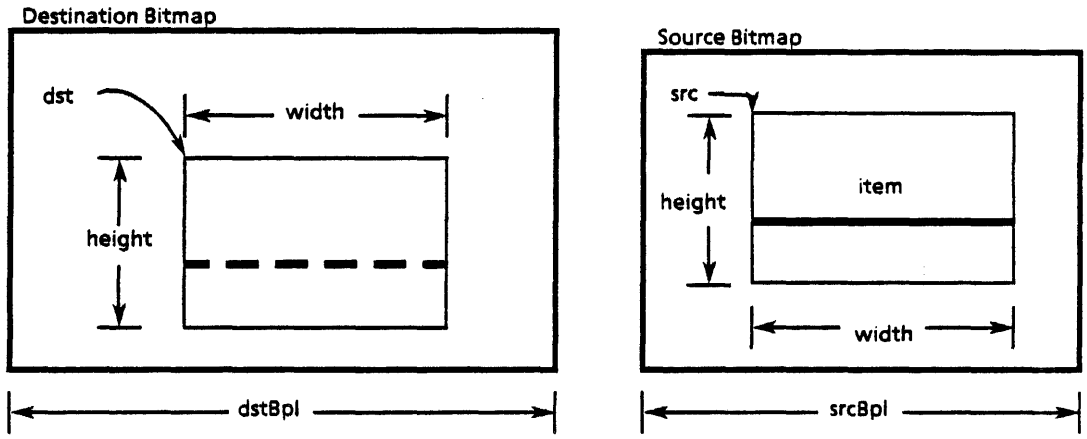


Figure 2.1. BitBlt Source and Destination

```
BitBlt.BITBLT: PROCEDURE [ptr: BBptr]
```

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.

```
BitBlt.AlignedBBTable: PROCEDURE [ip: POINTER TO BBTableSpace] RETURNS [b: BBptr] ;

BitBlt.BBTableSpace: TYPE = ARRAY [1..SIZE[BBTable] + BBTableAlignment) OF UNSPECIFIED;

BitBlt.BBTableAlignment: CARDINAL = 16;
```

AlignedBBTable ensures that the **BBTable** will be on a sixteen word boundary.

```
BitBlt.BBptr, BitBlt.BitBltTablePtr: TYPE = POINTER TO BBTable;

BitBlt.BBTable, BitBlt.BitBltTable: TYPE = MACHINE DEPENDENT RECORD [
  dst: BitAddress,
  dstBpl: INTEGER,
  src: BitAddress,
  srcDesc: SrcDesc,
  width: CARDINAL,
  height: CARDINAL,
  flags: BitBltFlags,
  reserved: UNSPECIFIED ← 0];
```

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

```
BitBlt.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 **BitBlt**s, as in scrolling.

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

If the direction is backward, then 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, then 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, then 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 that 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.

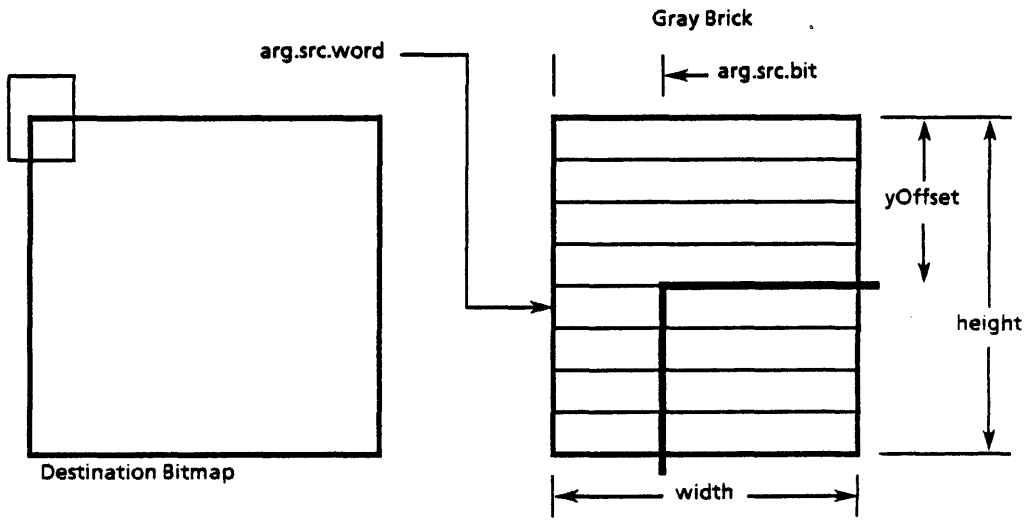


Figure 2.2 Gray Brick

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.

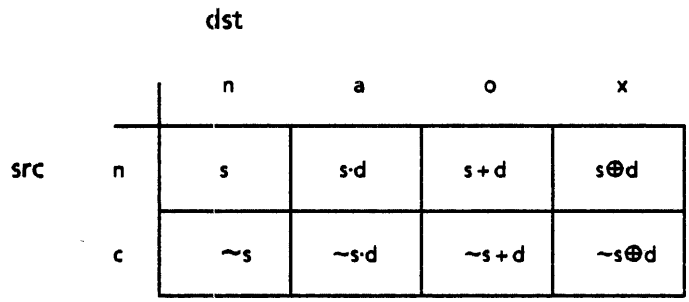


Figure 2.3 Source and Destination Functions

The **src** field has two options: the **null** selection indicates using the source rectangle as is for the destination function; the **complement** selection inverts 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. **And**ing the destination bits with the source bits leaves only those bits in common in the destination. "Painting" the destination requires **Or**ing, which leaves the union of the two sets of bits in the destination. The last function is the **XOR**, which essentially masks out the matching bits leaving the union but not the intersection of the bits in the destination rectangle.

2.2.2 Text block transfer

```
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. These functions are discussed in more detail 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, then **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 **display** function converts characters to their font representation in the destination bitmap, optionally widening or narrowing pad characters to perform line justification. The **format** function is used to calculate the number of characters that will fit on a line, given its right margin (in micas). 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. The programmer is responsibile for initializing the **bitPos** to allow for the left kerning of the first character and for supplying 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 item **Font** 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.

```
TextBltFontFormat.FontWidthsPtr: TYPE = LONG POINTER TO FontWidths;

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

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

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

```
TextBltFontFormat.FontCharPtr: TYPE = LONG POINTER TO FontChar;

TextBltFontFormat.FontChar: TYPE = ARRAY CHARACTER OF CharEntry;
```

CharEntry must be aligned on a two-word boundary.

```
TextBltFontFormat.charEntryAlignment: NATURAL = 2;

TextBltFontFormat.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, then 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.

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

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

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

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

TextBltFontFormat.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 specifies a stop character to terminate a TextBlt operation.

```
TextBltFontFormat.maxLeftKern: CARDINAL = 1;

TextBltFontFormat.maxRightKern: CARDINAL = 1;
```

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

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

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 operation

```
Checksum:DEFINITIONS...;
```

The **Checksum** 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 one's-complement add-and-left-cycle algorithm.

2.2.4 Byte block transfer

```
ByteBlt: DEFINITIONS...;
```

The only operation in this interface is **ByteBlt**, which provides a Mesa definition of a byte boundary block transfer operation. **ByteBlt** 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.StartIndexGreaterThanStopIndexPlusOne: ERROR;
```

A length of zero in either **to** or **from** is acceptable, resulting in no transfer. If a negative length (**startIndex** $\geq$ **stopIndexPlusOne**) is present in either **to** or **from**, then **ByteBlt** signals **ByteBlt.StartIndexGreaterThanStopIndexPlusOne**.

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.LongNumber 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)↑ ← (from + i)↑ ENDLOOP;
```

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

```
FOR i DECREASING IN [0..nwords) DO (to + i)↑ ← (from + i)↑ 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 **UDDivMod** and **SDDivMod** 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 following 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** is the denominator.

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

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

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

LongDiv 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 following 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 the following procedure:

```
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 the following: 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 5-word 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 perceivable 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 **UniversalID**s 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, then 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.AssignNetworkAddress**. Network addresses are guaranteed to be unique

between system restarts, but not across system restarts; that is, they are reused each time the system is restarted (see section 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

Pilot has three forms of timekeeping facilities: 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 years * 365 days + 16 leap days) * 24 hours * 60 minutes *60 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 2^{32} 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 the time-of-day clock is not correct, then the signal **SystemExtras.ClockFailed** is raised. However, if the switch **PilotSwitchesExtraExtraExtraExtras.ignoreClockFailures** is down, then the signal will not be raised.

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

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

AdjustGreenwichMeanTime 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 are kept accurately (to within a few seconds per month) by the hardware and are 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 setting 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.) An operation also allows a client to 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 Saving Time could end. Note that in any given year, Daylight Saving Time actually begins and ends at 2 A.M. on the

last Sunday not following the specified date. If Daylight Saving 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;
```

GetLocalTimeParameters 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, then 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 14.0.

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 must 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;
```

PowerOff causes the machine to be turned off. It does not return. Pilot causes all input/output activity to be suspended, purges all of its internal caches, forces out all mapped spaces to their file windows, stops all processes from further execution, and causes 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.

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

SetAutomaticPowerOn sets the internal clock of the processor to automatically turn on the system power at or after **time**. If **externalEvent** is **FALSE**, then power is turned on at the specified **time**. If **externalEvent** is **TRUE**, then 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; for example, the ringing signal of a data telephone.

If power is already on when this operation would turn it on, then the operation has 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 from which the system was booted (loaded) 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., the Installer) 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. If the booting agent provides switches other than **defaultSwitches**, then that value is used. If the boot file was constructed (by MakeBoot) to contain other than **defaultSwitches**, then that value is used. Otherwise, **defaultSwitches** is used.

Switch assignments are made by the Operating Systems group. Ranges of switches are allocated for Pilot, for the Xerox Development Environment, and for product systems.

The following interfaces provide the Pilot switches:

```
PilotSwitches: DEFINITIONS...;
PilotSwitchesExtras: DEFINITIONS...;
PilotSwitchesExtraExtras: DEFINITIONS...;
PilotSwitchesExtraExtraExtras: DEFINITIONS...;
PilotSwitchesExtraExtraExtraExtras: DEFINITIONS...;
PilotSwitchesExtras5: DEFINITIONS...;
PilotSwitchesExtras6: DEFINITIONS...;
PilotSwitchesExtras7: DEFINITIONS...;
PilotSwitchesExtras8: DEFINITIONS...;
```

Table 2.1 lists the Pilot switches, names, and meanings. Switches of special interest are described in detail following Tables 2.1 and 2.2.

Table 2.1. Pilot Switches: Value, Name, and Meaning

Value	Name	Meaning
&	PilotSwitches.hang	Hang with maintenance panel code 936 in lieu of going to the debugger.
0	PilotSwitches.break	Go to debugger as early as possible in Pilot initialization.
1	PilotSwitches.break1	Go to debugger as soon as all code is map-logged.
2	PilotSwitches.break2	Go to debugger just before calling PilotClient.Run.
3	PilotSwitches.tinyDandelionMemorySize	Simulate 192K memory size for a Dandelion with no display.
4	PilotSwitches.zeroScratchMem	Initialize scratch memory pages to zero.
5	PilotSwitches.remoteDebug	Go to the Ethernet for a debugger
6	PilotSwitches.heapOwnerChecking	Turn owner checking on for the system zones.
7	PilotSwitches.disableMapLog	Disable map logging.
8	PilotSwitches.interruptWatcher	Create a Pilot interrupt key watcher.
9	PilotSwitches.stdDandelionMemorySize	Simulate 256K memory size for a Dandelion with display.
:	PilotSwitches.breakFileMgr	(No longer supported)
;	PilotSwitches.breakVMMgr	(No longer supported)
<	PilotSwitches.noEthernetOne	Pretend that no Ethernet 1 is attached to the system element.
=	PilotSwitches.noStartCommunication	Do not initialize the Communication package at system start-up.
>	PilotSwitches.noEthernet	Pretend that no Ethernet is attached to the system element.
{	PilotSwitches.smallAnonymousBackingFile	Set the VM backing file size to 550 pages.
\| | PilotSwitches.mediumAnonymousBackingFile | Set the VM backing file size to 1200 pages. |
} | PilotSwitches.largeAnonymousBackingFile | Set the VM backing file size to 1800 pages. |
~ | PilotSwitches.heapChecking | Turn checking on for the system zones. |
? | PilotSwitches.debuggingOnUtilityPilot | (No longer supported) |
[| PilotSwitchesExtras.useTinyHeap | Create a tiny system heap, with tiny increment values. |
% | PilotSwitchesExtras.useStdHeap | Create a medium-size system heap, with medium-size increment values. (This is the default.) |
] | PilotSwitchesExtras.useLargeHeap | Create a large system heap with large increment values. |

- more -

Table 2.1. Pilot Switches: Value, Name, and Meaning - continued

Value	Name	Meaning
.	PilotSwitchesExtraExtraExtraExtras.ignoreClockFailures	Inhibit ClockFailed signal from being raised.
\200	PilotSwitchesExtraExtraExtras.continueBootingIfNoTime Server	Inhibit 937 MP hang during booting if invalid clock and no time server available.
\330.. \337	PilotSwitchesExtra8.CommunicationsSwitches	
\330		Data link layer control selector for Ethernet medium.
\331		Enable rate controlled transmit in Sequenced Packet Protocol.
\332		(Rate controlled transmit) Primary routes off local net are T1 speed links.
\333		(Rate controlled transmit) Primary routes off local net are 64k bit links.
\334		Enable Sequenced Packet Protocol parameter negotiation.
\335		Disable XNS protocol.
\336		Enable ISO 8073 Transport class negotiation.
\337		Reserved for Communications current and future use.
\350	PilotSwitchesExtra6.useHeapForSmallGlobalFrames	Allocate small global frames as nodes from a heap.
\351	PilotSwitchesExtra6.pcEmulationBank	Control allocation of real memory for PC Emulation.
\352	PilotSwitchesExtra5.tinyDoveMemSizeTinyVMTinyDisplay	Simulate 640k bytes Dove memory with 64 page VM Map, 15" display
\353	PilotSwitchesExtra5.tinyDoveMemSizeTinyVMBigDisplay	Simulate 640k bytes Dove memory with 64 page VM Map, 19" display
\354	PilotSwitchesExtra5.tinyDoveMemSizeMedVMTinyDisplay	Simulate 640k bytes Dove memory with 128 page VM Map, 15" display
\355	PilotSwitchesExtra5.tinyDoveMemSizeMedVMBigDisplay	Simulate 640k bytes Dove memory with 128 page VM Map, 19" display
\356	PilotSwitchesExtra5.tinyDoveMemSizeBigVMTinyDisplay	Simulate 640k bytes Dove memory with 256 page VM Map, 15" display
\357	PilotSwitchesExtra5.tinyDoveMemSizeBigVMBigDisplay	Simulate 640k bytes Dove memory with 256 page VM Map, 19" display
\360	PilotSwitches.germExtendedErrorReports	Display error code, global frame, and pc on boot loader errors.

- more -

Table 2.1. Pilot Switches: Value, Name, and Meaning - continued

Value	Name	Meaning
\361	PilotSwitchesExtra5.ieeeLevelZeroPacketFormat	(Not supported)
\362	PilotSwitchesExtra5.bilingualReception	(Not supported)
\363	PilotSwitchesExtra5.bilingualTransmission	(Not supported)
\364	PilotSwitchesExtra5.remoteCallDebugger	Remote call debugger.
\365	PilotSwitchesExtra7.verySmallAnonymousBackingFile	Set the VM backing file size to 325 pages.
\366	PilotSwitchesExtraExtras.saveDisplayPagesIndexA	Save 48 pages of display memory. Used in conjunction with \367, \372, and \373.
\367	PilotSwitchesExtraExtras.saveDisplayPagesIndexB	Save 64 pages of display memory. Used in conjunction with \366, \372, and \373.
\370	PilotSwitchesExtras.bypassDebuggerSubstitute	Bypass the debugger substitute by going to the real debugger.
\371	PilotSwitchesExtras.makeCodeOnePageSwapUnits	Tile code with one-page swap units.
\372	PilotSwitchesExtras.useSpecialMemory	Give display memory to Pilot for client use.
\373	PilotSwitchesExtras.useSpecialMemoryIfNoDisplay	Give display memory to Pilot for client use if no bitmap display.
\374	PilotSwitches.heapParamsFromClient	(No longer supported)
\375	PilotSwitches.fillMapLog	(No longer supported)
\376	PilotSwitches.eatGerm	Delete boot loader so the memory that it uses can be recycled.

- end -

In addition to retaining the above semantics, VM Backing File switches work in combination, as shown in Table 2.2.

Table 2.2. VM Backing File Switch Combinations

Switch			Meaning
{	|	}	(dn = down)
dn	-	-	Set the VM backing file size to 550 pages
-	dn	-	Set the VM backing file size to 1200 pages
-	-	dn	Set the VM backing file size to 1800 pages
dn	dn	-	Set the VM backing file size to 2500 pages
dn	-	dn	Set the VM backing file size to 3500 pages
-	dn	dn	Set the VM backing file size to 5000 pages
dn	dn	dn	Set the VM backing file size to 7000 pages
\365			Set the VM backing file size to 325 pages

Many Pilot boot switches are of interest only to the Pilot implementors themselves. Three such switches are listed below.

& Hang with a maintenance panel code 936 in lieu of going to the debugger.

0 Go to debugger as early as possible in Pilot initialization.

To use the 0 switch, you must have set debugger pointers in the boot file or be using an Ethernet debugger. In some cases, the boot file may have to be built with `DebugPilot.bootmesa` or `DebugUtilityPilot.bootmesa`.

1 Go to debugger as soon as all code is map-logged. ("Key Stop 1").

The debugger usually will not be able to set breakpoints in code until it has been map-logged. Also, note that from the time the boot button is pushed until shortly after key stop 1 in the system being invoked by the boot button, only an Ethernet debugger may be used; an attempt to use a local debugger will result in an MP code of 902.

The Pilot boot switches that are normally of interest to users are described below.

2 Go to debugger just before calling PilotClient.Run ("Key Stop 2").

This switch may be used to place breakpoints just before client code begins executing.

3 Simulate 192K memory size for a Dandelion with no display.

This switch is used for performance testing product software on large memory machines. See also the "9" switch.

4 Initialize scratch memory pages to zero.

This switch puts zeros in all the scratch real memory that is provided behind "dead" intervals, when they are page-faulted in or otherwise assumed to be read in. "Dead" means virtual memory mapped "dead" or had Space.Kill applied on it.

5 Go the the Ethernet for a debugger.

This switch instructs Pilot to go to the Ethernet when it needs a debugger. This instruction supersedes the presence of an installed debugger on the attached disk and/or debugger pointers which may have been set in the boot file.

6 Turn owner checking on for the system zones.

This switch causes Error[invalidNode] or [invalidOwner] to be called if the heap pointer is NIL or if the creator of heap is not the caller of the heap operation, Heap.CheckOwner.

7 Disable map logging.

In order for the debugger to access the Pilot virtual memory, it must be aware of the current mappings between virtual memory and backing storage. It does this by consulting the virtual memory map log normally produced by Pilot.

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 a debugger is 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").

Boot switch "7" will cause Pilot to stop map logging when PilotClient.Run is called (at key stop 2), thus increasing performance but seriously limiting the ability of the debugger to diagnose problems.

8 Create a Pilot interrupt key watcher.

This switch instructs Pilot to call the debugger when the LOCK and both SHIFT keys are held down and then the STOP key is pressed. The debugger will report "Pilot Emergency Interrupt." Since the Pilot process doing the job runs at the highest priority, this feature is useful for debugging Pilot itself and user input handlers. Do not attempt to Interpret Call from the debugger back into the debuggee because of the high priority level involved. The keytop name STOP is for the American Level IV keyboard; consult the keyboard mapping documentation for the equivalent key on other keyboards. Since the keys used are on the standard keyboard, a system having only a character terminal attached cannot access this feature.

9 Simulate 256K memory size for a Dandelion with display.

This switch is useful for performance testing product software on large memory machines. See also the "3" switch.

< Pretend that no Ethernet 1 is attached to the system element.
= Do not initialize the Communication package at system start-up.
> Pretend that no Ethernet is attached to the system element.

The above three switches are of interest to Pilot communication implementors.

\365 Use a tiny data space backing storage cache.
{ Use a small data space backing storage cache.
| Use a medium data space backing storage cache.
} Use a large data space backing storage cache.

Pilot allocates a cache of file space to be used for backing storage for data spaces. (The file space is allocated on the system volume.) If the size of this cache is too small for an application's needs, poor performance may result. The use of these switches allows the explicit specification of the size of this cache. If no switches are given, then Pilot uses an amount based on the size of the volume booted from. In the current version of Pilot, the switches "{", "|", and "}" can be used singly or in conjunction to allocate various sizes of backing file. See Table 2.2 for actual sizes.

^ Turn checking on for the system zones.

If this switch is set, then checking is turned on for system and system MDS heaps. This switch aids in debugging heap errors, since Heap.Error[invalidNode] will be raised when attempting to free a node from the wrong heap or to free random memory, and so forth.

[Create a tiny system heap, with tiny increment values.
% Create a medium-size system heap, with medium-size increment values.
] Create a large system heap, with large increment values.

These switches control the size of the initial system heap. They are provided for those clients that want to alter the standard setting. If "[" is set, then the system heap has an initial value of 40, increment value of 4, and largeNodeThreshold value of 128. If "%" is

set, then the system heap has an initial value of 40, increment value of 20, and largeNodeThreshold value of 260. Lastly, if "]" is set, then the system heap has an initial value of 100, increment value of 50, and largeNodeThreshold value of 260. If no switches are set, then the values for "%" are assumed.

. Inhibit ClockFailed signal from being raised.

Pilot periodically checks to see if the time of day clock is running correctly. If it appears that it is not, then the signal **SystemExtras.ClockFailed** is raised. However, if the switch "." is down, then the signal is not raised.

\200 Inhibit 937 MP hang during booting .

This switch allows Pilot clients to bypass a 937 maintenance panel hang if the clock is invalid and no time server is available. This switch can be dangerous. Any Pilot client booted with \200 should verify the validity of the time. Pilot will set the clock to a value near **gmtEpoch** if it could not reach a server and the clock was apparently invalid.

Switches \330 through \337 are communication switches. Descriptions of each follow.

\330 Data link layer control selector for Ethernet medium

If \330 is set, the Pup Protocol will use the "old style" packet type numbers which are incompatible with standard IEEE 802.2 data link encapsulation. If this switch is set, the OSI Protocol will use Ethernet encapsulation instead of IEEE 802.2 data link encapsulation to avoid conflict. If the switch is not set (i.e., up), then the Pup Protocol will use newly allocated Ethernet packet type values and the OSI Protocol will use 802.2 data link encapsulation. This switch should never be used outside of Xerox.

\331 Enable rate controlled transmit in Sequenced Packet Protocol

If \331 is set, the transmission rate of SPP data packets will be moderated for all non-local connections (at least one hop away). The intent is to avoid swamping the first router which, in turn, may allow that router to support additional active streams. Unless one of \332 or \333 is set, the assumption is made that the primary routes off the local net are 9.6k bit links.

\332 (Rate controlled transmit) Primary routes off local net are T1 speed links

This switch is meaningful only if the \331 switch is set. The \332 switch should be set if the primary routes off the local net are T1 speed links. If this switch is set, the \333 switch must not be set.

\333 (Rate controlled transmit) Primary routes off local net are 64k bit links

This switch is meaningful only if the \331 switch is set. The \333 switch should be set if the primary routes off the local net are 64k bit links. If this switch is set, the \332 switch must not be set.

\334 Enable Sequenced Packet Protocol parameter negotiation

Normally, the Sequenced Packet Protocol does not request parameter negotiation for the underlying network stream. If \334 is set, SPP will request parameter negotiation at connection establishment time.

\335 Disable XNS protocol

The XNS Protocol is automatically started during Pilot Communication initialization. If \335 is set, then the XNS Protocol will not be started.

\336 Enable ISO 8073 Transport class negotiation

Normally, class negotiation is disabled and the preferred class is the lowest acceptable class for the underlying network service. If \336 is set, the class will be negotiated and the preferred class will be the highest class implemented.

\337 Reserved for Communications current and future use.

\350 Allocate small global frames as nodes from a heap.

If \350 is set, then packaged global frames will be allocated from a heap, resulting in significant savings of MDS space for "old" style 'non-mds-relieved' modules or modules packaged with the 'old' modules. Otherwise MDS space is lost in the overhead of "page breakage" when allocating the global frames as a swapUnit (i.e., an integral number of pages) as guaranteed by the packager.

\351 Control allocation of real memory for PC Emulation.

If \351 is set, and if the type of Pilot being run is UtilityPilot, then real memory is provided for PC Emulation. If the switch is down and the type of Pilot being run is Pilot, then no real memory is provided for PC Emulation. If the switch is up, and if the type of Pilot being run is UtilityPilot, then real memory is not provided for PC Emulation. If the switch is up and the type of Pilot being run is Pilot, then real memory is provided for PC Emulation. This switch is defined in this way so that in the default case (no switches set), PC emulation is available in Pilot (assuming that the hardware supports it) and not available in UtilityPilot.

\352 Simulate 640k bytes Dove memory with 64 page VM Map, 15" display
\353 Simulate 640k bytes Dove memory with 64 page VM Map, 19" display
\354 Simulate 640k bytes Dove memory with 128 page VM Map, 15" display
\355 Simulate 640k bytes Dove memory with 128 page VM Map, 19" display
\356 Simulate 640k bytes Dove memory with 256 page VM Map, 15" display
\357 Simulate 640k bytes Dove memory with 256 page VM Map, 19" display

These switches simulate a Dove machine with 640k bytes of real memory, a 15" or 19" display, a 22-bit, 23-bit, or 24-bit virtual address space. If \352 is set, then a 64 page VM Map with a 15" display is simulated. If \353 is set, a 64 page VM Map with a 19" display is simulated. If \354 is set, then a 128 page VM Map with a 15" display is simulated. If \355 is set, a 128 page VM Map with a 19" display is simulated. If \356 is set, a 256 page VM Map with a 15" display is simulated. If \357 is set, a 256 page VM Map with a 19" display is simulated.

\360 Display error code, global frame, and pc on boot loader errors.

If this switch is set, then upon boot loader errors the maintenance panel will cycle numbers representing the error code, global frames,and pcs of the error stack.

\361 Use IEEE 802.2 Logical Link Layer protocol.
\362 Use bilingual reception.

\363 Use bilingual transmission.

If \361 is set, then Ethernet packets will be transmitted using IEEE 802.2 Logical Link Layer protocol. If \362 is set, packets will be accepted from the Ethernet in either IEEE 802.2 Logical Link Layer or Ethernet version 1.0 format. If \363 is set, the machine will transmit packets to hosts in the format that the receiver desires. \363 overrides the setting of \361.

Note: None of the switches described in the previous paragraph have been implemented.

\364 Remote call debugger.

If \364 is set, then the machine can be forced into the debugger by a suitable message from a remote machine. This facility is intended to allow forcing machines into the debugger that have no convenient means of doing so from their user interface; for example, server machines or machines whose user interface is otherwise "locked up."

\365 Set the VM backing file size to 325 pages.

Pilot allocates a cache of file space to be used for backing storage for data spaces. This switch sets the size of this cache to the minimum size. See Table 2.2 for other backing file switch settings.

\366 Save 48 pages of display memory. Used in conjunction with \367, \372, and \373.

\367 Save 64 pages of display memory. Used in conjunction with \366, \372, and \373.

When boot switches \372 or \373 are set, the amount of reserved display memory to give back to Pilot for client use is selected by switches \366 and \367. If neither \366 or \367 is down, but either \372 or \373 is down, then all display memory is given to Pilot for client use. \366 will reserve 48 pages; \367 will reserve 64 pages, and both reserve 128 pages of display bank reserved memory.

\370 Bypass the debugger substitute by going to the real debugger.

This switch setting will expect a real debugger (rather than a debugger substitute) after displaying MP codes.

\371 Tile code with one-page swap units.

This switch maps all swappable spaces that were created by `MakeBoot` (so are part of the bootfile) 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.

When booting with either \372 or \373, the entire real memory reserved for bitmap displays is made available to Pilot for client use. (On Dandelions, the size of reserved display memory is 256 pages). The only difference between using the two switches is that \373 first checks if a bitmap display is enabled before giving up the reserved memory; \372 performs no such checking. Either of these switches used in conjunction with \366, \367, or both, makes available to Pilot different ratios of reserved display memory. See explanations for \366 and \367 for more details. (Note: these switches should be used only with configurations that have no display or no bitmap display (i.e., Lear Siegler ttys).

\376 Delete boot loader so that the memory that it uses can be recycled.

If this switch is used, then the debugger will be inaccessible. In addition, the system will be unable to perform software-initiated boots of logical volumes. The only booting action available will be a boot-button boot (which may be initiated by software).

2.4 Mesa run-time support

This section describes low-level facilities used to support the execution of Mesa programs: 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, Process.Abort, and Process.Detach) 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 type is necessary since no generic type 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, then Process.InvalidProcess is 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];
```

Long timeout intervals may be set by the operation

```
Process.Seconds: TYPE = CARDINAL;

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

Caution: Because of the limited range of the process timer, the maximum timeout that may be set is about 980 seconds (16 minutes).

2.4.1.2 Timeouts

Condition variables that are initialized automatically do not time out. The timeout 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

The number of co-existing processes allowed by Pilot is limited. Attempts to fork too many processes will result in the error

```
Process.TooManyProcesses: ERROR;
```

This error 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 **FORK**ed but will never be **JOIN**ed 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, then 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, then 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, then 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, then 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.

Abort provides 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, then **EnableAborts** will have no immediate effect on it. However, if the process times out or is **NOTIFY**ed, 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, then the abort will be delayed until the process **WAIT**s 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, **Pause**s or **WAIT**s. 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 operation 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, then **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, then **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 cannot verify that it actually is a valid local frame.

```
Runtime.nullProgram: PROGRAM = NIL;
```

For backwards compatiblity, 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, then **Runtime.InvalidGlobalFrame** is raised. If the program was not created by **NEW <*program*>**, then the debugger is called.

Caution: When a program is **UnNew**ed, no processes can be 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. If the program was not created by **NEW <*program*>**, then the debugger is called.

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**, no *other* processes can be 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];

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, then **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 several instances of that interface item are being exported by already-loaded objects files, then 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, then **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), then **Runtime.ConfigError[missingCode]** is raised. If the object file exports a **TYPE** that differs from that exported by an already loaded program, then **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, then **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, then **Runtime.ConfigError[invalidConfig]** is raised. If the object file is not completely contained in the file, then **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, then 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**, no processes can be 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 a pointer to 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, then 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 **START**ed but it had been started previously (perhaps by a start trap), or if **dest** was **RESTART**ed but it had not **STOP**ped.

Note: If a program does **START** *<program>* but **program** is not valid, then **Runtime.InvalidGlobalFrame** is raised. This error occurs when **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 **START**ing any client programs calls **Run**, the first client procedure, as follows:

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

This procedure causes a start trap within the program containing **Run**, and thus starts 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 . . . ;
```

The Supervisor 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 use 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., ViewPoint and XDE), then those systems must agree on the values of the events which are common to both systems. In this case, a common definitions module, **SupervisorEventIndex**, 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, catch-all **Events** are not allowed.

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];
```

MesaEventIndexes are used by Mesa source and object files. **CommonSoftwareEventIndexes** 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 those 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 compatiblity, 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];
```

CreateSubsystem 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 not to retain or 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 on which the client depends act on it. Duplicate direct dependencies are ignored. If **implementor** is already registered as being directly or indirectly dependent on **client**, then **CyclicDependency** is raised. If **client** or **implementor** do not describe a valid subsystem, then **Supervisor.InvalidSubsystem** is raised.

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

When an event occurs, 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};
```

NotifyAllSubsystems calls the agent procedures of all subsystems. If **whichFirst** is **clients**, then a subsystem is notified only after all of its clients have been notified. If **whichFirst** is **implementors**, then 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, then **Supervisor.InvalidSubsystem** is raised.

Caution: No client of Tajo, CoPilot, or the Development Environment, versions 14.0, should call **NotifyAllSubsystems**. Doing so 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];
```

NotifyRelatedSubsystems calls the agent procedures of all subsystems which are directly or indirectly clients or implementors of **subsystem**. For **which** equal to **clients**, the operation calls all agent procedures that are direct or indirect clients of **subsystem**. For **which** equal to **implementors**, it calls all agent procedures that are the direct or indirect implementors of **subsystem**. For **whichFirst** equal to **clients**, the operation 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, then **Supervisor.InvalidSubsystem** is raised.

Caution: **NotifyRelatedSubsystems** is not implemented in Pilot 14.0.

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];
```

NotifyDirectSubsystems calls the agent procedures of all subsystems which are directly related to **subsystem**. For **which** equal to **clients**, the operation 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, then **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 signals 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 has the following 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};
```

Allocate finds, and marks as allocated, a range of **count** objects. If **willTakeSmaller** is FALSE and **count** contiguous objects cannot be found, then **Error[insufficientSpace]** is raised. If **willTakeSmaller** is TRUE, then. **Allocate** allocates the largest range of objects whose size does not exceed **count**. In this case, **Error[insufficientSpace]** is raised only if no free objects can be found. In either case, the returned range is guaranteed to be the range with the smallest **interval.first** 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**ExpandAllocation**. If the objects **[where..where + count)** are all free, then they are marked as allocated, and **extendedBy** is set to **count**. If only the objects **[where..where + countFree)** are free, where **0 <= countFree < count**, then the result depends upon the value of **willTakeSmaller**. If **willTakeSmaller** is FALSE, then **extendedBy** is returned as zero and no objects are marked allocated. If **willTakeSmaller** is TRUE, then 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 **Free**. If not all of the named objects are contained in **pool**, then **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 **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, then the specified objects are marked as free with no checking performed.

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

An **AllocationPool** may be initialized by calling **InitializePool**, which sets the initial state of all of the objects in the pool to the specified state.

Note: In any call to **Allocate**, **ExpandAllocation**, **Free**, or **InitializePool**, an **ADDRESS FAULT** may result if any part of the allocation pool is unmapped. Additionally, **ObjAlloc** provides no serialization; the client is responsible for serializing access to the database.

3. Streams

3

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, the Stream Facility

- 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.
- permits processes or subsystems to transmit arbitrary data to or from storage media in a device-independent way.
- 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.
- provides an environment for implementing simple transformations to be performed on the data as it is being transmitted.
- 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, the *stream interface* is the set of procedures and data types by which a client actually controls the transmission of a stream of information. Each operation 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, §3.1, §3.2, and §3.3 will be of interest to all clients. §3.4 will be of interest only to those clients wishing to control the physical record characteristics of a particular stream; §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 8-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; that is, 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; that is, 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, then 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.

When the client no longer wishes to transmit data to or from a stream, the stream is deleted. Deleting a stream indicates the end of an output stream and frees any resources associated with the stream. A stream may be deleted by the operation

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

For a stream used as output, the client will delete the stream when it has sent all of the data. For a stream used as input, the client will delete the stream when it no longer wishes to fetch data, either when the end of the input data is reached or earlier. For streams used both as input and output, the client will delete the streams when both of the above conditions are true. 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 Principal data transfer operations

The principal operations for transferring blocks of data are **Stream.GetBlock** and **Stream.PutBlock**. Both are inline procedures. Each 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 described in this and the next section 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 **RETRY**ed, then 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.

3.2.1.1. Block input: GetBlock

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 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

does not terminate until the entire block of bytes is filled unless a timeout occurs. 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 returns 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 following signal is generated:

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

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 if the change in **SubSequenceType** had not occurred; that is, in the same block of bytes.

If an attention is detected in the byte stream, then **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 following signal is generated:

Stream.Attention: SIGNAL [nextIndex: CARDINAL];

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 if the **Attention** had not occurred; that is, 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. For example, the NetworkStream implementation described in Chapter 6 does not provide the end-of-stream feature. However, other stream implementations abide by the end-of-stream semantics described in the following paragraphs. Clients should consult the individual stream documentation for actual end-of-stream usage.

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 transmision 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.

Note: Stream.GetBlock raises **specialSystem.Unimplemented** if the default **DefaultGetBlock** is used in conjunction with the default **DefaultGetByte** or **DefaultGetWord**. However, implementing **GetByte** and using **DefaultGetBlock** will work.

3.2.1.2. Block output: PutBlock

The principal block output operation is Stream.PutBlock.

```
Stream.PutBlock: PROCEDURE [sH: Stream.Handle, block: Stream.Block,
  endRecord: BOOLEAN ← FALSE];
```

PutBlock 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), then **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**. Consult the documentation for individual streams 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 §3.2.41 and §3.2.5).

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];
```

SendNow 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; for example, to identify control information, indicate end-of-file. 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**, **WaitForAttention** returns the byte of data associated with the attention. The client program is responsible for determining 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**, then 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**, then 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 Creation of 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 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 3.1.

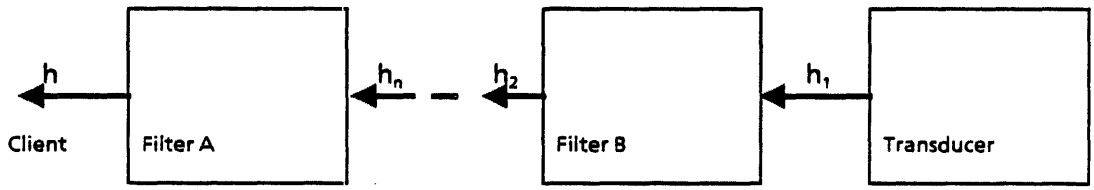


Figure 3.1

Figure 3.1 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 3.2 illustrates 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.

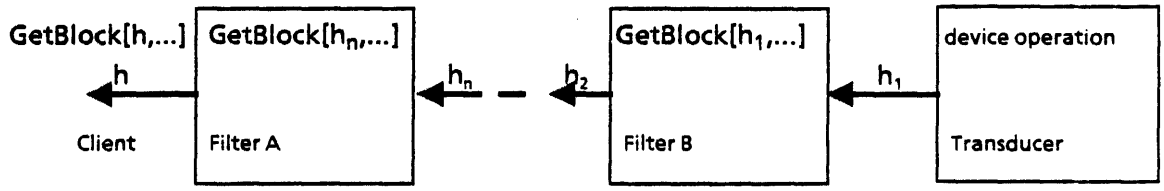


Figure 3.2

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 procedure 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 information 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 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 are discussed separately below.

On output, bytes are placed in turn into the same physical record until one of the following events occurs:

1. The **SendNow** procedure is called. The call 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), then 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 are 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, then 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 ceases with the entire contents of the physical record in the block of bytes, and the number of bytes transmitted is 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 Filter and transducer representation

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: Stream.GetPositionProcedure
  setPosition: Stream.SetPositionProcedure
  sendNow: Stream.SendNowProcedure,
  clientData: LONG POINTER,
  getSST: Stream.GetSSTProcedure,
  getTimeout: Stream.GetTimeoutProcedure,
  setTimeout: Stream.SetTimeoutProcedure];
```

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** typically contains one or more calls on **GetBlock**, **GetByte**, **GetChar**, or **GetWord** with a **Stream.Handle** parameter pointing to the next

stream component in the pipeline; that is, the parameter passed at the time this filter was created. In a transducer, the body of a **GetProcedure** typically has 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];
```

PutProcedure 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 no previous output, change in **SubSequenceType**, or attention flag is still waiting to be sent. This procedure may generate the signal **TimeOut** if necessary.

In a filter, the body of a **PutProcedure** typically contains 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; that is, the parameter passed at the time this filter was created. In a transducer, the body of a **PutProcedure** typically has 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];
```

PutByteProcedure 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];
```

PutWordProcedure 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];
```

SetSSTProcedure 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 are 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];
```

The 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: Stream.GetPositionProcedure;

Stream.GetPositionProcedure: 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];
```

The 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). The stream component manager is responsible for creating (or controlling 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**ed after the calling procedure returns to its own caller. This operation has the effect of placing a "self-destruct" mechanism in the module which takes 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** is from a procedure named in the **delete** entry of the **Stream.Object**. The component manager calls **h.delete[h]** (where **h** is a **Stream.Handle**). This procedure performs the necessary finalization, such as flushing buffers, closing files or connections, releasing storage and resources, etc. It then calls **Runtime.SelfDestruct** and finally returns to the component manager. After this return, the module representing this instance of the stream component is automatically deleted and space occupied by the component's global frame is freed.

Caution: The client must ensure that there are no outstanding references to the component module being deleted; that is, 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.

3.6 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**.

3.6.1 Errors

IndexOutOfRange: 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** raises **IndexOutOfRange**.

3.6.2 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 §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 existence of the block of virtual memory specified in **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 **Create** raises the error **IndexOutOfRange** (see §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.

4.

File Storage and Memory

4

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; that is, 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 (see §4.7) allows recording of information in a client-supplied log file.

4.1 Physical volumes

```
PhysicalVolume: DEFINITIONS...;
```

This section describes the 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, §4.1.5 with initializing a physical volume, and §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 appears 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**, then the operation 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; that is, 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 facility is an operation which examines the disk and determines whether 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 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 access 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. The following operations inform Pilot of the desired access. Upon return from these operations, 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. 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*.

If **instance** is not in the inactive state, then **Error[alreadyAsserted]** is raised. If Pilot's data structures are not in order, then **NeedsScavenging** is raised (see §4.1.6 on scavenging). **Error[notReady]** and **Error[invalidHandle]** may also be raised.

```
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.

Error[physicalVolumeUnknown] or **Error[containsOpenVolumes]** may be raised by this operation.

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. On return, unlimited access to the device is permitted by Pilot through special direct access facilities. If the drive is not currently in the inactive state, then **Error[alreadyAsserted]** is raised. **Error[invalidHandle]** may also be raised.

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, ...};
```

CreatePhysicalVolume creates a physical volume upon **instance**. 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, then 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, then **Error[alreadyAsserted]** is raised. If Pilot cannot do the necessary disk access required to create a physical volume on the disk, then **Error[badDisk]** or **Error[diskReadError]** is raised. This operation may also raise **Error[notReady]** and **Error[invalidHandle]**.

4.1.6 Scavenging operation

Scavenging is the process of returning a physical or logical volume to a consistent state. The process 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 volume version, i.e., not compatible with the volume version expected by the Pilot boot file which is running, then it must be made so before any access is allowed. Invoking **Scavenge** with **okayToConvert** = **TRUE** causes 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.

Note: **okayToConvert** is ignored in Pilot 14.0, and physical volume conversion is not supported.

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**, then **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**, then 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**, then 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**, then 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 from the Installer or disk diagnostics.

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**, then 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**, then 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 because 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

okay is returned 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: 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: Currently, 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];
```

GetNextLogicalVolume is a stateless enumerator. The enumeration begins and ends with **Volume.nullID**. This operation may raise **Error[physicalVolumeUnknown]** and **Error[noSuchLogicalVolume]**.

The physical volume that contains a given logical volume is returned by

```
PhysicalVolume.GetContainingPhysicalVolume: PROCEDURE [lvID: System.VolumeID]
  RETURNS [pvID: PhysicalVolume.ID];
```

If **lvID** is unknown to Pilot, then **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];
```

GetNext is a stateless enumerator. The enumeration begins and ends with **PhysicalVolume.nullID**. If **pvID** is not known to Pilot, then **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**, then only the characters which will fit into the string are returned.

A **layout** value of **singleLogicalVolume** indicates that one entire logical volume is on **pvID**; **multipleLogicalVolumes** indicates that more than one logical volume is on **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**, then only the first **maxNameLength** characters are used. If **newName** does not contain at least one character, then **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, then close the logical volume containing that page before invoking **MarkPageBad**. This closing is not checked by Pilot. Moreover, after the 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. See §8.3 for a description of Pilot facilities for identifying bad pages. If Pilot's table of bad pages is full, then **Error[badSpotTableFull]** is raised and **badPage** is not remembered as being bad. **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];
```

GetNextBadPage is a stateless enumerator. Enumeration begins and ends with **nullBadPage**. This operation may raise **Error[physicalVolumeUnknown]**.

4.2 Logical volumes

```
Volume: DEFINITIONS...;
```

In this section the term *volume*, where not specified as logical or physical, refers 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 the Installer (see *XDE 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 2^{23} 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: No mechanism exists to create a logical volume which spans multiple physical volumes.

The volume known as the *system volume* is 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: UtilityPilot-based systems have 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 is 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; that is, 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, causes **ReadOnly** to be raised.

```
Volume.NeedsScavenging: ERROR [volume: Volume.ID];
```

NeedsScavenging indicates that Pilot data structures on the volume are inconsistent or incorrect. This situation 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 not enough space is 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 Logical volume creation and erasure

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};
```

Create creates a new logical volume on **pvID** of type **type** and containing **size** pages. (See §4.2.6, Volume open and close operations, 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**, then **Error[tooManySubvolumes]** is raised. If **pvID** is not a valid physical volume, then **PhysicalVolume.Error[physicalVolumeUnknown]** is raised. If **size** is not enough pages to make a volume, then **Error[pageCountTooSmallForVolume]** is raised. If there is insufficient unused space on **pvID** to create the logical volume, then **PhysicalVolume.Error[insufficientSpace]** is raised. If **name** is **NIL** or its length is zero, then **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, then **Error[subvolumeHasTooManyBadPages]** is raised. Hardware errors encountered in creating the volume cause **PhysicalVolume.Error[hardwareError]** to be raised.

```
Volume.Erase: PROCEDURE [volume: Volume.ID];
```

Erase erases a logical volume, destroying its previous contents.

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.nullID**. 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 Volume open and close operations

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), then 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 causes **ReadOnly** to be raised. **Open** may also raise **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. This is true for any client whose open volume is opened for read-write by another client.

The client may close an open volume by calling

```
Volume.Close: PROCEDURE [volume: Volume.ID];
```

Close ensures 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; some attributes can be set.

```
Volume.GetAttributes: PROCEDURE [volume: Volume.ID]
  RETURNS [volumeSize, freePageCount: Volume.PageCount, readOnly: BOOLEAN];
```

GetAttributes 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; that is, 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 the ID is equal to **Volume.nullID**, then no debugger is 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 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 operation

```
Volume.GetLabelString: PROCEDURE [volume: Volume.ID, s: LONG STRING];
```

If the length of the volume label exceeds that of **s**, then 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, volume: Volume.ID←Volume.systemID]
  RETURNS [file: File.File];
```

The file previously stored for a given file type may be retrieved by calling LookUpRootFile. If the root directory has no entry in 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 . . . ;

FileTypesExtrasExtras: 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. Because 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 File addressing (internal)

Pilot files may hold up to 2^{32} bytes (2^{23} 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; -- 2^23-1

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. The file type 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 italiczed text lists the user of the file type.

Note: Consult the documentation of the appropriate system to see how the specific file types have been defined.

`FileTypes.MesaFileType: TYPE = CARDINAL[...];`	*Mesa source and object files*
`FileTypes.DCSFileType: TYPE = CARDINAL[...];`	*development common software*
`FileTypes.TestFileType: TYPE = CARDINAL [...];`	*test tools*
`FileTypes.SBSOFileType: TYPE = CARDINAL [...];`	*OPD Small Business Systems Operation*
`FileTypes.CommonSoftwareFileType: TYPE = CARDINAL [...];`	*product common software*
`FileTypes.DocProcFileType: TYPE = CARDINAL [...];`	
`FileTypes.FileServiceFileType: TYPE = CARDINAL [...];`	*file server*
`FileTypes.ServicesFileType: TYPE = CARDINAL [...];`	*print server*
`FileTypes.MesaDEFileType: TYPE = CARDINAL [...];`	*Mesa development environment*
`FileTypes.PerformanceToolFileType: TYPE = CARDINAL [...];`	*storage of binary data typically generated by performance tools*
`FileTypes.DiagnosticsFileType: TYPE = CARDINAL [...];`	*diagnostics software*
`FileTypes.CADFileType: TYPE = CARDINAL [...];`	*computer aided design software.*
`FileTypes.CedarFileType: TYPE = CARDINAL [...];`	*PARC Cedar project*
`FileTypes.VersatecFileType: TYPE = CARDINAL [...];`	*Versatec*
`FileTypesExtras.InterlispFileType: TYPE = CARDINAL[...];`	*InterLisp products*
`FileTypesExtrasExtras.BWSFileType: TYPE = CARDINAL[...];`	*Basic Workstation products*
`FileTypesExtrasExtras.FSFileType: TYPE = CARDINAL[...];`	*file services products*

The Extras and ExtrasExtras interfaces will be merged with their parent interface in future releases.

```
FileTypes.tUntypedFile: File.Type = [LAST[CARDINAL]];
```

The type **tUntypedFile** may be used as a null value, denoting the absence of a type. This use 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.tFileList: File.Type = [...];
CommonSoftwareFileTypes.tBackstopDebugger: File.Type = [...];
CommonSoftwareFileTypes.tBackstopDebuggee: File.Type = [...];
```

These file types 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 because of 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. Significant performance penalties are 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, then Error[invalidParameters] is raised. If the volume does not have enough space to contain the file, then 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, then 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 no such file is 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 attributes 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, then Pilot attempts to allocate disk space physically adjacent to the end of the file. Pilot 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 causes Error[invalidParameters] to be raised. Volume.ReadOnly will result if the volume is read-only, and Volume.InsufficientSpace will be raised is not enough free pages are 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 method 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 The scavenging operation

```
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 requirement is enforced at the time that **Volume.Open** is called. If the volume is detected as damaged by Pilot, then **Volume.NeedsScavenging** is raised. A volume is repaired using the **Scavenger** interface.

4.4.1 Volume scavenge

The purpose of the **Scavenge** operation is two-fold. First, it allows Pilot to place in order its own data structures 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.

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 volume to be scavenged is given by **volume**. If **volume** is open, then the error **Error[volumeOpen]** is raised. The log file is created on the volume **logDestination**. If **logDestination** equals **volume**, then 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 scavengee, because it may not be possible to build a log file on a damaged volume. If **repair** is **safeRepair**, then 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, then **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 volume 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**, then scavenging converts a volume from the previous version to the current one. If the volume version is incompatible but **okayToConvert** is **FALSE**, then **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 not to be converted forward.

If a previous log file for this volume exists, then 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, then **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 **FileEntrys**. 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**, then 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, then the elements of **bootFilesDeleted** corresponding to the deleted boot files will be **TRUE**. Boot files are deleted only in very unusual situations; the client must replace any deleted boot files. 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, then **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,then a **missing Problem** entry is created. If a page has an unreadable label, then 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 two or more pages claim 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, then 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 other operations; for example, **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 operation

```
Scavenger.MakeFileList: PROCEDURE [volume, logDestination: Volume.ID]
  RETURNS [logFile: File.File];
```

MakeFileList generates a **Log** for the volume **volume** without the overhead of actually scavenging the volume. If either of the specified volumes is not open, then **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; that is, 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 no log file exists, then **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, then 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 Investigation and repair of damaged pages

The damage reported in the log file may be investigated and repaired through 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. Operations that 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 writeable; 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 an 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 operation encountered errors, then the data is *not* guaranteed to be reliable. If **page** does not exist or

lies beyond the end of file, then Error[noSuchPage] is raised. If the target disk is not ready, then 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), then 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 errors were encountered while trying to rewrite the specified page. If **writeErrors** returns FALSE, then the page should be considered to be rehabilitated. Clients should first attempt to rewrite bad file pages using **RewritePage**. If the attempt 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**, then Error[noSuchPage] is raised. If no page can be allocated to replace a **missing** page, then Volume.InsufficientSpace is raised. If the target disk is not ready, then Error[diskNotReady] is raised. If the target disk reports a drive-level failure, then 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 to a new page. In addition, it frees or discards the disk page that the file currently occupies.

```
Scavenger.ReplaceBadPage: PROCEDURE [
  file: File.File, page: File.PageNumber, source: Space.PageNumber]
  RETURNS [writeErrors: BOOLEAN];
```

ReplaceBadPage allocates a new page for the specified file page. If it can rehabilitate the original page, then it marks the page free; otherwise the page is left marked in use. The page is not added to the physical volume's bad page list.

The returned value **writeErrors** indicates whether 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 writeable or an address fault or write protect fault occurs. The 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, then **readErrors** is returned TRUE and the returned data is not guaranteed to be accurate.

Caution: No validity check is made 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**, then **Error[orphanNotFound]** is returned. If the target disk is not ready, then **Error[diskNotReady]** is raised. If the target disk reports a drive-level hardware failure, then **Error[diskHardwareError]** is raised.

The client should delete an orphan page when finished with it.

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, then it is left marked in use. It is not added to the bad page list for the physical volume containing **volume**.

If **id** does not refer to a valid page on **volume**, then **Error[orphanNotFound]** is raised.

Caution: No validity check is made 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;

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

LongPointerFromPage returns a LONG POINTER to the first word of a page.

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

PageFromLongPointer returns the number of the page containing **pointer**. If **pointer** is NIL, then the value returned is undefined; no signal is raised.

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 . . . ;
```

The **Interval** is a basic concept used to describe parts of virtual memory.

```
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*, which 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 are 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 *(a)* one of its pages has been written into, or *(b)* 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, then 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 raises 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 File mapping 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, then 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, then uniform-sized swap units should be specified; this case is the default. If there is no knowledge of the proper size for the uniform swap unit size, then 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, then 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, as shown in Figure 4.1.

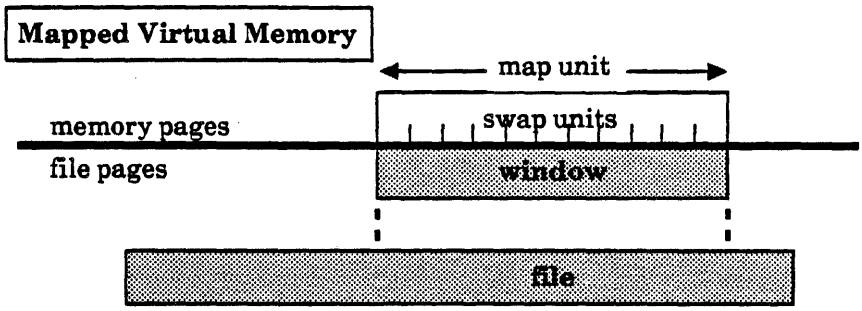


Figure 4.1. Break-down of client-specified swap unit

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 are writeable. 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, as summarized in Table 4.1.

Table 4.1. Conditions Causing **Map** To Raise Errors

Condition	ERROR
Actual window length is 0	Space.Error[noWindow]
Not enough contiguous free virtual memory	Space.InsufficientSpace[1]
window.base not in File.PageNumber range	Space.Error[invalidWindow]
window.base > file size	Space.Error[noWindow]
Required element of SwapUnits.sizes = 0	Space.Error[invalidSwapUnitSize]
Sum of SwapUnits.sizes < **window.count**	Space.Error[incompleteSwapUnits]
Volume cannot 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

[1] Space.InsufficientSpace passes back the maximum amount that could have been allocated.

The interval is mapped with the **access** given. If **access** = **readOnly**, then **life** is ignored and the interval is mapped with **life** = **alive**. If **access** = **readWrite** but **window.volume** is read-only, then 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, then the map unit is subdivided into equal-sized swap units of the indicated size. If size equals defaultSwapUnitSize or 0, then Pilot will choose an appropriate size. If size equals or exceeds the size of the map unit, then the swap unit serves no purpose; in this case specifying unitary swap units is more efficient.

If swapUnits.swapUnitType = irregular, then 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, then 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, then this swap unit will be shorter than requested. If any required element of swapUnits.sizes is 0, then Space.Error[invalidSwapUnitSize] is raised. If any required element of swapUnits.sizes is unmapped storage, then an address fault results. If the sum of the elements of swapUnits.sizes is less than the size of the map unit, then Space.Error[incompleteSwapUnits] is raised.

If swapUnits.swapUnitType = unitary, then 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 more convenient than Map for allocating temporary storage.

```
Space.ScratchMap: PROCEDURE [
  count: PageCount, usage: Space.Usage ← Space.unknownUsage]
  RETURNS [pointer: LONG POINTER];


Space.Unmap: PROCEDURE [
  pointer: LONG POINTER, returnWait: Space.ReturnWait ← wait]
  RETURNS [nil: LONG POINTER];

Space.ReturnWait: TYPE = {return, wait};

Space.ErrorType: TYPE = {..., notMapped, ...};
```

Unmap removes the association between the map unit containing pointer and the map unit's window, freeing the map unit's virtual memory for other uses. If returnWait = wait, then the operation does not return until the contents of the window reflect the contents of the interval. If returnWait = return, then 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, then the information in the window is discarded. If pointer is not contained in a map unit, then Space.Error[notMapped] is raised. If the data in the interval cannot be written to the window, then 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 Virtual memory explicit read and write operations

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.

```
Space.CopyIn: PROCEDURE [pointer: LONG POINTER, window: Space.Window]
  RETURNS [countRead: Environment.PageCount];

Space.ErrorType: TYPE = {..., readOnly, ...};
```

CopyIn reads the contents of **window** into virtual memory starting at the page that contains **pointer**. If **window.count** is 0, then **CopyIn** is a no-op. **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, then **Space.Error[readOnly]** is raised. If any portion of the virtual memory involved is unmapped, then **Space.Error[notMapped]** is raised. If the data cannot be read from the window, then **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.

```
Space.CopyOut: PROCEDURE [pointer: LONG POINTER, window: Space.Window]
  RETURNS [countWritten: Environment.PageCount];
```

CopyOut writes the current contents of virtual memory, starting at the page that contains **pointer**, out to **window**. If **window.count** is 0, then **CopyOut** is a no-op. **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, then **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, then **Space.IOError** is raised. In both 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, then **Volume.ReadOnly** is raised.

CopyIn and **CopyOut** both raise the following exceptions. If **window.base** > **File.lastPageNumber**, then **Space.Error[invalidWindow]** is raised. If the volume cannot be located, then **Volume.Unknown** is raised. **Volume.NotOnline** is raised if any part of the volume is not online. If the volume is closed, then **Volume.NotOpen** is raised. If the file does not exist on the volume, then **File.Unknown** is raised. If any of the required pages of **window** do not exist, then **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 no real memory page is available, 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.

In Pilot, swapping is caused in two ways: 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 action is known as *demand swapping*. The suspended **PROCESS** is blocked until the read operation is complete. Of course, any other ready **PROCESS**es 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; that is, 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 ensure that the current contents of an interval are accurately reflected in its backing window. This operation is useful for transactional systems.

The operations **Activate**, **Deactivate**, and **Kill** allow the client to advise Pilot so it can better manage swapping. **ForceOut** allows the client to ensure 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 **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 **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, then normal usage is **ActivateProc[LOOPHOLE[proc, PROCEDURE]]**. If **proc** is not a valid procedure, then **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.

```
Space.Kill: PROCEDURE [interval: Space.Interval];
```

Kill asserts to Pilot that the current contents of **interval** are of no further value. The operation may be used in 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 writeable 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).

```
Space.ForceOut: PROCEDURE [interval: Space.Interval];
```

ForceOut causes the window(s) of the interval to agree with the current contents of virtual memory. The operation 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 cannot be written to the given window, then **Space.IOError** is raised. If **ForceOut** causes pages to be written to backing storage, then the swap units containing those pages are marked **alive**.

Any temporary disagreement between an interval and its window should be invisible during normal operation of the system. **ForceOut** is intended 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**, then 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, then **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, then **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, then this operation may affect less virtual memory than that implied by the client-specified swap unit structure, 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 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.

```
Space.MDS: PROCEDURE RETURNS [Space.Interval];
```

MDS 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 its own MDS. Information inside the MDS can be accessed by a POINTER, which is interpreted relative to the beginning of the MDS. Information outside 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 MDSes.

Note: Although the Mesa Processor allows multiple MDSes, 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 ← Space.virtualMemory,
  base: Environment.PageOffset ← Space.defaultBase]
  RETURNS [interval: Space.Interval];

Space.defaultBase: Environment.PageOffset = . . . ;

Space.ErrorType: TYPE = {. . . , alreadyAllocated, invalidParameters, . . . };
```

Allocate allocates an interval of unmapped virtual memory within an arbitrary containing interval. If **count** is zero, then **Space.Error[invalidParameters]** is raised.

Managing an allocated interval is the responsibility of the client. Part or all of the interval may 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 **within** does not have enough contiguous unallocated pages, then **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, then Space.Error[alreadyAllocated] is raised. If the end of the interval would exceed the end of the containing interval, then 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 space is still available in the MDS.

```
Space.Deallocate: PROCEDURE [interval: Space.Interval];

Space.ErrorType: TYPE = { ..., notAllocated, stillMapped, ... };
```

Deallocate deallocates interval, making it available for other uses. interval should only contain virtual memory obtained from Space.Allocate or Space.UnmapAt.

If any portion of interval is mapped, then Space.Error[stillMapped] is raised. If any portion of interval is already deallocated, then Space.Error[alreadyDeallocated] is raised. If interval exceeds the limits of implemented virtual memory, then 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];
```

MapAt 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 unallocated pages, then Space.Error[notAllocated] is raised. If the end of the map unit would exceed the end of at, then Space.Error[invalidParameters] is raised. The operation is otherwise analogous to Space.Map (*q.v.*).

```
Space.UnmapAt: PROCEDURE [
  pointer: LONG POINTER, returnWait: Space.ReturnWait ← wait]
  RETURNS [interval: Space.Interval];
```

UnmapAt 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, then the associated interval *remains* the property of the client. If the virtual memory of the map unit was originally obtained from Map, then the client *acquires* the associated interval. The client retains this interval until it is Deallocated. The 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

```
Space.GetMapUnitAttributes: PROCEDURE [pointer: LONG POINTER]
  RETURNS [mapUnit: Space.Interval, window: Space.Window,
  usage: Space.Usage, class: Space.Class, swapUnits: Space.SwapUnitOption];
```

GetMapUnitAttributes returns the location and length of the map unit which contains **pointer**, the **window** to which it is mapped, the use of the map unit, its swapping class, and the swap unit structure.

If the map unit is mapped to a data window, then the returned **window** is **[[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**, then the returned **swapUnits.size** is the actual size of the swap units; **defaultSwapUnitSize** is never returned. If **swapUnits.swapUnitType = irregular**, then 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, then 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**, then **Space.Error[invalidParameters]** is raised.

```
Space.GetSwapUnitAttributes: PROCEDURE [pointer: LONG POINTER]
  RETURNS [swapUnit: Space.Interval, access: Space.Access, life: Space.Life];
```

GetSwapUnitAttributes 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, then the operation returns **swapUnit = Space.nullInterval**.

```
Space.PagesFromWords: PROCEDURE [wordCount: LONG CARDINAL]
  RETURNS [pageCount: Environment.PageCount] = ...;
```

PagesFromWords returns the number of pages required to contain a specified number of words.

```
Space.Pointer: PROCEDURE [pointer: LONG POINTER] RETURNS [POINTER];
```

Pointer converts a **LONG POINTER** to an equivalent **POINTER**. If the argument is not in the MDS of the calling **PROCESS**, then **Space.Error[invalidParameters]** is raised.

```
Space.PointerFromPage: PROCEDURE [page: Environment.PageNumber] RETURNS [POINTER];
```

PointerFromPage 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**, then **Space.Error[invalidParameters]** is raised.

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

LongPointerFromPage returns a **LONG POINTER** which points to the first word of the argument page.

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

PageFromLongPointer returns the page in which **pointer** lies.

4.6 Pilot memory management

Four different facilities are available for acquiring and managing storage areas:

- Global frame space, which may reside in the MDS, 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. See § 1.3.2 for conditions under which global frames reside in the MDS.
- Local frames, existing only as long as their 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. Local frames reside in the MDS.
- 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.
- A Pilot free storage package manages 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:

Heaps are more automatic, occupying system-designated (rather than client-designated) virtual memory, and expanding automatically (rather than requiring a client call).

Heaps are designed to support the Mesa language facilities for dynamic storage allocation (**UNCOUNTED ZONES**, **NEW**, **FREE**).

Some care is taken to treat large nodes (e.g., larger than 128 words) efficiently.

No mechanism exists for filing away a heap and re-creating 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 Donald 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 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 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**, then a status of **zoneTooSmall** is returned. All segments of a zone must lie entirely within a single 64K word address space; that is, all of the zone must be addressable by 16 bit relative pointers based on **base**. If such is not the case, or if the zone size is not in the range $[0..2^{16})$, then a status of **storageOutOfRange** is returned.

Caution: In this version of Pilot, zone sizes are restricted to the range $[0..2^{15})$.

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.

```
Zone.GetAttributes: PROCEDURE [zH: Zone.Handle]
  RETURNS [zoneBase: Zone.Base, threshold: Zone.BlockSize,
  checking: BOOLEAN, storage: LONG POINTER, length: Zone.BlockSize,
  next: Zone.SegmentHandle];

Zone.SegmentHandle: TYPE [1];

Zone.nullSegment: READONLY Zone.SegmentHandle;
```

GetAttributes returns the attributes of a zone. 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.

```
Zone.SetChecking: PROCEDURE [zH: Zone.Handle, checking: BOOLEAN]
  RETURNS [s: Zone.Status];

Zone.Status: TYPE = {..., invalidZone, invalidSegment, invalidNode, nodeLoop, ...};
```

SetChecking is used to enable or disable consistency checking of the zone. If **checking** is **TRUE**, then a consistency check is made that all of the nodes in the zone, and the data structures of the zone, are well-formed.

Status meanings are defined below.

invalidZone
The basic data structures of the zone identified by **zH** are malformed.

invalidSegment
Although the primary block of virtual memory in the zone is okay, one of its segments (see §4.6.1.2) is malformed.

invalidNode
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.

nodeLoop
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 following procedure:

```
Zone.AddSegment: PROCEDURE [zH: Zone.Handle, storage: LONG POINTER,
  length: Zone.BlockSize]
  RETURNS [sH: Zone.SegmentHandle, s: Zone.Status];

Zone.Status: TYPE = {..., segmentTooSmall, ...};
```

AddSegment 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, then 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 $[0..2^{16})$. 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**, then a status of **segmentTooSmall** is returned.

Caution: In this version of Pilot, segment sizes are restricted to the range $[0..2^{15})$.

All segments of a zone are linked 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 therefore 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 described in 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, then 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, then 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];
```

FreeNode 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 if the node is not marked in use, then 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];
```

SplitNode 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];
```

NodeSize 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 . . . ;
```

```
HeapExtras: DEFINITIONS . . . ;
```

The heap facility consists of the Pilot interfaces **Heap** and **HeapExtras**, together with some language features built into Mesa. The operations in **Heap** and **HeapExtras** 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.

HeapExtras includes the capability to provide backing storage for heaps on open volumes other than the system volume. If this capability is not needed, then the **Heap** interface should be used.

There are two different underlying heap implementations: **HeapImpl** (or **UnpackedHeapImpl**), which is based on the **Zone** interface; and **SOSP83HeapImpl**, which manages allocation using its own internal structures. Both implementations provide both the **Heap** and **HeapExtras** interfaces. Both implementations are discussed in this section.

4.6.2.1 Heap management

Heaps are of three types: *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. 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. When 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, then uniform heaps are more efficient, since less overhead is required for each node.

To create additional normal heaps, call Heap.Create or HeapExtras.NewCreate.

```
Heap.Create: PROC [
  initial: Environment.PageCount,
  maxSize: Environment.PageCount ← Heap.unlimitedSize,
  increment: Environment.PageCount ← 4,
  swapUnitSize: Space.SwapUnitSize ← Space.defaultSwapUnitSize,
  threshold: Heap.NWords ← Heap.minimumNodeSize,
  largeNodeThreshold: Heap.NWords ← Environment.wordsPerPage/2,
  ownerChecking: BOOLEAN ← FALSE,
  checking: BOOLEAN ← FALSE]
  RETURNS [UNCOUNTED ZONE];

HeapExtras.NewCreate: PROC [
  initial: Environment.PageCount,
  maxSize: Environment.PageCount ← Heap.unlimitedSize,
  increment: Environment.PageCount ← 4,
  swapUnitSize: Space.SwapUnitSize ← Space.defaultSwapUnitSize,
  threshold: Heap.NWords ← Heap.minimumNodeSize,
  largeNodeThreshold: Heap.NWords ← Environment.wordsPerPage/2,
  ownerChecking: BOOLEAN ← FALSE,
  checking: BOOLEAN ← FALSE,
  volumeID: System.VolumeID←Volume.systemID]
  RETURNS [UNCOUNTED ZONE];
```

To create additional uniform heaps, call Heap.CreateUniform or HeapExtras.NewCreateUniform.

```
Heap.CreateUniform: PROC [
  initial: Environment.PageCount,
  maxSize: Environment.PageCount ← Heap.unlimitedSize,
  increment: Environment.PageCount ← 4,
  swapUnitSize: Space.SwapUnitSize ← Space.defaultSwapUnitSize,
  objectSize: Heap.NWords,
  ownerChecking: BOOLEAN ← FALSE,
  checking: BOOLEAN ← FALSE]
  RETURNS [UNCOUNTED ZONE];

HeapExtras.NewCreateUniform: PROC [
  initial: Environment.PageCount,
  maxSize: Environment.PageCount ← Heap.unlimitedSize,
  increment: Environment.PageCount ← 4,
  swapUnitSize: Space.SwapUnitSize ← Space.defaultSwapUnitSize,
  objectSize: Heap.NWords,
  ownerChecking: BOOLEAN ← FALSE,
  checking: BOOLEAN ← FALSE.
  volumeID: System.VolumeID←Volume.systemID]
  RETURNS [UNCOUNTED ZONE];
```

To create additional MDS heaps, call Heap.CreateMDS or HeapExtras.NewCreateMDS.

```
Heap.CreateMDS: PROC [
  initial: Environment.PageCount,
  maxSize: Environment.PageCount ← Heap.unlimitedSize,
  increment: Environment.PageCount ← 4,
  swapUnitSize: Space.SwapUnitSize ← Space.defaultSwapUnitSize,
  threshold: Heap.NWords ← Heap.minimumNodeSize,
  largeNodeThreshold: Heap.NWords ← Environment.wordsPerPage/2,
  ownerChecking: BOOLEAN ← FALSE,
  checking: BOOLEAN ← FALSE]
  RETURNS [MDSZone];

HeapExtras.NewCreateMDS: PROC [
  initial: Environment.PageCount,
  maxSize: Environment.PageCount ← Heap.unlimitedSize,
  increment: Environment.PageCount ← 4,
  swapUnitSize: Space.SwapUnitSize ← Space.defaultSwapUnitSize,
  threshold: Heap.NWords ← Heap.minimumNodeSize,
  largeNodeThreshold: Heap.NWords ← Environment.wordsPerPage/2,
  ownerChecking: BOOLEAN ← FALSE,
  checking: BOOLEAN ← FALSE,
  volumeID: System.VolumeID←Volume.systemID]
  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, then 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 there is not enough contiguous memory for the Heap allocation request. **Volume.InsufficientSpace** may be raised if there is not enough file space on the volume backing the heap.

If a nondefault **swapUnitSize** is specified, then 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. The space for these large nodes is not included in the initial size of the heap. Large nodes are included in maxSize.

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, then owner checking is enabled (see the description in §4.6.2.3 below of the operation **CheckOwner**). The argument **checking** indicates whether 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, use of the option noticeably degrades performance.

A heap may be deleted with one of the following 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] is 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. The maximum node size is 32766. Heap.Error[invalidSize] is raised if an attempt is made to create a node with a size which is too large.

For the remainder of this section, assume that **z** and **mz** have been declared as an 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.

```
mz.NEW[T ← t]
```

is similar, except that 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 that 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.

mz.NEW[StringBody[*n*]]

is similar, except that a short STRING is returned.

4.6.2.3 Miscellaneous operations

The initial parameters and current statistics of a heap can be determined by calling the appropriate one of the following operations:

```
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];

HeapExtras.NewGetAttributes: PROC [z: UNCOUNTED ZONE]
  RETURNS [
    heapPages, maxSize, increment: Environment.PageCount,
    swapUnitSize: Space.SwapUnitSize,
    ownerChecking, checking: BOOLEAN,
    volumeID: System.VolumeID,
    attributes: Heap.Attributes];
```

```
HeapExtras.NewGetAttributesMDS: PROC [z: MDSZone]
  RETURNS [
    heapPages, largeNodePages, maxSize,
    increment: Environment.PageCount,
    swapUnitSize: Space.SwapUnitSize,
    threshold, largeNodeThreshold: Heap.NWords,
    ownerChecking, checking: BOOLEAN,
    volumeID: System.VolumeID];
```

If a heap is created through **Heap.Create**, **Heap.CreateUniform**, or **Heap.CreateMDS**, then **HeapExtras.NewGetAttributes** and **HeapExtras.NewGetAttributesMDS** will return **volumeID** = **Volume.systemID** (or **Volume.NullID** if there is no system volume).

If a heap is created through **HeapExtras.NewCreate**, **HeapExtras.NewCreateUniform**, or **HeapExtras.NewCreateMDS**, then **HeapExtras.NewGetAttributes** and **HeapExtras.NewGetAttributesMDS** will return the volume ID of the volume on which the heap was created.

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, then 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];
```

These operations may raise **Heap.Error[insufficientSpace]** or **Volume.InsufficientSpace**.

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, then it may be possible to release some of the virtual memory occupied by that heap. The following operations examine each of the spaces containing expansions to the heap **z**, releasing any containing no nodes.

```
Heap.Prune: PROCEDURE [z: UNCOUNTED ZONE];
```

```
Heap.PruneMDS: PROCEDURE [z: MDSZone];
```

If a heap was created with **ownerChecking** = **TRUE**, then the following procedures may be called to determine whether a node was allocated by the same module (global frame) as the caller of the **CheckOwner** procedure.

```
Heap.CheckOwner: PROCEDURE [p: LONG POINTER, z: UNCOUNTED ZONE];
```

```
Heap.CheckOwnerMDS: PROCEDURE [p: LONG POINTER, z: MDSZone];
```

```
Heap.ErrorType: TYPE = {..., invalidOwner, ...};
```

If the node was not allocated by the same module, then **Heap.Error[invalidOwner]** is raised.

It may be determined whether 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, ...};
```

At times it may be convenient to allocate untyped storage; for example, 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];
```

These operations may return Heap.Error[insufficientSpace], Heap.Error[invalidSize], or Volume.InsufficientSpace.

The following two procedures deallocate the specified node. If p is NIL, then 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 Facilities

```
Log: DEFINITIONS ...;
LogFile: DEFINITIONS ...;
```

The Log and LogFile 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 Log file write operations

The procedures in the Log interface are used to write into the log file, to install the log file, to start and stop logging, and to manage other control functions. The file used for the log is supplied by the client. Its properties (length, type, etc.) are not changed by the logging

package; only its content is modified. This feature allows the client to retain control of the log file in order to examine it, copy it, show it to field service personnel, and so forth.

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 formats the file starting at **firstPageNumber**. Pages preceding **firstPageNumber** are not used by the logging package. **Log.Error[illegalLog]** is raised if a current log file already exists. **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 operation 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). The 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, then 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 no current log file exists. 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. The 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 Logging control

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** resets 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, then the log starts over at the beginning (thus invalidating all previous entries). If **disable** is specified, then logging is turned off; log entries will continue to be accepted, but their contents will be discarded. If **wrap** is specified, then the log behaves 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 no current log file exists.

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 no current log file exists.

4.7.2 Log file read operation

The procedures defined in **LogFile** interface are used to examine a log file. They should not be applied to the current log file. If the current log file must be read, then the client must **Log.Close** it first.

```
LogFile.InvalidFile: ERROR;

LogFile.IllegalEnumerate: ERROR;
```

If the file supplied to any **LogFile** operation does not appear to be formatted as a log file, then the error **InvalidFile** is raised. If the file is the current log file, then the error **IllegalEnumerate** is raised.

```
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];
```

GetCount and **GetLost** can be used to determine the properties of a log file. They parallel those of the same name in the **Log** interface.

```
LogFile.GetNext: PROCEDURE
  [file: File.File, current: Log.Index, firstPageNumber: File.PageNumber ← 1]
  RETURNS [next: Log.Index];

LogFile.Inconsistent: ERROR;
```

GetNext enumerates the entries of a log file. The procedure is a stateless enumerator with a starting and ending value of **nullIndex**. If **current** appears to contain garbage, then **GetNext** raises **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 returns 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**, then **size** is the number of words in the block. If **type** is set to **string**, then **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, then the error **LogFile.DifferentType** is raised.

```
LogFile.Reset: PROCEDURE [file: File.File, firstPageNumber: File.PageNumber ← 1];
```

Reset resets 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, then **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.

5.

I/O Devices

5

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, then 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, then 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, then 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. The 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**, and so forth.

```
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 a 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 the status indicates that the channel has been suspended, then 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; for example,

```
ExampleDevice.SetNumberOfRetries: PROCEDURE [channel: ExampleDevice.ChannelHandle,
  numberOfRetries: CARDINAL];
```

Others might invoke an I/O operation which is not a data transfer; for example,

```
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; for example,

```
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 record 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 . . . ;

LevelIIVKeys: DEFINITIONS . . . ;

LevelVKeys: DEFINITIONS . . . ;

JLevelIIVKeys: DEFINITIONS . . . ;
```

The state of the keys on the keyboard is described by an array of bits. These bits are packed into an array of words maintained by Pilot but readable by the client. The following exported variable provides access to the 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 plan permits Pilot to support more than one kind of keyboard layout. The current version of Pilot has three such **DEFINITIONS** modules: **LevelIIVKeys** defines the bits for the U.S. Dandelion keyboard; **JLevelIIVKeys** defines the Japanese Dandelion keyboard; and **LevelVKeys** defines a superset of the other two **DEFINITIONS** modules (Dandelion and Japanese Dandelion) and that of the Dove keyboard. **LevelVKeys** may be used in place of any or all keyboards; a program need not know what type of keyboard it actually has.

Note: The **Keys** and **KeyStations** modules are obsolete and are included only for backward compatibility.

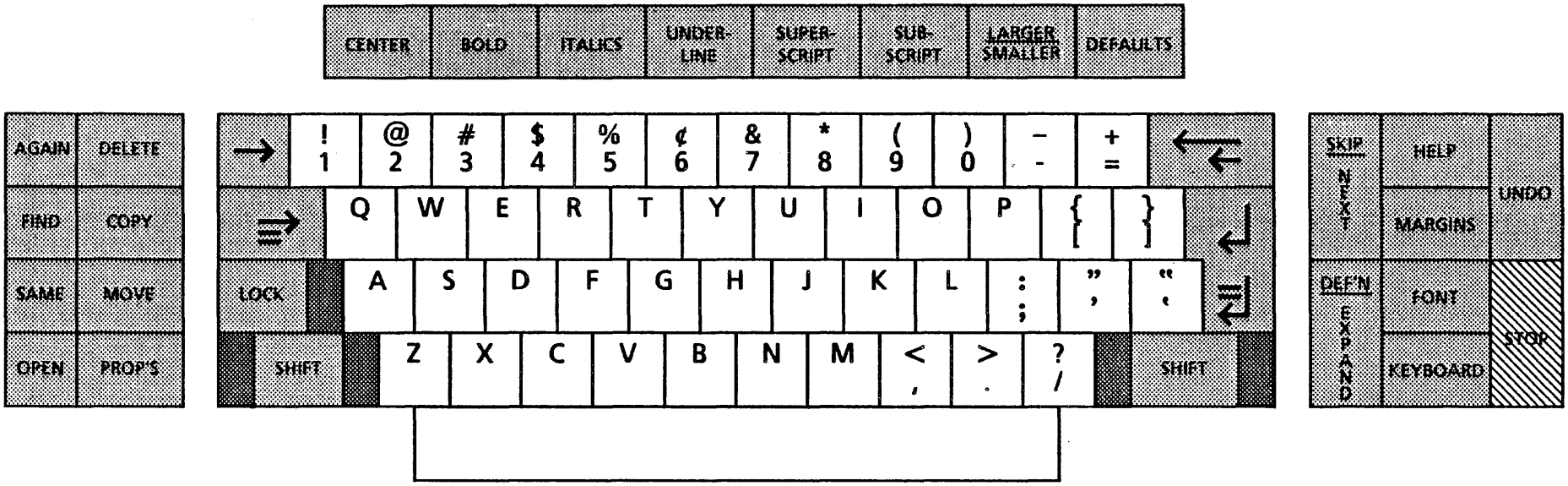
Figures 5.2a, 5.2b, and 5.2c at the end of this section show the assignments of keys on the keyboards to bits in the **UserTerminal.keyboard** array.

Table 5.1 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.

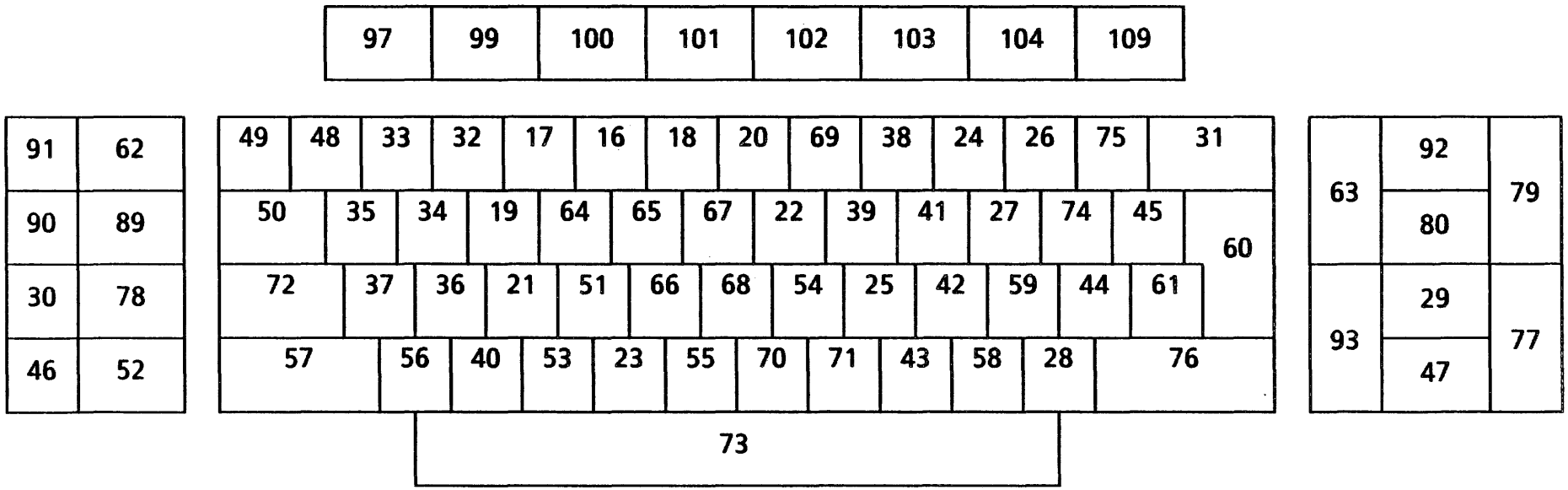
Bit:	the *n*th element in the **UserTerminal.keyboard** bit array.
Name:	the key name used to refer to this bit.

Table 5.1. Keyboard Bit Assignment

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
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

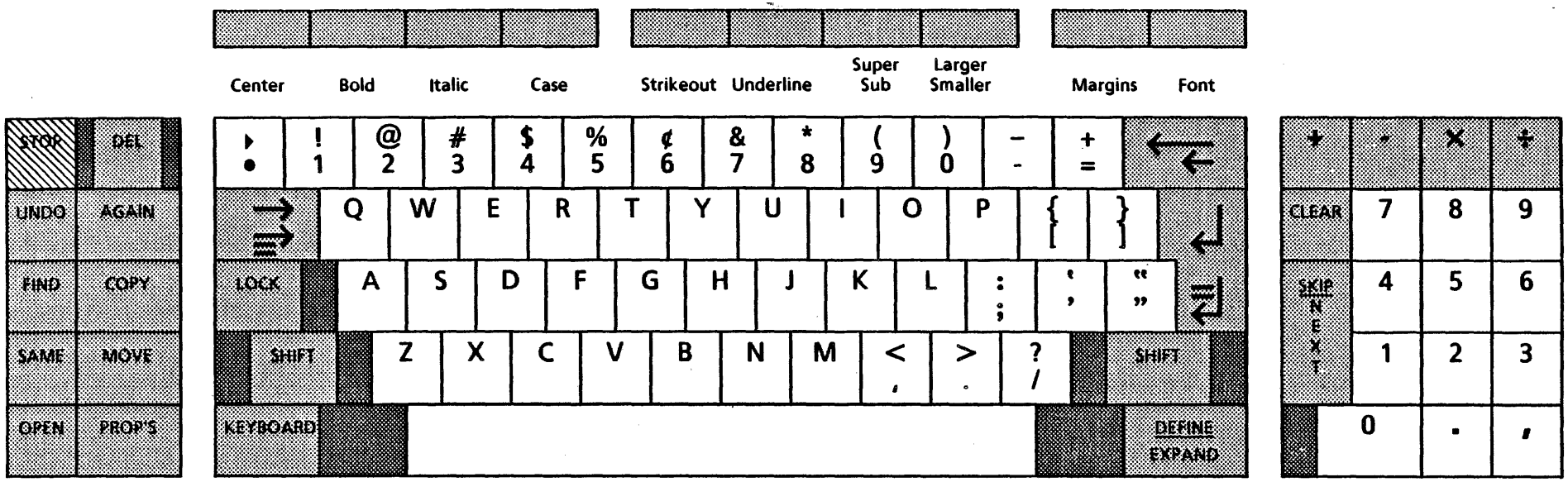


Dandelion US Key Assignments



Dandelion US Key Numbering

Fig. 5.2.a - Level IV Keyboard



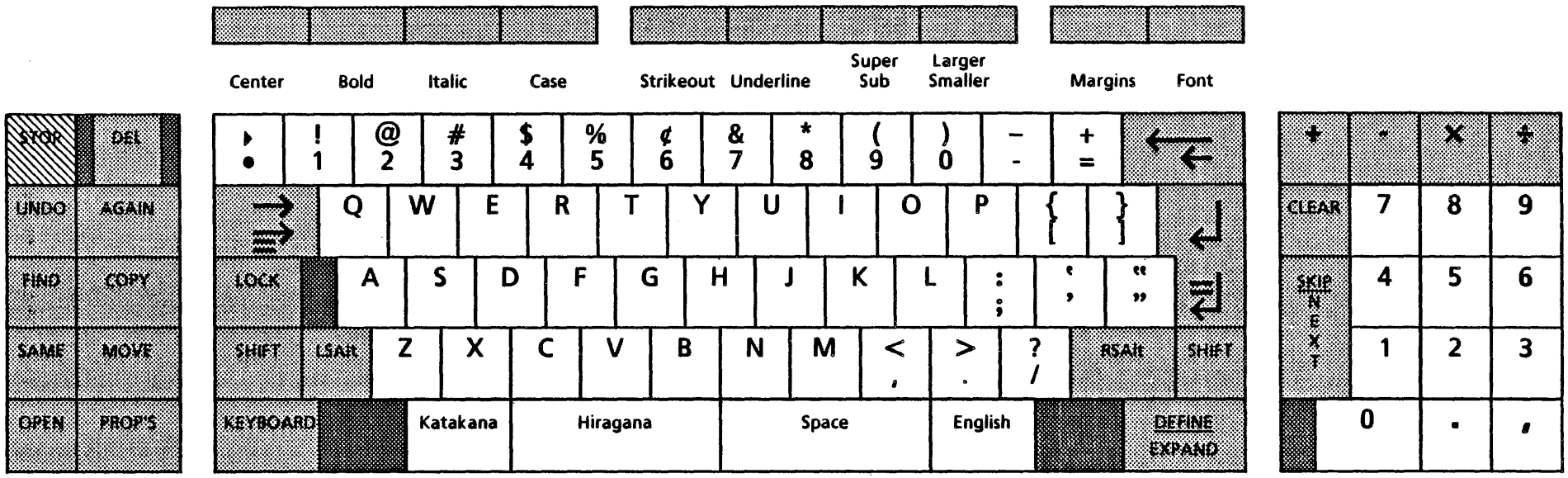
Dove US Key Assignments

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	8	9	10	11
79	91	50	35	34	19	64	65	67	22	39	41	27	74	45	60	12	81	82	83
90	89	72	37	36	21	51	66	68	54	25	42	59	7	108		63	84	85	87
30	78	57	56	40	53	23	55	70	71	43	58	28	76				94	5	6
46	52	47	73											93		98		105	106

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	1	48	33	32	17	16	18	20	69	38	24	26	75	31	8	9	10	11
79	91	50	35	34	19	64	65	67	22	39	41	27	74	45	60	12	81	82	83
90	89	72	37	36	21	51	66	68	54	25	42	59	7	108		63	84	85	87
30	78	57	107	56	40	53	23	55	70	71	43	58	28	111	76		94	5	6
46	52	47		88		110			73			86			93	98		105	106

Dove Total Key Numbering

Fig. 5.2.c - Level V Keyboard

5.3 The user terminal

```
UserTerminal: DEFINITIONS . . . ;

UserTerminalExtras: DEFINITIONS . . . ;

UserTerminalExtras2: 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. The interface assumes the configuration of the user terminal is as is given above, but does allow the client to deal with variations such as the number of keys or the size and resolution of the display. The 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. **UserTerminalExtras2** gives the type of keyboard attached to the workstation. The Extras interfaces 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.

```
UserTerminal.Coordinate: TYPE = MACHINE DEPENDENT RECORD [x, y: INTEGER];
```

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.

The state of the display is defined as

```
UserTerminal.State: TYPE = {on, off, disconnected};
```

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 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.

```
UserTerminal.WaitForScanLine: PROCEDURE [scanLine: INTEGER];
```

WaitForScanLine is provided for clients who must synchronize bitmap alteration with display refresh. Waiting for scan line zero is also a common way for a user input handler to wait between polls of the keyboard and mouse buttons.

```
UserTerminal.GetBitBltTable: PROCEDURE RETURNS [bbt: BitBlt.BBTable];
```

GetBitBltTable returns a BitBlt table with the bitmap fields filled in for the current bitmap.

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 from **screenWidth** if the display implementation has non-visible padding bits appended to each line.

WaitForScanLine and **GetBitBltTable** 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 ORed 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 (**locn.x**) and the width of the scroll window (**width**) must be multiples of **scrollXQuantum**. The vertical bit position of the scroll window (**locn.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 **locn**, **height**, and **width** are not observed, then **Error[xQuantumError]** or **Error[yQuantumError]** is raised. If a scroll window already exists, then **Error[multipleWindows]** is raised.

```
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: LONG POINTER TO UNSPECIFIED, lineCount: CARDINAL,
  increment: INTEGER];
```

```
UserTerminalExtras.ErrorType: TYPE = {..., noScrollWindow, lineCountError, ...};
```

Scroll 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, then the error **Error[noScrollWindow]** is raised. If **lineCount** is not a multiple of **scrollYQuantum**, then **Error[lineCountError]** is raised.

The scroll window may be deleted by calling

```
UserTerminalExtras.DeleteScrollWindow: PROCEDURE;
```

If **scrollingInhibitsCursor** is **TRUE**, then the cursor's reappearance may be delayed while the scan lines in the scroll window are being sorted into their proper order.

The error **Error[noScrollWindow]** is 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 DEPENDENT {
    learSiegler(0), level4(1), jLevel5(2), level5(3), eLevel5(4), other(LAST[CARDINAL])};
```

keyboardType gives the type of the keyboard attached to the system element. **learSiegler** 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; **eLevel5** is a European 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 following procedure generates 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 and floppy tape are supported in Pilot in two modes: as a Pilot floppy or floppy tape 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 and tape provides the client direct sector-level access to the floppy disk and floppy tape. 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 or floppy tape 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; **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 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. For floppy tapes, this status means that a retention is required; **otherError** will continue to be returned until the tape is retentioned.

5.4.4 Transfer operations

Transfer procedures move the specified number of sectors to or from the diskette or floppy tape.. 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**, then succeeding sectors are read and written advancing through the buffer. If it is **FALSE**, then 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 the readability of sectors.

WriteSectors and **WriteDeletedSectors** do a read-after-write to verify that the data is readable.

ReadID reads three 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**. **Error[invalidDrive]** is raised by **GetHandle** and **GetNextDrive** if they are passed an invalid **Drive**.

5.5 Floppy file system

```
Floppy: DEFINITIONS ...;

FloppyExtras: DEFINITIONS ...;

FloppyExtrasExtras: DEFINITIONS ...;

FloppyExtras3: DEFINITIONS ...;

FloppyExtras4: 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`.

The FloppyExtras4 interface allows multiple floppies or a floppyTape and one or more floppy devices to be supported on a single machine.

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; that is, 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; that is, 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 ...};
```

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 no diskette is in the drive or if 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, then **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, then 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, then 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];
```

The **Read** and **Write** operations are analogous to **Space.CopyIn** and **Space.CopyOut**; that is, 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**), then 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, ...};
```

CopyFromPilotFile and **CopyToPilotFile** are simple extensions of **Floppy.Read** and **Floppy.Write**. The operations 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];
```

ReplaceBadSector 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, then the **readError** result of this procedure is **TRUE**. If, however, Pilot believes that it has made an exact copy, then 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 14.0.

```
FloppyExtras4.nullDrive: CARDINAL = ...;

FloppyExtras4.GetNextFloppyDrive: PROCEDURE [drive: CARDINAL]
  RETURNS [nextDrive: CARDINAL];
```

GetNextFloppyDrive is a stateless enumerator which enumerates all existing floppy devices attached to the machine, beginning and ending with **FloppyExtras4.nullDrive**. If **drive** does not exist, then **Floppy.Error[noSuchDrive]** is raised.

5.5.2 Snapshotting and replication of the floppy volume

To facilitate easy snapshotting and replicating of floppies, the following procedures may be used.

```
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, then the floppy is formatted.

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 IDs on the image file is invalid. If the size of the image file is greater than the available space on the floppy, then **Error[floppySpaceTooSmall]** is raised.

Note: In Pilot 14.0, **CreateFloppyFromImage** may raise the error **Floppy.AlreadyFormatted** even though **reformatFloppy** is set to TRUE. A work-around is to ensure that the floppy is closed.

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 obtain information about the image stored in an image file.

5.5.3 Managing the floppy volume

The floppy diskette may be formatted 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];
```

The **Format** 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, then the resumable signal **AlreadyFormatted** is raised. This notice 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 formatted because of problems with either the diskette or the drive, then **Error[badDisk]** is raised.

Note: In Pilot 14.0, **Floppy.AlreadyFormatted** is raised only if the floppy is open.

```
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, . . .};
```

GetAttributes 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, independent 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 **maxFileListEntries** 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 . . .};
```

CreateFile 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 no block of free space is large enough, then **Error[insufficientSpace]** is raised. An attempt to create a zero-sized file fails with the error **Error[zeroSizeFile]**. If the file list is full, then the file is not created and **Error[fileListFull]** is raised.

CreateFile, 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];
```

DeleteFile 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 = ...;
```

GetNextFile enumerates the files on a floppy volume in the standard style of a Pilot stateless enumerator. Files are enumerated in the order of their occurrence 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];
```

SetRootFile 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, then **Error[fileNotFound]** is raised.

```
Floppy.Compact: PROCEDURE [volume: Floppy.Volume];
```

Compact 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 14.0.

```
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];
```

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. The operation 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 14.0. (However, see **FloppyExtras.NewScavenge**.)

```
FloppyExtras.Erase: PROCEDURE [
    drive: CARDINAL, maxNumberOfFileListEntries: CARDINAL,
    labelString: LONG STRING ← NIL];

FloppyExtras.ExtrasErrorType: TYPE = {..., notFormatted, ...};
```

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, then the current label is replaced; 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, then **Floppy.Error[noSuchDrive]** is raised. If the length of **labelString** exceeds **Floppy.maxCharactersInLabel**, then the label is 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, ...};

FloppyExtras4.problem:FloppyExtras4.ScavengerProblemType;

FloppyExtras4.ScavengeProblemType: TYPE =
    {allocMapInconsistent, badPageTable, bootFile, duplicateFileID, duplicateFileList,
    fileList, fileListEntry, freeSpaceConflict, ioError, none, sectorNine};
```

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**, then the floppy was, or was made, consistent. **problem** provides information to allow knowledgeable clients to understand the error which caused **NewScavenge** to fail; **none** is the value of **problem** if **okay** returned **TRUE**.

If **drive** does not describe a drive currently in the system, then **Floppy.Error[noSuchDrive]** is raised. If the diskette is write protected, then **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 14.0 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**, **FloppyExtrasExtras**, **FloppyExtras3**, and **FloppyExtras4** 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, ...};
```

CreateInitialMicrocodeFile 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 a file is already there or because some sector is bad, then **Floppy.Error[initialMicrocodeSpaceNotAvailable]** or **Floppy.Error[badSectors]** is 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, ...};
```

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. **SetBootFiles** is synchronous.

If the specified file page(s) do not exist, then **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 on Dandelions by Pilot 14.0. Clients must use MakeDLionBootFloppy tool to create bootable floppies in Pilot 14.0.

Note: In the current release of Pilot, bad sectors are not allowed in boot files. The results are undefined if this occurs.

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.CharsAvailable: 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.

lineSpeed selects the timer constant for the baud rate generator which provides the clocking for transmissions to and from the TTY Port. The default is **bps1200**.

parity selects the parity of the transmissions. The default is **none**.

stopBits is the number of stop bits. The default is **two**.

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.Quiesce**.

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: `TTYLearSiegler.bcd` or that provided by the Xerox 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.CreateTTYInstance[name: LONG STRING,
    backingStream:Stream.Handle, tty:TTY.Handle]
  RETURNS [ttyImpl, backing:Stream.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 **ttyImpl** is **NIL** and there is no default TTY implementation, then the error **NoDefaultInstance** is raised. If another **Handle** cannot be created, then **OutOfInstances** is raised.

CreateTTYInstance is the procedure interface through which an instance of the default TTY implementation is exported. **Create** calls this procedure when **ttyImpl** is **NIL**. Parameter use is at the discretion of the implementation.

ttyImpl and **backing** are returned to be used in an implementation-dependent fashion to implement the backing file for the TTY.

Note: The Lear Siegler TTY implementation ignores the parameters of **CreateTTYInstance**.

```
TTY.SetBackingSize: PROCEDURE [h: TTY.Handle, size: LONG CARDINAL];
```

SetBackingSize sets an upper limit on the number of bytes in the backing file and forces the backing file to be used in a wrap-around 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

```
TTY.LineOverflow: SIGNAL [s: LONG STRING] RETURNS [ns: LONG STRING];
```

LineOverflow 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.

```
TTY.Rubout: SIGNAL;
```

Rubout 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**, then this operation returns the **Stream.Handle** for it. If none was created, then the error **TTY.NoBackingFile** is raised.

```
TTY.CharsAvailable: 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**, then no characters are echoed; if it is **stars**, then 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, then all existing are removed before **PopAlternateInputStreams** returns.

5.7.5 String input operations

```
TTY.GetChar: PROCEDURE [h: TTY.Handle] RETURNS [c: CHARACTER];
```

GetChar 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**, then the character **c** passed to it should terminate the string. If **t** returns **ok**, then the character **c** should be appended to the string. If **t** returns **ignore**, then 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.maxlength** 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 (backword)	-	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**; the 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**, then the character **c** passed to it terminates the string. If **t** returns **ok**, then the character **c** will be appended to the string. If **t** returns **ignore**, then 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.

```
TTY.GetPassword: PROCEDURE [h: TTY.Handle, s: LONG STRING];
```

GetPassword calls **GetEditedString** with echoing set to **stars**, then restores the previous echoing mode.

5.7.6 String output operations

```
TTY.PutChar: PROCEDURE [h: TTY.Handle, c: CHARACTER];
```

PutChar 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 88 09:23:29 PDT (Wednesday)
full:	27-Jul-88 9:23:29 PDT
dateTime:	27-Jul-88 9:23:29
noSeconds:	27-Jul-88 9:23
dateOnly:	27-Jul-88

```
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.

```
TTY.RemoveCharacter, RemoveCharacters: PROCEDURE [h: TTY.Handle,
    n: CARDINAL ← 1];
```

RemoveCharacter(s) 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.

5.8 FloppyTape file system

```
FloppyTape: DEFINITIONS ...;

FloppyTapeExtras: DEFINITIONS ...;

SpecialFloppyTape: DEFINITIONS ...;
```

5.8.1 Accessing files on the tape

The floppyTape contains a collection of files. As with Pilot volumes on rigid disks, each file is a sequence of 512-byte blocks called pages. A page corresponds to a sector on the floppyTape.

```
FloppyTape.FileID: TYPE [2];

FloppyTape.nullFileID: READONLY FloppyTape.FileID;
```

Files are identified by values of the type **FileID**. These are uninterpreted 32-bit quantities assigned uniquely within a given floppyTape. **FileIDs** are not unique from one tape to another. In particular, if a tape is copied, the new tape may 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 tape; that is, it is possible for a **FileID** to be assigned to a file and later the tape **Erased** and the **FileID** to be subsequently reused.

In order to access the floppyTape, the client must specify a handle of type

```
FloppyTape.VolumeHandle: TYPE [2];

FloppyTape.nullVolumeHandle: READONLY FloppyTape.VolumeHandle;
```

A **VolumeHandle** is assigned when the floppyTape is opened (using **FloppyTape.OpenVolume**). A **VolumeHandle** becomes invalid if the floppyTape is removed, or removed and reinserted, even if the floppyTape remains the same. Values of type **VolumeHandle** are not reused within a given instantiation of Pilot; that is, from one boot to the next.

A complete specification of a floppyTape file is given by

```
FloppyTape.FileHandle: TYPE = RECORD [
  volume: FloppyTape.VolumeHandle, file: FloppyTape.FileID];
```

FileHandleFromFileID is provided to construct a **FileHandle** from a **VolumeHandle** and a **FileID**.

```
FloppyTape.FileHandleFromFileID: PROCEDURE [
  fileId: FloppyTape.FileID, volume: FloppyTape.VolumeHandle] RETURNS [file: FileHandle];
```

Errors raised by the implementation are listed below.

```
FloppyTape.IOError: ERROR [
  file: FloppyTape.FileHandle, byte: FloppyTape.ByteOffset,
  firstHole: FloppyTape.SectorNumber, howManyHoles: CARDINAL];

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

```
FloppyTape.ErrorType: TYPE = {
  badTape, badSectors, fileNotFound, hardwareError, inUse,
  invalidByteOffset, invalidVolumeHandle, insufficientSpace, needsScavenging,
  noSuchDrive, notFormatted, notReady, volumeOpen, writeInhibited};
```

For all operations, if the floppy tape software is currently busy processing a **ChangeVolume**, **Erase**, **Format**, **OpenVolume**, **ReserveDiagnosticArea**, **Scavenge**, or **Retention**, then **Error[inUse]** is raised. For all other operations, subsequent FloppyTape calls will wait.

Caution: On some pieces of hardware, it is possible to remove a tape in the middle of a read or write. Doing so may lead to undefined results.

All write operations in this interface are *asynchronous*, unlike the **Floppy** interface which is *synchronous*. That is, some **FloppyTape** write operations may return to the client before the data has actually been written to tape. If a client wishes to force a synchronous write, then either of the procedures **ForceOut** or **ForceOutBuffersOnly** must be called.

```
FloppyTape.ForceOut : PROCEDURE [volume: FloppyTape.VolumeHandle];
```

ForceOut moves all the files buffered by the file system to the tape and forces a write of other internal file structures to the tape. **ForceOut** returns only after the write operations are complete.

FloppyTape.Error[invalidVolumeHandle] is raised if the floppyTape was removed and/or reinserted, the volume was closed, or an otherwise bogus handle was specified. See **FloppyTape.WriteFile** error conditions for other errors occurring during the **ForceOut**.

```
FloppyTape.ForceOutBuffersOnly: PROCEDURE [
  volume: FloppyTape.VolumeHandle];
```

ForceOutBuffersOnly moves all the files buffered by the file system to the tape, but does NOT write other internal file structures to the tape. Therefore, it should complete faster than **ForceOut**, although the contents of the tape may be harder to recover if a **Scavenge** is required at a later time. **ForceOutBuffersOnly** returns only after the write operations are complete.

FloppyTape.Error[invalidVolumeHandle] is raised if the floppyTape was removed and/or reinserted, the volume was closed, or an otherwise bogus handle was specified. See **FloppyTape.WriteFile** error conditions for other errors occurring during the **ForceOut**.

Note: Use of the procedures **ForceOut** and **ForceOutBuffersOnly** overrides the streaming capabilities estimated for the Wangtek tape drives. For **ForceOut**, the tape will rewind to the first stream, first track, first sector, before completion of the operation. Rewinding may take up to one and a half minutes. Limited use of both procedures is recommended.

```
FloppyTape.Drive: CARDINAL;
FloppyTape.localDrive: FloppyTape.Drive = 0;
FloppyTapeExtras.nullDrive: FloppyTape.Drive = ...;
```

localDrive is usually the drive containing the floppyTape unit. However, to be sure that a floppyTape unit is referenced, clients must use the stateless enumerator **GetNextFloppyTapeDrive**.

```
FloppyTapeExtras.GetNextFloppyTapeDrive: PROCEDURE [
  drive: FloppyTape.Drive]
  RETURNS [nextDrive: FloppyTape.Drive];
```

Enumeration of floppyTape devices begins and ends with FloppyTapeExtras.nullDrive. If drive does not exist, then FloppyTape.Error[noSuchDrive] is raised. If there are no floppyTape devices, then FloppyTapeExtras.nullDrive is returned on the first enumeration as the value of nextDrive.

5.8.1.1 Opening, closing, and changing volume

In order to access the floppyTape at all, the volume must be opened.

```
NotifyClientProc: TYPE = PROCEDURE [
  drive: FloppyTape.Drive, which: {start, stop}];

FloppyTape.OpenVolume: PROCEDURE [
  drive: FloppyTape.Drive ← FloppyTape.localDrive,
  readOnly: BOOLEAN ← FALSE,
  notifyClientOfRetention: FloppyTape.NotifyClientProc,]
  RETURNS [volume: FloppyTape.VolumeHandle];
```

The operation OpenVolume opens the floppyTape volume and prepares it for all subsequent operations. The drive argument indicates which floppyTape drive is intended. If readOnly is TRUE, then the floppyTape is opened only for read access. If FALSE, the tape is opened for read and write, forcing an automatic three minute retention pass. If the client wishes to be informed of retention passes occurring during use of the tape after an OpenVolume, then the call back procedure, notifyClientOfRetention, must be provided. Once a volume is opened, no other clients can open that volume; if an attempt is made, then Error[volumeOpen] is raised.

If no floppyTape is in the drive or if for some other reason the drive is not ready, then Error[notReady] is raised. If drive specifies an unknown device, then Error[noSuchDrive] is raised. If a hardware drive failure occurs, then Error[hardwareError] is raised. If the floppyTape is not formatted, then Error[notFormatted] is raised. If the tape does not allow the proper context to be set, then Error[badTape] is raised. If there is insufficient virtual memory for Pilot's use, then Error[insufficientSpace] is raised. If the volume otherwise appears not to be well formed (Pilot data structures cannot be read or inconsistencies occur in the structures), then Error[needsScavenging] is raised. If the tape is opened read/write, but the hardware says readOnly, then Error[writeInhibited] is raised. In any of these cases, the volume is not opened.

```
FloppyTape.CloseVolume: PROCEDURE [volume: FloppyTape.VolumeHandle];
```

An open volume must be closed by calling CloseVolume. If the volume is not closed after having been opened for write, the next open may raise Error[needsScavenging]. Error[invalidVolumeHandle] is raised if the floppyTape was removed and/or reinserted or an otherwise bogus handle was specified. Closing a closed volume is a no-op.

Note: If the tape is not positioned at the beginning, then both OpenVolume and CloseVolume will rewind to the first stream, first track, first sector before completion of the operation. Rewinding may take up to one and a half minutes.

```
FloppyTape.ChangeVolume: PROCEDURE [volume: FloppyTape.VolumeHandle]
  RETURNS [newVolume: FloppyTape.VolumeHandle];

FloppyTape.ChangeTapeNow: SIGNAL[drive: FloppyTape.Drive];
```

ChangeVolume locks **volume** while a tape is removed and another tape inserted. This procedure is provided for those clients that do not wish to give up their access to the drive while tapes are being changed. A new volume handle, **newVolume**, is returned after the next tape is inserted. The tape is supposed to actually be changed when **ChangeTapeNow** is raised in the middle of the call to **ChangeVolume**. Thus, the signal should be resumed.

Possible errors are the same as for **CloseVolume** and **OpenVolume**. The new tape will have the same access that the old tape had; that is, readOnly or read/write.

5.8.1.2 Data transfer procedures

The principal operations on floppyTape files are to read from or write to them a sequence of bytes.

Note: All byte counts and byte offsets must be in increments of the page size.

```
FloppyTape.ByteCount, ByteOffset: TYPE = LONG CARDINAL;

FloppyTape.VMBuffer: TYPE = RECORD [count: ByteCount, vm: LONG POINTER];

FloppyTape.ScatteredVMSeq: TYPE = RECORD [
  SEQUENCE length: CARDINAL OF FloppyTape.VMBuffer];

FloppyTape.ScatteredVM: TYPE = LONG POINTER TO FloppyTape.ScatteredVMSeq;

FloppyTape.ReadFile: PROCEDURE [
  file: FloppyTape.FileHandle, first: FloppyTape.ByteOffset
  scatteredVM: FloppyTape.ScatteredVM];

FloppyTape.WriteFile: PROCEDURE [
  volume: FloppyTape.VolumeHandle, type:File.Type,
  scatteredVM: FloppyTape.ScatteredVM]
  RETURNS [fileId: FloppyTape.FileID];
```

ReadFile and **WriteFile** cause a sequence of pages to be copied to or from the areas in virtual memory designated by **scatteredVM**. **scatteredVM** is a sequence of long pointers to **vm** and a **count** to copy; each **vm** pointer must point to the beginning of a page. The total **counts** is the amount copied. Each chunk of virtual memory is concatenated with the next as it is copied to the floppyTape.

ByteCount limits the tape to approximately 4194 megabytes.

scatteredVM is provided during write operations for performance reasons; that is, to keep the floppyTape streaming. Otherwise, one **vm** pointer would have been sufficient. During read operations, **scatteredVM** is provided for clients requiring more than one buffer; that is, a single buffer is not sufficient to hold the data or a leader page extracted from the file data portion.

ReadFile reads from the floppyTape file beginning at the location designated by **first** and ends when the total **counts** specified by **scatteredVM** is exhausted or the file is exhausted. **ReadFile** returns to the client upon completion of the read.

If the **file** argument does not specify a known file on that floppyTape, then **Error[fileNotFound]** is raised. If **first** is not an increment of a page size or is not a page within the file, then **Error[invalidByteOffset]** is raised. If certain Pilot structures are unreadable, then **Error[needsScavenging]** is raised. If the drive suddenly does not appear ready, then **Error[notReady]** is raised. If the hardware does not seem to be functioning properly, then **Error[hardwareError]** is raised.

In order to read in an entire file, **ReadFile** may be called multiple times by adjusting the **first** parameter. The area to or from which data is copied must be mapped virtual memory, page aligned, and if necessary, writeable; otherwise, an address fault or write protect fault results.

If a read error occurs during transmission of the data (due to an unexpected error resulting in a bad sector), then the error **FloppyTape.IOError** is raised and data transmission stops. This error is raised after the data transmission occurs. The values returned with this error indicate the offending file, byte offset into the file where the first bad sector appeared, the sector number on the tape where the first bad sector appeared, and the number of holes that were caused during transmission.

Note: Bad sectors occurring during a read operation are placed in the file system's bad sector table. These sectors will appear as holes in the files in which they are contained. The sectors will not be read, but will have zeroed-out data substituted for them. If too many bad pages are encountered (i.e., if the number exceeds the limit specified internally), then **Error[badTape]** is raised.

WriteFile creates a file of type **type** and a size of the total **counts** specified by **scatteredVM**. **fileId** is the file id of the newly created file. **WriteFile** returns to the client after the data has been copied to the file system buffers, implying the data may not immediately be written to tape.

Writing beyond the physical end-of-tape raises **Error[insufficientSpace]** before any data is transferred. If an attempt to write to the floppyTape fails because the tape is write-protected or was opened read-only, then the error **Error[writeInhibited]** is raised. **Error[invalidVolumeHandle]** is raised if the floppyTape was removed and/or reinserted, the volume was closed, or an otherwise bogus handle was specified. If the drive suddenly does not appear to be ready, then **Error[notReady]** is raised. If the hardware does not seem to be functioning properly, then **Error[hardwareError]** is raised.

All bad sectors occurring during write operations are handled by the file system. A client is not informed of the bad sectors occurring. However, **Error[badTape]** is raised if the number of bad sectors exceeds the limit specified by the file system.

Note: Because of the asynchronous behavior of the write operation, errors detected during the write are reported on the following floppyTape operation. Clients must handle catching errors from the previous asynchronous tape operations.

Caution: In Pilot 14.0, it is recommended to keep a buffer of 259 unused sectors at the end of a floppyTape in order to aid in scavenging and to guarantee not overflowing the tape on an asynchronous write that encounters bad pages.

```
FloppyTape.AppendFile: PROCEDURE [
  file: FloppyTape.FileHandle, scatteredVM: FloppyTape.ScatteredVM,
  updateEndOfFile: BOOLEAN];
```

AppendFile grows **file** on the floppyTape and writes into it the contents of **scatteredVM** (see above paragraphs for usage and restrictions of **scatteredVm**). **file** will become the last file on the floppyTape. If the client wishes to further append to the same file, **updateEndOfFile** should be set to **FALSE**. If the client is done with the file, then **updateEndOfFile** should be **TRUE**. **AppendFile** returns to the client after the data has been copied to the file system buffers, implying the data may not immediately be written to tape.

If the **file** argument does not specify a known file on that floppyTape, then **Error[fileNotFound]** is raised. Writing beyond the physical end-of-tape raises **Error[insufficientSpace]** before any data is transferred. If an attempt to write to the floppyTape fails because the tape is write-protected or the volume was opened read-only, then the error **Error[writeInhibited]** is raised. **Error[badTape]** is raised if the new bad sector appears on the floppyTape and the total number of bad sectors exceeds the limit specified by the file system (256 sectors). If certain Pilot structures are unreadable, then **Error[needsScavenging]** is raised. If the drive suddenly does not appear to be ready, then **Error[notReady]** is raised. If the hardware does not seem to be functioning properly, then **Error[hardwareError]** is raised.

Note: **AppendFile** makes **file** the last file on the floppyTape. All data files following this file will be erased. The recommended approach is to **AppendFile** to the last file on the tape. However, client-level scavengers may want to intentionally erase files on a tape beyond a certain point.

Note: The **AppendFile** procedure carries a performance penalty, since the floppyTape may stop streaming. If and when **updateEndOfFile** is finally **TRUE**, the floppyTape may be positioned back to the beginning of the file to write the final size of the file and then positioned back to the end of the file.

```
FloppyTape.RewriteFile: PROCEDURE [
  file: FloppyTape.FileHandle, first: FloppyTape.ByteOffset,
  scatteredVM: FloppyTape.ScatteredVM];
```

RewriteFile allows clients the flexibility of rewriting over a portion of an already existing file with the contents specified in **scatteredVM**. **first** specifies where in **file** to start writing. If the total **scatteredVM.counts** plus **first** is greater than the size of the existing file, then **Error[insufficientSpace]** is raised. This operation may be valuable to clients maintaining a directory type structure. **RewriteFile** returns to the client upon completion of the write to tape.

If the **file** argument does not specify a known file on that floppyTape, then **Error[fileNotFound]** is raised. If **first** is not an increment of a page size or is not a page within the file, then **Error[invalidByteOffset]** is raised. If an attempt to write to the floppyTape fails because the tape is write-protected or the volume was opened with **readOnly** set to **TRUE**, then the error **Error[writeInhibited]** is raised. If a sector becomes bad during the **RewriteFile**, then **FloppyTape.IOError** is raised. (See description of **ReadFile** for an explanation of **FloppyTape.IOError**.) If certain Pilot structures are unreadable, then **Error[needsScavenging]** is raised. If the drive suddenly does not appear to be ready, then

Error[notReady] is raised. If the hardware does not seem to be functioning properly, then Error[hardwareError] is raised.

5.8.1.3 Miscellaneous facilities

```
FloppyTape.GetFileAttributes: PROCEDURE [
  file: FloppyTape.FileHandle] RETURNS [size: ByteCount, type: File.Type];
```

GetFileAttributes gets the attributes of the **file**. If the **file** argument does not specify a known file on that floppyTape, then **Error[fileNotFound]** is raised. If certain Pilot structures are unreadable, then **Error[needsScavenging]** is raised. If the drive suddenly does not appear to be ready, then **Error[notReady]** is raised. If the hardware does not seem to be functioning properly, then **Error[hardwareError]** is raised.

```
FloppyTape.GetNextFile: PROCEDURE [
  previousFile: FloppyTape.FileHandle]
  RETURNS [nextFile: FloppyTape.FileHandle];
```

GetNextFile enumerates the files on the floppyTape in the standard style of a Pilot stateless enumerator. Files are enumerated in the order in which they occur on the floppyTape. The enumeration begins by supplying the **nullFileID** and the appropiate volume and it ends with the same value. Enumeration, if started from a non-null file, does not wrap around searching all files. It returns when it reaches the logical end of a file list.

If the **previousFile** argument does not specify a known file on that floppyTape or **nullFileID**, then **Error[fileNotFound]** is raised. If certain Pilot structures are unreadable, then **Error[needsScavenging]** is raised. If the drive suddenly does not appear to be ready, then **Error[notReady]** is raised. If the hardware does not seem to be functioning properly, then **Error[hardwareError]** is raised.

Note: **GetNextFile** caches the attributes associated with the file returned. Thus, a call to **GetFileAttributes** of that file will not cause a tape access.

```
FloppyTape.SetRootFile: PROCEDURE [
  fileId: FloppyTape.FileId, volumeHandle: FloppyTape.VolumeHandle];
```

SetRootFile allows the client to record the **FileID** of a file in the volume data structures for later use; for example, the pointer to a client level directory or to some other data structure.

If an attempt to write to the floppyTape fails because the tape is write-protected or the volume was opened with **readOnly** set to **TRUE**, then **Error[writeInhibited]** is raised. If the volume handle is no longer valid, then **Error[invalidVolumeHandle]** is raised. If **SetRootFile** is never initialized, then the root file is set to **FloppyTape.nullFileID**. If certain Pilot structures are unreadable, then **Error[needsScavenging]** is raised. If the drive suddenly does not appear to be ready, then **Error[notReady]** is raised. If the hardware does not seem to be functioning properly, then **Error[hardwareError]** is raised.

5.8.2 Managing the floppyTape volume

The floppyTape may be formatted using the following operation. The volume must not be open.

```
FloppyTape.maxBytesInName: CARDINAL = 100;
FloppyTape.VolumeName: TYPE = LONG STRING;

FloppyTape.AlreadyFormatted:SIGNAL [labelString: FloppyTape.VolumeName];

FloppyTape.FeedBack: TYPE = {none, erasePass, retentionPass, formatPass, verifyPass};
FloppyTape.FeedBackPtr: TYPE = LONG POINTER TO FloppyTape.FeedBack;

FloppyTape.Format: PROCEDURE [
  drive: FloppyTape.Drive ← FloppyTape.localDrive,
  name: FloppyTape.VolumeName, clientWord: UNSPECIFIED ← 0,
  feedBack: FloppyTape.FeedBackPtr ← NIL];
```

Format erases the tape by writing special marks on every client usable sector, sets bad sectors in a bad sector table, and sets file system data structures. A retention pass automatically occurs before writing the special marks to tape, guaranteeing the integrity of the data written. A **name** is also written to the floppyTape. **clientWord** is a client-specified two words of storage written to tape. **feedBackPtr** is an optional client word of storage to which messages by the format procedure are posted. These messages indicate progress during the procedure call. It is intended to be used by clients wanting the progress reports defined by **FloppyTape.FeedBack**.

If the floppyTape is already formatted, then a resumable signal **AlreadyFormatted** is raised. This gives the client a chance to perform an **Erase** (quicker operation) instead. If the volume on the tape is open, then **Error[volumeOpen]** is raised. If the length of **name** exceeds **FloppyTape.maxCharactersInLabel**, then the label is truncated to the maximum length. If the floppyTape cannot be formatted due to problems with the floppyTape, then **Error[badTape]** is raised. If no floppyTape is in the drive or if for some other reason the drive is not ready, then the error **Error[notReady]** is raised. If **drive** specifies an unknown device, then **Error[noSuchDrive]** is raised. If an attempt to write to the floppyTape fails because the tape is write protected, then the error **Error[writeInhibited]** is raised. **Error[hardwareError]** is raised for unexpected failures accessing the floppyTape drive.

Format is a client-abortable procedure; though it cannot abort instantly, it will abort within 1-1/2 to 3 minutes.

```
FloppyTape.GetVolumeAttributes: PROCEDURE [
  volume: FloppyTape.VolumeHandle, name: FloppyTape.VolumeName]
  RETURNS [ freeSpace, usedSpace: FloppyTape.ByteCount,
  rootFile: FloppyTape.FileID, clientWord: UNSPECIFIED,
  drive: FloppyTape.Drive, numberOfBadSectors: CARDINAL];
```

GetVolumeAttributes gets relevant attributes about a floppyTape volume. The value of the label string is stored in **name** (unless a **NIL** argument was supplied). The label string is truncated if **name** is too short. Other attributes are returned in the result list. The result **freeSpace** indicates the approximate number of free bytes on the floppyTape accessible from the current logical end-of-tape position. The result **usedSpace** indicates the number of bytes on the floppyTape preceding the current logical end-of-tape position and the number of known bad spots. The sum of the two will equal the size of the tape volume. The **rootFile** is a distinguished file identified by the client.

If the root file was not set by FloppyTape.SetRootFile, then FloppyTape.nullFileId is returned. clientWord is a client-specified word of storage. drive is the drive to which the volume handle corresponds. Error[invalidVolumeHandle] is raised if the floppyTape was removed and/or reinserted, the volume was closed, or an otherwise bogus handle was specified. numberOfBadSectors is the current number of bad sectors on the tape.

Note: The maximum size of a file created is freeSpace minus one sector size for file system overhead. This calculation assumes only the bad sectors listed in the badSector table are known. Thus, any new bad spots will alter the amount of free space existing on the floppyTape.

```
FloppyTape.Erase: PROCEDURE [
  drive: FloppyTape.Drive ← FloppyTape.localDrive,
  newName: FloppyTape.VolumeName,
  clientWord: UNSPECIFIED ← 0, securityErase: BOOLEAN ← FALSE];
```

The operation Erase resets all the floppyTape file system data structures and all file and microcode pointers. If a newName is specified, it replaces the current name; otherwise the current name remains unchanged. clientWord is a client-specifed word of storage written to tape. If securityErase is TRUE, then the entire data portion of the tape will be rewritten with a well-known value. If securityErase is FALSE, then only some internal structures are reset, and the contents of the tape remain but are inaccessible. The default is FALSE.

If drive does not describe a drive currently in the system, then Error[noSuchDrive] is raised. The error Error[volumeOpen] is raised if the floppyTape volume is open. If the length of newName exceeds FloppyTape.maxCharactersInLabel, then the label is truncated to the maximum length. Error[badTape] is raised if the tape cannot be accessed. Error[notReady] is raised if no floppyTape is in the drive or if the drive is not ready. If the floppyTape cartridge is physically write protected, then Error[writeInhibited] is raised. Error[notFormatted] is raised if the floppyTape has invalid formatting information. Error[badTape] is raised if the tape context cannot be properly set or if some part of the tape that was previously writeable is now not writeable. Error[hardwareError] is raised for unexpected failures accessing the floppyTape drive.

```
FloppyTape.Scavenge: PROCEDURE [
  drive: FloppyTape.Drive ← FloppyTape.localDrive]
  RETURNS [okay: BOOLEAN];
```

Scavenge recovers the contents of a malformed floppyTape by restoring the Pilot data structures (root page) and repairing bad marker pages. The volume must not be open. The return value okay indicates whether the scavenge was successful: if okay returns TRUE, the floppyTape was, or was made, consistent.

All the errors raised during the Erase operation also apply to Scavenge.

Note: Scavenge may have difficulty (i.e., may take a long time) recovering a tape that was previously written and only erased with securityErase = FALSE.

```
FloppyTape.SectorNumber: TYPE = LONG CARDINAL;
FloppyTape.nullSectorNumber: FloppyTape.SectorNumber = 0;

FloppyTape.MarkSectorBad: PROCEDURE [
  drive: FloppyTape.Drive ← FloppyTape.localDrive,
  sector: FloppyTape.SectorNumber];
```

MarkSectorBad places the sector specified in the file system's bad sector table. The sector will then be avoided on subsequent writes and reads. This procedure need only be used by error recovery clients, such as diagnostics.

Error[volumeOpen] is raised if the volume is open. **Error[noSuchDrive]** is raised if the drive is not a floppy tape drive or does not exist. **Error[badTape]** is raised if there are too many bad pages. **Error[notReady]** is raised if the tape does not appear to be ready. **Error[writeInhibited]** is raised if the tape is physically write-protected. **Error[notFormatted]** is raised if the tape does not appear to be formatted. **Error[hardwareError]** is raised on other unexpected errors.

```
FloppyTape.GetNextBadSector: PROCEDURE [
  drive: FloppyTape.Drive ← FloppyTape.localDrive,
  sector: FloppyTape.SectorNumber]
  RETURNS [FloppyTape.SectorNumber];
```

GetNextBadSector is a stateless enumerator of the known bad spots on the tape. The same errors may be raised as for **MarkSectorBad**, except that **Error[badTape]** and **Error[writeInhibited]** will not be raised.

Note: **GetNextBadSector** will read information off the tape whenever **sector** = **nullSectorNumber** or the tape has changed state (e.g., ready to not-ready and back). Otherwise, the information is reported out of a cache.

```
FloppyTape.Retention: PROCEDURE [drive: FloppyTape.Drive];
```

Retention moves the floppyTape from one end to another, averaging about three minutes. If a floppyTape is not retentioned, soft/hard errors may occur.

Error[notReady] is raised if the floppyTape was removed and/or reinserted. **Error[volumeOpen]** is raised if the volume is open. **Error[noSuchDrive]** is raised if the drive is not a floppy tape drive or does not exist. **Error[hardwareError]** is raised if some other unexpected error occurs.

5.8.3 Booting from the tape

Several special operations are necessary to support Pilot-bootable floppyTapes. All may raise the errors raised by **MarkSectorBad** and for the same reasons.

```
SpecialFloppyTape.CreateInitialMicrocodeFile: PROCEDURE [
  volume: FloppyTape.VolumeHandle, initial: File.File,
  size: FloppyTape.ByteCount, type: File.Type,
  startingOffset: FloppyTape.ByteOffset];
```

CreateInitialMicrocodeFile creates the initial microcode file at the exact location demanded by the hardware boot facility. In particular, the page of the file indicated by **startingOffset** will appear where the hardware expects to read the first block from the floppyTape 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 floppyTapes having no bad sectors on track zero.

If it is not possible to create such a file because some sector is bad, then FloppyTape.Error[badSectors] is raised. If **startingOffset** is not an increment of a page size or is not a page within the file, then FloppyTape.Error[invalidByteOffset] is raised. If the floppyTape was removed and/or reinserted, the volume was closed, or an otherwise bogus handle was specified, then FloppyTape.Error[invalidVolumeHandle] is raised. If the floppyTape is write protected or the volume was opened readonly, then FloppyTape.Error[writeInhibited] is raised.

```
SpecialFloppyTape.BootFilePointer: TYPE = RECORD [
  file: FloppyTape.FileID, offset: FloppyTape.ByteOffset];

SpecialFloppyTape.nullBootFilePointer:SpecialFloppyTape.BootFilePointer;

SpecialFloppyTape.SetBootFiles: PROCEDURE [
  volume: FloppyTape.VolumeHandle, pilotMicrocode,,
  diagnosticMicrocode, germ, pilotBootFile: FloppyTape.BootFilePointer ←
  FloppyTape.nullBootFilePointer];

SpecialFloppyTape.GetBootFiles: PROCEDURE [volume: Floppy.VolumeHandle]
  RETURNS [initialMicrocode, pilotMicrocode, diagnosticMicrocode,
  germ, pilotBootFile: FloppyTape.BootFilePointer];
```

SetBootFiles sets the pointers to the relevant boot files in the volume data structures. These data structures are read by bootable tools, to construct an initial microcode root page at a fixed location set by initial and used by initial microcode during the booting process.

Both a **FileID** and a byte offset into the file are specified so that leader pages may be included in floppyTape boot files if desired. **SetBootFiles** sets the pointer in track zero's data structures for any of its arguments with a non-null **FileID**. Boot file pointers with **nullFileID** are cleared.

If **offset** is not an increment of a page size or is not a page within the file, then FloppyTape.Error[invalidByteOffset] is raised. FloppyTape.Error[invalidVolumeHandle] is raised if the floppyTape was removed and/or reinserted, the volume was closed, or an otherwise bogus handle was specified. If the floppyTape is write-protected or the volume was opened read-only, then FloppyTape.Error[writeInhibited] is raised.

The remaining boot files on the floppyTape, apart from the initial microcode boot file, are all read by the initial microcode file. Thus, they can be located anywhere.

Note: Pilot 14.0 initial microcode does not interpret the bad sector table. Therefore, bootfiles, germ, and microcode cannot contain bad sectors in the areas in which they are placed; otherwise, FloppyTape.Error[badSectors] is raised during the making of bootable tapes.

GetBootFiles gets the pointers to all of the boot files using track zero's volume structures. The operation also returns pointers to initial microcode. FloppyTape.Error[invalidVolumeHandle] can be raised.

Note: Initial microcode is located in a reserved location on the tape.

It is recommended that clients assign distinguished Pilot file types to boot files to allow the boot file pointers to be reset after scavenging, if necessary.

Possible alternate bootfiles may also be placed on the tape through the normal tape operations (FloppyTape.WriteFile, FloppyTape.AppendFile, and FloppyTape.RewriteFile) and use of SpecialFloppyTape.CheckBootFile.

```
SpecialFloppyTape.CheckBootFile: PROCEDURE [file: FloppyTape.FileHandle];
```

If the specified file does not exist, then **CheckBootFile** may raise the error FloppyTape.Error[fileNotFound]. If there are bad sectors in the file, then Error[badSectors] is raised (since boot files may not contain bad sectors).

Note: Clients must use FloppyTapeCommands to create bootable floppyTapes in Pilot 14.0.

```
SpecialFloppyTape.ReserveDiagnosticArea: PROCEDURE [
  drive: FloppyTape.Drive ← FloppyTape.localDrive]
  RETURNS [ableToReserve: BOOLEAN];
```

ReserveDiagnosticArea reserves and writes diagnostic data to the diagnostic data area. Reserving diagnostic area logically results in truncating the tape approximately in half. **ReserveDiagnosticArea** is a client-abortable procedure; though it cannot abort instantly, it will abort within one and a half to three minutes.

Note: In Pilot 14.0, aborting **ReserveDiagnosticArea** may take longer than one and a half to three minutes.

```
SpecialFloppyTape.GetDiskAddress :PROCEDURE [file: FloppyTape.FileHandle,
  byteOffset: FloppyTape.ByteOffset]
  RETURNS [diskAddress: FloppyChannel.DiskAddress];
```

GetDiskAddress returns the disk address of the file at the specified byte offset into that file. If the file does not exist, then Error[fileNotFound] is raised.

6. Communication

6

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* (see §6.1). Socket numbers in the range **[0..3000)** 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 = ...;

NSConstantsExtras: 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 are 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 = ...

Additional well known sockets are assigned for specific purposes and are defined in the interface **NSConstantsExtras**.

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 retransmission 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 **XNS.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.

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])};
```

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 identifier 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,
    requestType: 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;
```

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. Because overflow is possibile, 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; that is, 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 ensure 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*. Attempting to send requests or replies longer than **PacketExchange.maxBlockLength** causes 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. **ErrorReason** is defined below.

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 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 ← PacketExchange.defaultWaitTime,
    retransmissionInterval:PacketExchange.WaitTime ←
        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 raises no signals.

```
PacketExchange.CreateReplier: PROCEDURE [
    local: System.NetworkAddress, requestCount: CARDINAL ← 1,
    waitTime: PacketExchange.WaitTime ← PacketExchange.defaultWaitTime,
    retransmissionInterval:PacketExchange.WaitTime ←
        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, then 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 call 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 that PacketExchange.maxBlockLength. The reply packet has 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 returns only after a valid response has been received. When it returns, replyBlk contains nBytes of client data, and the reply received is of type replyType. This procedure may signal PacketExchange.Error or PacketExchange.Timeout. The latter may be RESUMEd causing the request to reenter the timeout interval. It is important to note the difference between RESUMEing and RETRYing. RESUMEing will not assign a new exchange identifier permitting the replier to suppress any retransmissions as duplicates if appropriate. RETRYing 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 Process.Abort. 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** are accepted.

PacketExchange.WaitForRequest returns 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**.

WaitForRequest 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 **Process.Abort**. 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 **XNS.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 asigned 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 the conversion results in an overflow, subsequent timed operations will never time out. 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; for example, 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 Network stream creation

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 Section 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];
```

CreateTransducer does 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 causes a unique address to be generated. That address is generated by calling NetworkSteam.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 does not consume that socket, but generates 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, then the create process transmits connection requests to the remote. If it is FALSE, then the create process merely listens for the connection requests. In some cases, both parties know the entire set of connection parameters, including the connection identifiers. This implies that 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, then timeout is 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; that is, 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. If there is disagreement, then transactional is assumed.

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 exists 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, which is done by calling NetworkStream.Listen. When an acceptable connection request packet arrives at the address specified in CreateListener, Listen returns 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 actions cause an error packet to be transmitted to the requestor.

If no suitable packet arrives at the socket in listenTimeout milliseconds, then Listen raises 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 NetworkSteam.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 aborts that process and the signal **ABORTED** is 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 which implies the stream has been established but now is failing. The signal carries with it a reason for the suspension, as described below.

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 acknowleged. If a packet is retransmitted more than 30 times without acknowlegement, 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 acknowlegement 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 may internet routers or because 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 timeout of **NetworkStream.CreateTransducer** has expired and the packets requesting connection establishment have not been acknowleged. This and only this value of **why** may be **RESUME**d.

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 **activelyEstablish** is **TRUE** (e.g., **NetworkStream.Create**), 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. No phone number is currently registered for access to the network specified. The remote machine has not been accessed.

circuitInUse

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

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 was made 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 was made 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. Errors are defined below.

illegalAddress

The local address specified in **NetworkStream.CreateListener** is illegal, because either **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, so this status should never be observed.

```
NetworkStream.ListenTimeout: SIGNAL;
```

NetworkStream.ListenTimeout is 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 avaible to **NetworkStream** clients. In general, the utilities 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. The address 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 returns 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];
```

SetWaitTime 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 a future Pilot release.

6.3.4.4 Closing streams

An implementation of *a* close protocol is provided by Network streams. This method of terminating dialogue on a stream is suggested in the **NS Internet Protocol Specification**. Use of these routines (or any like them) is 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 transmits an empty packet with a **Stream.SubSequenceType** of **NetworkStream.closeSST**. As a side effect, all buffered data is transmitted before the empty packet. After the **closeSST** is transmitted, the procedure attempts to receive a **NetworkStream.closeReplySST**. All data not of subsequence type **closeReplySST** is ignored. When a **NetworkStream.closeReplySST** is received, the procedure transmits **NetworkStream.closeReplySST** and returns 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 transmits a **NetworkStream.closeReplySST**, the side effect of which forces transmission of all currently buffered data. After sending the **closeReplySST**, the procedure attempts to receive a packet with subsequence type of **closeReplySST**. **NetworkStream.CloseReply** raises no signals.

NetworkStream.CloseStatus has the following definitions.

good The close protocol terminated cleanly. All data that was buffered by the stream implementation prior to initiating the close was transmitted and acknowleged 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 was transmitted, but may not have been acknowleged.

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 was transmitted and acknowleged. 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*. Because of the distributed nature of the streams, clients may find some behavior unique. This section points 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, the client process may be blocked *indefinitely* if the remote partner in the connection is not consuming data. This state 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 described below.

inputOptions

The **defaultInputOptions** defined by the Stream interface are almost always inappropriate for Network streams. In particular, **terminateOnEndRecord** should be **TRUE** because Network streams do not implement the *end of stream* concept, but do have the concept of a message or *logical record.* If **terminateOnEndRecord** is **FALSE**, then 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, then it will invariably find itself waiting to complete a transfer when it should be responding to the information it has in hand.

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 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 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 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 Retrieves the number of bytes specified in **block** (**Environment.Block**). If the number of bytes requested is actually transferred, then 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**, then packet boundary semantics are applied to the byte stream and the client is notified by the appropriate signal **Stream.LongBlock** or **Stream.ShortBlock**. These two input options should be used only by clients wishing to directly control buffering, and those clients

would be well advised to use something other than Network streams in their application.

put Appends the specified **block** (**Environment.Block**) to the byte stream. The addition may cause any (bounded) number of internal buffers to be transmitted. The parameter **endRecord** may be set to **TRUE**, causing 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 is detectable by the receiving client.

setSST Causes 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, then an empty buffer is transmitted carrying only the changed **sst** state. Lastly, the new **sst** is recorded and declared to be the current **sst**. When a stream is created, it is assigned a default **sst** of value 0.

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 SPP allocation window constraints, if necessary. **SendAttention** assumes the receiver is taking equally heroic action. Because of the additional overhead in such operations, it is advised that attentions be used judiciously.

waitAttention
Allows the client process to wait for an out-of-band attention notification. If a sending client is transmitting attentions, then the receiving client is responsible for processing 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 is no longer able to receive data.

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. The client is responsible for ensuring that the application data has been satisfactorily delivered before deleting the connection.

getPosition is not implemented by Network streams.

setPosition is not implemented by Network streams.

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 is detectable by receiving clients.

clientData is not used by Network streams.

getSST Returns to the caller the current output SST; that is, 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, then the stream implementation ignores logical record boundaries in incoming packets and continues to process incoming packets until the **get** request is satisfied. If **terminateOnEndRecord** is TRUE, then the stream implementation assumes an exceptional condition at the end of a packet that carries a logical record boundary status, terminates the transfer, and returns with an **endRecord** completion code.

signalEndOfStream is not implemented by Network streams.

6.3.5.3 Completion codes

The following codes are returned from the **get** procedure.

normal A **normal** return is one that satisfies the transfer request; that is, the number of bytes requested.

endRecord

In an attempt to satisfy the input request, a buffer that carries a end of logical record status was consumed and the input options indicated that the request should **terminateOnEndRecord**. The input request *may* not be complete.

sstChange

The *data stream type* of the data stream has changed and the next byte of data in the byte stream will be of type **sst**. The **get** procedure's transfer is not complete.

endOfStream is not implemented by Network streams.

attention

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 **XNS.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, phonenet, clusternet};

PhysicalMedium defines the various types of networks on the device chain. **ethernet** is a 10 Mbit Ethernet, as defined by *The Ethernet*, Version 1.0, September 30, 1980. Also referred to as the *experimental* Ethernet, **ethernetOne** is a 3 Mbit ethernet. Based on the create procedure in RS232C, **phonenet** is a phone line network. **clusternet** is a clusternet network, a group of one or more RS232C ports used for remote workstations.

Router.RoutersFunction: TYPE = {vanillaRouting, interNetworkRouting};

The type of routing function for the current router is defined by **RoutersFunction**.

vanillaRouting is the function for all simple routers. These routers are capable only of requesting routing information, receiving the responses from the internetwork routers and maintaining a table.

interNetworkRouting is the function for internetwork routers. These routers 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**, **NetworkNonExistent** indicates the device specified in the call does not exist.

```
Router.NoTableEntryForNet: ERROR;
```

Raised only by **GetDelayToNet**, **NoTableEntryForNet** indicates that 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];
```

AssignAddress 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 sets it to the number of the local network on the best known route to the destination net. The host field is 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 specifies the number of hops to the remote network the client is interested in. **previous** is the network number obtained from the last call; if this is the first call to the procedure, then **previous** should be set to **startEnumeration**.

EnumerateRoutingTable returns 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 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 gratutious, cause information about networks **maxDelay** or less away to be saved in the local routing table. That information is continuously updated as new information is received. The protocol used to maintain the routing table is by specification unreliable. Therefore, the data being maintained in the table is also unreliable and should be regarded as a hint.

FillRoutingTable followed by **EnumerateRoutingTable** can be used to determine the networks within the desired number of hops from the local net. The filling continues 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, then the greatest **maxDelay** specified is 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** returns 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 specifies the number of the network that the client is interested in; **delay** specifies the number of hops from the local net to **net**.

If **net** is not found in the current routing table, then Pilot requests routing information from local internetwork routers . If **net** is unknown to the local machine and cannot be obtained from the internetwork router, then **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:Router.PhysicalMedium]
  RETURNS [System.NetworkNumber];
```

GetNetworkID provides the network number of any network directly attached to the local machine.

physicalOrder specifies the index of the network driver on the device chain; the *primary* network always has a physical order of 1. **medium** is the type of network involved.

GetNetworkID raises the error **NetworkNonExistent** if there is no such device.

```
Router.GetRouterFunction: PROCEDURE RETURNS [Router.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: Router.PhysicalMedium,
    newNetID: System.NetworkNumber]
    RETURNS [oldNetID: System.NetworkNumber];
```

This protocol should not be called. To do so could jeopardize the integrity of the system.

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 support allows the client access to the *equipment* connected to the RS232C port.

In addition to a channel interface, Pilot provides facilities to start and stop the RS232C channel code, and to dial telephone numbers via RS366 dialing hardware associated with RS232C ports.

The RS232C facilities are implemented by the configuration `RS232CIO.bcd`.

6.5.1 Correspondents

```
RS232CCorrespondents: DEFINITIONS . . . ;
```

The **RS232CCorrespondents** 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**, then 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** is not set to **autoRecognition**.

Note: The auto-recognition facility is not currently supported. A call to **AutoRecognitionWait** always results in a outcome of **illegal**.

6.5.2 Environment types and constants

The **RS232CEnvironment** interface defines the environment of the RS232C channel, including all the parameters of the line.

Types and constants of **RS232CEnvironment** are described below.

```
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 = LONG 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, a number of channel parameters must be specified. 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 specifies a half duplex line or a full duplex line. **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** is the line speed and its choice is dictated by the modem. **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.Correspondent: 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. Flow control is only applicable to asynchronous lines and will be ignored for synchronous lines. When the flow control type is specified as xOnxOff for asynchronous lines, the local system will use the client-specified characters xOn and xOff as a means of controlling the data flow over the line.

```
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 a frame 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 (on 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**. 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**. 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**. 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 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.ClockSource: TYPE = RS232CEnvironment.ClockSource;

RS232C.EncodeData: TYPE = RS232CEnvironment.EncodeData;

RS232C.IdleState: TYPE = RS232CEnvironment.IdleState;

RS232C.DeviceStatus: TYPE = RECORD[statusAborted, dataLost, breakDetected,
  clearToSend, dataSetReady, carrierDetect, ringHeard, ringIndicator,
  deviceError: BOOLEAN];
```

DeviceStatus defines the status of the RS232C device. It is accessed via RS232C.GetStatus and RS232C.StatusWait.

Status codes have the following definitions.

statusAborted

Normally 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.

dataLost

May be returned by two procedures, and has a different meaning for each. **dataLost** returned from RS232C.TransferWait indicates that the receive buffer was not large enough to contain the entire incoming frame. **dataLost** returned from RS232C.GetStatus implies that one or more frames were received with no buffer available. In the latter case, the status is a "latched" state and must be cleared by the procedure for clearing latched status bits.

breakDetected

Applicable only for **lineType = asynchronous**; 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 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.

ringHeard

This latched status indicates that a transient ring indicator status was observed. This status is latched so that the interested process does not miss the temporary raising of the incoming call status.

Note: This indicator is not correctly implemented and its presence or absence is irrelevant.

ringIndicator

Indicates an incoming call on a switched circuit. This status is transitory, but will result in a latched status being set as noted above.

Note: This indicator is not correctly implemented and its presence or absence is irrelevant.

deviceError

Set if a data overrun situation occurred in the hardware, resulting in data loss.

RS232C.DialMode: TYPE = {manual, auto};

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.

```
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};
```

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**. A client wishing to abort all the operations 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
Specifies the number of data bits in a character. The number of bits, right justified, is 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
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
Corresponds to the state of the DTR circuit to the Data Communications Equipment (DCE). It should be set by the client as described in *EIA Standard RS-232-C*. Normally, **dataTerminalReady** is set to **FALSE** when the client wishes to disconnect the communication line.

echoing Specifies whether echoing of input characters should be done by the RS232C channel. If **echoing** is **TRUE**, then all input characters received are echoed by the RS232C channel. If it is **FALSE**, then client using the RS232C channel is responsible for echoing input characters.

encodeData
Used with SDLC. The default for any line type is **nrz**.

flowControl
Specifies whether the channel should perform flow control. If **type** is **xOnXOff**, then the RS232C channel stops output when it receives an **xOff** character and resumes output when it receives an **xOn** character.

frameTimeout
Specifies the intra-frame timeout in milliseconds. 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, then 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 Indicates the state of the line when it is idle; that is, whether to transmit **flags** or **mark**. Default for bit synchronous is **flag**; default for byte synchronous and asynchronous is **mark**.

latchBitClear
Defines the mask used for clearing the latch bits of the **RS232C.DeviceStatus**. Only the latch bits which are set in this mask will be cleared.

lineSpeed
Defines the speed of the line. Its choice is dictated by the modem.

maxAsyncTimeout
Used in conjunction with **frameTimeout** for multiport asynchronous frame timing. Default is 0 (infinite timeout).

parity Specifies the type of parity to be used.

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
Indicates 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 table shows the valid and default values following calls to **Create** or **SetLineType**.

Valid and Default Parameter Settings

	asynchronous	byteSynchronous	bitSynchronous
charLength	any[1] (8)	7,8[3] (8)	any (8)
correspondent	xerox800, ttyHost (xerox800)	xerox850,system6, cmcII (system6) siemens9750	nsSystemElement (nsSystemElement
dataTerminalReady[4]	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)
requestToSend[4]	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**. Clients who wish to never attempt to gain ownership of a line already being used should use **preemptNever**. Clients who wish to always attempt to gain ownership of such a line should use **preemptAlways**. 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. Status types are defined below.

success Returned normally when data transfer has successfully completed.

dataLost Occurs 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
The transfer should be considered successful, but a non-recoverable "shouldn't happen" hardware or software error has occurred. Note that such status changes cause the completion of any pending **RS232C.StatusWait** call (see §6.5.3.5). The **dataLost** latch bit is set in the **DeviceStatus** record if data arrives when no **PhysicalRecord** has been allocated via a **Get** operation. **deviceError** is returned as the **TransferStatus** on all data transfer operations until the **dataLost** latch bit is cleared.

frameTimeout
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; that is, stop bits are missing.

invalidChar,invalidFrame
These states are not implemented.

aborted Occurs if **RS232C.Suspend** is called while the data transfer is outstanding.

disaster Returned when the transmit count equals 0, when the receive byte count equals 0, or when the lower layer code returns a completion of disaster.

6.5.3.2 Signals and errors

RS232C.ChannelInUse: ERROR;

If the channel is active and reservation (pre-emption) fails, then **ChannelInUse** 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;

If the **lineNumber** supplied to the **Create** procedure is invalid, then **InvalidLineNumber** is generated.

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, preemptOthers, 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, then the error **RS232C.NoRS232CHardware** is raised.

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, then the error **RS232C.InvalidLineNumber** is raised.

preemptOthers, preemptMe

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** always succeeds. 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, then 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**, as shown below.

		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.

The client who called **RS232C.Create** is responsible for releasing the channel when the channel is pre-empted or when 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

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];

The **Delete** 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**) are 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, then 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];
```

Get 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];
```

Put 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 specifies the handle obtained via the **RS232C.Create** procedure. **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 is the handle obtained via the **RS232C.Create** procedure. **event** is the completion handle that identifies the activity upon which to wait; that is, the handle returned from the **Get** or **Put** data transfer operation. **byteCount** specifies the number of data bytes transferred upon completion of the call. **status** specifies the status of the completed call; 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];
```

Suspend 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: RS232C.OperationClass];
```

Restart 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. This information 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, then 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**, then all channels are suspended.

```
RS232CControl.Start: PROCEDURE;
```

Start allows channel creation after a **Stop** call.

```
RS232C.NoCommunicationHardware: ERROR;
```

NoCommunicationHardware is raised when a client attempts to create a channel after **RS232C.Stop** has been called.

6.5.5 Auto-dialing

```
Dialup: DEFINITIONS . . . ;

DialupExtras: 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 are directed to the equipment answering the telephone call.

Note: 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

```
DialupExtras.DialExtra: PROCEDURE [dialerNumber: CARDINAL, number: LONG POINTER TO
  Dialup.Number,retries: Dialup.RetryCount, dialerType: DialupExtras.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 };

DialupExtras.DialerType: TYPE = MACHINE DEPENDENT {RS366(0), Ventel(1),
  smartmodem(2), RacalVadic(3), V25bis(4), V25(5), other(6), last(LAST[CARDINAL])};
```

Note: Use DialupExtras.DialExtra instead of Dialup.Dial.

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 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 Dialup.pause, the dialup implementation attaches no semantics to any of the bit patterns it receives; they are simply passed to the dialer hardware. The client is responsible for knowing what bit patterns represent special characters, such as EON and SEP, for the 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).

6.5.5.1 Outcome

```
Dialup.Outcome: TYPE = { success, failure, aborted, formatError, transmissionError,
  dataLineOccupied, dialerNotPresent, dialingTimeout, transferTimeout };
```

The values of **Outcome** are 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 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 situation 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 *lack-of-hardware* 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 no answer) nor successfully transferred control to the modem. This situation indicates a hardware problem which should be investigated.

6.5.5.2 DialerType

```
DialupExtras.DialerType: TYPE = MACHINE DEPENDENT {RS366(0), Ventel(1), smartmodem(2),
  RacalVadic(3), V25bis(4), V25(5), other(6), last(LAST[CARDINAL])};
```

The **number** field in the **DialupExtras.DialupExtra** interface call is closely tied to the **DialupExtras.DialerType** that is passed to the **DialupExtras.DialExtra** call.

For **DialupExtras.DialExtra** calls with the **RS366** dialer type set, dial number specifications are unchanged.

For **DialupExtras.DialExtra** calls with the **Ventel** dialer type, the following semantics apply for the **number** field.

The phone number is a string of 0 to 30 characters from the following set: the dial digits **0** through **9** and the dial modifiers **&**, **%** and **space**. The dial modifiers have the following semantics:

& **Wait for Dial Tone.** The **&** modifier causes the Ventel to wait for up to five seconds for a secondary dial tone before dialing the following numbers.

% **Pause.** The **%** modifier causes the Ventel to pause for five seconds before dialing the subsequent numbers.

space Has no effect on the dialing of the specified number and is included only for readablity.

For example,
9&1234% %408 555 1212

In this example, the **9** was dialed to get an outside line. When a second dial tone was detected, the four digit billing code was issued and the modem waited for ten seconds before dialing the rest of the number.

For **DialupExtras.DialExtra** calls with the **smartmodem** dialer type, the following semantics apply for the **number** field.

The phone number is a string of 0 to 39 characters from the following set. The dial digits **0** through **9 A B C D #** and *****, the dial modifiers **P T R W , @ !** and **S** and the punctuation symbols **(,), -** and **space**. These punctuation symbols have no effect on the number dialing and are only provided for user readability.

The symbols **A B** and **D** have the following meaning.

A Off-hook in answer mode.

B Selects CCITT V.22 operation when communicating at 1200 bps.

D Dial number which follows **D** in the command line.

The symbols **P T R W , @** and **!** are dial modifiers and have the following effect when encountered in a dial number string.

P **Pulse Dial.** The **P** modifier causes the smartmodem to pulse dial the numbers that follow.
Pulse dialing example:
P 5551212

T **Touch-Tone Dial.** The **T** modifier in the dial string causes the modem to Touch-Tone dial the numbers that follow (default).
Touch-Tone dialing example:
T 5551212
Dialing an outside line example:
T9,5551212
In this example the modem dials 9 to access an outside line and waits two seconds before dialing the rest of the number (see **,** below).

R **Originate Call In Answer Mode.** The **R** modifier must be the last character in the dial string. It causes the smartmodem to originate the call in answer mode, and is used to dial an originate-only modem.

W **Wait For Dial Tone Before Dialing.** The **W** modifier in the dial string causes the smartmodem to wait up to thirty seconds for a dial tone before dialing the numbers that follow.
W modifier example:
9 W 5551212
In this example the modem dials **9** and waits up to thirty seconds for a subsequent dial tone and then dials the rest of the numbers.

, **Delay A Dial Sequence.** The **,** modifier in a dial string causes the smartmodem to pause two seconds before processing the next symbol in the string. Multiple commas can be used consecutively to generate longer pauses.

@ **Wait For Quiet Answer Before Dialing.** The **@** modifier in the dial string causes the smartmodem to wait up to thirty seconds for one or more rings followed by five seconds of silence before processing the next symbol in the dial string.

! **Initiate Flash.** The ! modifier causes the smartmodem to go on-hook for one-half second. This is equivalent to holding the switch-hook botton on your telephone down for one-half second.

For DialupExtras.**DialExtra** calls with the **RacalVadic** dialer type, the following semantics apply for the **number** field.

The phone number is a string of 0 to 16 or 0 to 32 characters, depending on how the modem hardware is configured, from the following set: the dial digits **0** through **9** and the dial modifiers **=** . The dial modifier **=** allows for tandem dialing and causes the modem to wait for a secondary dial tone before dialing the subsequent numbers.

For example,
9=4085551212
In this example, the **9** was dialed to get an outside line, the modem waited for a secondary dial tone, then dialed the rest of the number.

For DialupExtras.**DialExtra** calls with the **V25bis** dialer type the following semantics apply for the **number** field.

The phone number is a string of characters from the following set: the dial digits **0** through **9** and the dial modifiers **:, <, =, >**. The dial modifier causes the modem to wait for a secondary dial tone before dialing the following numbers. The modifiers **<, =, >** are for national use; the individual modem manual should be consulted for their use.

For example,
9:4085551212
Here, the **9** was dialed to get an outside line, the modem waited for a secondary dial tone, and dialed the rest of the number.

Note: The value of the **frameTimeout** in the RS232C channel will be changed according to the following list of specifications when using the **Dial** and **DialExtra** procedures:

For calls with the **smartmodem** dialer type, the frame timeout will be set to 100 ms when the call is completed.

For calls with the dialer type **Ventel**, **RacalVadic**, or **V25bis**, the frameTimeout will be set to 30 ms.

Caution: The frame timeout should not be reset after a connection has been established, since the DTR signal is reset when the frame timeout is changed.

6.5.5.3 Miscellaneous facilities

```
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 RS366 (dialer) ports available.

```
Dialup.pause: Environment.Byte = LAST[Environment.Byte];
```

When passed in the parameter **number**, **pause** causes the dialer to wait six 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 modeling and controlling the interaction of programs through type-safety and signals, Courier provides facilities for modeling 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.

disjoint data — In general, any Mesa structure that causes Courier to access data outside the current parameter area is disjoint. Examples of disjoint data are **StringBody**s and **ARRAY**s described by **LONG DESCRIPTOR**s.

parameter area — 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.

remote program — A remote program usually represents a complete *service*, and the remote procedures it contains represent the *operations* of that service.

RPC — Remote Procedure Call. In this context it refers to the actual processing of arguments and results, that function being separate from *bulk data*, for instance.

server — The Courier server is the Mesa Courier client that provides or exports a service.

transport — 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.

user — 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 result in the signal Courier.Error[invalidHandle] being raised.

Note: **Delete** does not 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 (export) that service via **ExportRemoteProgram**. That action provides a template needed by Courier to complete the binding process as it is 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 are rejected. **UnexportRemoteProgram** does 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. Courier relieves 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 is 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: Because of 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 is 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 does 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. If that transport fails, then the local machine abandons the call and raises the signal **Courier.Error[transportTimeout]**. Courier watches to ensure that the call returns in a client-specified period of time (**timeoutInSeconds**). If that time expires, then Courier raises 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 signal is 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 **UNWIND**ing. 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; 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 §6.6.6.2) 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 searches its internal lists of exports for *an* appropriate export and calls 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. Even if there are no procedure arguments, **arguments** must still be called with parameters of Courier.nullParameters. If actual arguments do exist, then the **location** field of the **Parameters** record is assumed to point to an uninitialized but writeable 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 §6.6.6.2) 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

Courier performs a considerable amount of error processing. 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
: Most likely to happen during initial attempts at establishing a connection, but could happen at any time. The error 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; indicates that the device is either currently or permanently unavailable. 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

Applies to circuit-oriented media only; 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. The condition 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 may 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

The service has taken too long to formulate its reply. The calling machine has abandoned the call, and the results cannot be delivered. The server *never* retries operations. This error is observed only on server machines.

unknownErrorInRemoteProcedure

An undefined error has occurred. The server machine's integrity is in doubt and retrying could compound the problem. This error is observed only by user machines.

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, then 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

Bulk data transfer protocol implementors are clients of the filtered byte stream provided by Courier for that purpose. The 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**. This error code will only be observed by implementors of the bulk data transfer protocol.

parameterInconsistency

Client parameter processing error, probably due to a malformed Mesa data item or an invalid implementation of the client's protocol (in the description routine). In such cases, it 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

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. This error will only be observed at the user during initial transport creations

duplicateProgramExport

The **programNumber** and **versionRange** parameters of **Courier.ExportRemoteProgram** matched *exactly* with those already known by Courier. This error code is observed only when attempting to export a service.

noSuchProgramExport

The **programNumber** and **versionRange** specified in the unexport request (**Courier.UnexportRemoteProgram**) did not have an equivalent known to Courier. This error code is observed only when attempting to unexport a remote service.

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. Do not retry the operation.

noError

This error code 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, then it notifies 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.Arguments];

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 **NoSuchProcedureNumber**, 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 is 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 is translated by Courier to **Courier.Error[invalidArguments]** at the user. Both server and user code may raise this signal; the server does not translate the error locally, but it rejects the call, sends the information to the user, where **Error[invalidArguments]** is 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, then **arguments** should be assigned (or defaulted) a value of **Courier.nullParameters**.

Note: Courier will call the client's argument description routine before **UNWIND**ing 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 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; that is, 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**, then the value returned for **sH** is **NIL**. If the parameter **requestDataStream** is **TRUE**, then 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**.

If an attempt is made to release a stream that was never checked out, then the error **Courier.Error[streamNotYours]** is 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. However, because the Courier Protocol is intended for a heterogeneous internet, 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.

Courier data type	Corresponding Mesa data type
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. The Courier client is responsible for using the restricted form shown below.

Courier data type	Corresponding Mesa data type
BOOLEAN	MACHINE DEPENDENT RECORD [zeros: [0..77777B], value: BOOLEAN]
{$id_1(v_1)$, ... $id_n(v_n)$}	MACHINE DEPENDENT {$id_1(v_1)$, ... $id_n(v_n)$, LAST[CARDINAL]}
RECORD[id_1: $Type_1$, ... id_n: $Type_n$]	MACHINE DEPENDENT RECORD [id_1: $Type_1$, ... 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.

Courier data type	Corresponding Mesa data type
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 => [$Type_{n+m}$]]
SEQUENCE *n* OF *Type*	DESCRIPTOR FOR ARRAY 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 translate Mesa data types to Courier data types. This action is sometimes referred to as *serialization* or *marshalling* of data. The action occurs when call

parameters are processed by the user or when return parameters are processed by the server.

store To translate Courier data types to Mesa data types. This action is sometimes referred to as *deserialization* or *unmarshalling*. The 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 allocates 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 To release storage nodes. 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 POINTER TO 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** is allocated from the notes object **zone** by the store operation.

Note: The maxlength attribute of the Mesa *StringBody* will be lost in the fetching operation. Consequently, stored strings will always have a maxlength 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, then 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.: TYPE = PROCEDURE [
  site: LONG POINTER, elementSize, upperBound: CARDINAL];
```

NoteArrayDescriptor 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 is 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 calls that routine, and it in turn calls **noteSize**. The beginning of the disjoint area and the size define a new parameter area. That parameter area is allocated from the notes object **zone** during store operations. Pointers are not Courier data types. The pointer is dereferenced and the dereferenced object processed during a fetch operation. An appropriate object is allocated from the notes object **zone** and a pointer to that object is placed in the client parameter area during store operations. No notion of a pointer (or its absence) is 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 no pointer is 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 is 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 an 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; for example, **notes.noteDeadSpace[site, 0]** causes all *unnoted* data in the current parameter area to be processed and then consumes 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 processes only the bytes defined by the **Environment.Block**, and consume snothing from the current parameter area. Moreover, Courier does not 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 ensuring 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 POINTER**s, 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; for example, 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];
```

Serialize and Deserialize 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];
```

LocalSystemElement 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 makes 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 are 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.

6.7 Network Binding

In order to complete a remote procedure call, the active machine must bind to an appropriate passive system element. A required part of that binding is a network address (System.NetworkAddress). The Network Binding protocol provides a flexible mechanism for finding the network address of a passive machine or set of passive machines as well as information that may be applied to objectively qualify the response.

NetworkBinding is a broadcast-based protocol that supports a machine selection abstraction based on user-defined predicates and responses. The goals of this protocol are:

1. To quickly locate an optimal machine on a specified network currently satisfying a user-defined selection criterion
2. To locate with reasonable speed an optimal machine in an area of the internet currently satisfying a user-defined selection criterion
3. To reliably locate all machines on a specified network currently satisfying a user-defined selection criterion

This protocol is not meant to replace all other remote procedure call binding mechanisms. When a service's topology and the information used in making a selection remain static as compared with the propagation time of the Clearinghouse, a static binding mechanism based on the Clearinghouse is sufficient, appropriate, and preferred.

Additionally, binding mechanisms based on broadcasting can suffer from problems not found in a Clearinghouse-based mechanism. Broadcast messages are expensive, so care must be taken to limit the amount of binding done by the stubs. The cost increases rapidly if the binding mechanism must search several networks.

Since the Network Binding protocol will be used to bind to any Courier program, it has no dependencies on any other Courier programs or protocols (outside of Courier itself). Issues like authentication and consistency are not handled by this protocol; they should be addressed as needed by the system elements once the binding to the passive element is made and the initial remote procedure call (RPC) is made.

Appendix G provides an example of how the NetworkBinding operations are used in a software package.

6.7.1 Description

NetworkBinding: DEFINITIONS = ...

The Network Binding facility is included in `CourierConfig.bcd`. The modules `NetworkBindingClient.bcd`, `.mesa` and `NetworkBindingServer.bcd`, `.mesa` implement the facility. Public client access is made available via the interface **NetworkBinding**.

6.7.2 Types and constants

NetworkBinding.Conjunct: TYPE = RECORD [LONG CARDINAL];

The client and service stubs agree a priori on the selection criterion a conjunct represents and the number and type of the parameters needed by the service stub to determine whether the conjunct is met, and the number of parameters, if any, in the response. The allocation of conjunct numbers is associated with the **NetworkBinding.RemoteProgram** being specified. The allocation of **NeworkBinding.Conjunct**, **NetworkBinding.cTRUE**, and conjunct

numbers associated with the NetworkBinding.RemoteProgram of NetworkBinding.dontCare are allocated by Xerox.

```
NetworkBinding.cTRUE: NetworkBinding.Conjunct = [0];
```

If the client stub does not require any selection rule other than the remote program, then the conjunct cTRUE can be specified. This conjunct can never be reallocated by stub implementors. Any bind request that uses the conjunct cTRUE is restricted to an *empty* response.

```
NetworkBinding.defaultHops: CARDINAL = 3;
```

One of the Network Binding procedures (NetworkBinding.BindToFirstNearby) is capable of searching all networks within a client-specified radius of the requester. The default and recommended maximum radius for such calls is three hops.

```
dontCare: NetworkBinding.RemoteProgram = [0, 0];
```

If the client stub does not care whether a specific remote program is exported, then it can specify a program of NetworkBinding.dontCare. This might happen, for example, if the client stub is meant to communicate with a remote program using a protocol other than Courier.

```
NetworkBinding.Responses: TYPE = LONG POINTER TO NetworkBinding.ResponseSequence;
NetworkBinding.ResponseSequence: TYPE = RECORD [
  elementSize: NATURAL,
  elementCount: NATURAL,
  element: SEQUENCE COMPUTED NATURAL OF WORD];
```

Responses to a NetworkBinding.BindToAll request are packaged together in a Mesa sequence and passed to the client as a NetworkBinding.ResponseSequence. Within this record the field definitions are as follows:

elementSize
: The value of this field is the size of each element of the element sequence in words. Since the element sequence always includes a System.NetworkAddress and the maximum length of the data portion of any response is limited to approximately 250 bytes, the expected range of elementSize is between 6 and 125.

elementCount
: The elementCount field contains the number of responses in the element sequence. The minimum value for this field is zero, indicating that there were no affirmative responses to the query.

element
: The sequence of responses, concatenated into a single node. Each element of the sequence includes as its first field the network address of the machine that sent the response. The remainder of each element of the sequence is that response, in the Mesa type defined by the client supplied Courier description routine.

The sequence must then be coerced into the appropriate client-defined Mesa type. The actual Mesa type will be of the form

```
ClientType: TYPE = RECORD[
  address: System.NetworkAddress,
  response: {client supplied type}];
```

This particular form of return is used for two reasons. First, Network Binding does not know the format of the response. All it knows is how to concatenate the responding station's network address with some number of bytes of data specified in a client implemented description routine. Second, the form permits the client to delete all storage allocated by the procedure call with one call into the heap package.

Note: If the bind request specified **NetworkBinding.cTRUE** as the conjunct, then the response is limited to an empty **NetworkBinding.ResponseParameter** record; that is, **response.elementLength** will be equal to six.

```
NetworkBinding.nullPredicate: NetworkBinding.PredicateRecord = [
  NetworkBinding.pTRUE, Courier.nullParameters];
```

In appropriate cases, such as when simply trying to find the first or all stations that implement the network binding protocol, the constant **NetworkBinding.nullPredicate** may be used.

```
NetworkBinding.nullResponse: NetworkBinding.ResponseRecord = Courier.nullParameters;
```

A common case is when no response other than the network address of the responding host is required. The proper response record is defined by **NetworkBinding.ResponseRecord**.

```
NetworkBinding.Predicate: TYPE = RECORD [
  program: NetworkBinding.RemoteProgram,
  conjunct: NetworkBinding.Conjunct];
```

A predicate contains two conditions, both of which must be met by a service stub for it to be selected. The first condition is that a specific remote program is exported by the passive system element. A remote program is specified as its Courier program number and version. The second is a matching conjunct number. The **program** field may have the value of **NetworkBinding.dontCare** and/or the **conjunct** field may have the **NetworkBinding.cTRUE**.

```
NetworkBinding.PredicateRecord: TYPE = RECORD [
  pred: NetworkBinding.Predicate, param: Courier.Parameters];
```

A predicate and the parameters to the conjunct are represented as a predicate parameter. A predicate parameter is specified in all Network Binding procedure calls. It includes the **predicate** defined for the procedure and supplies the location and the routine to describe the predicate parameter. (See §6.6.6 for definition of description routines.)

```
NetworkBinding.pTRUE: NetworkBinding.Predicate = [
  NetworkBinding.dontCare, NetworkBinding.cTRUE];
```

The predicate **pTRUE** is satisfied by any service stub. Its utility is for the most part limited to finding all of the passive system elements (i.e., Network Binding servers) on a network.

```
NetworkBinding.RemoteProgram: TYPE = RECORD [
  programNumber: LONG CARDINAL, version: CARDINAL];
```

The specification of the NetworkBinding.RemoteProgram restricts stations that do not currently export that version of the specified remote program from responding. For example, it might be set to [2, 3], specifying that only machines exporting the Clearinghouse Protocol, Version 3, should respond.

```
NetworkBinding.ResponseRecord: TYPE = Courier.Parameters;
```

The NetworkBinding.ResponseRecord is used to pass the description routine of the expected response to Network Binding. (See §6.6.6 for definition of description routines, etc.)

6.7.3 Errors

```
NetworkBinding.DataTooLarge: ERROR;
```

DataTooLarge may be raised by either the client or server stub. It indicates that the data specified in the predicate or response is too large to fit into a single internet packet. Because of the overhead of the protocol, predicates and responses are limited to approximately 250 bytes. Even that space is in competition with the protocol. Consequently, heavy client usage of *predicate* data will cause convergence of any of the network binding procedures that are looking for all instances of a service to be slow.

```
NetworkBinding.NoBinding: ERROR;
```

NoBinding may be raised by either the client or server stub, but with differing semantics. An implementor of a *response procedure* would raise this signal if it found that the predicate could not be satisfied. The client of the *predicate procedure* must be prepared to catch this signal if there is no positive response to the binding query.

6.7.4 Client procedures

```
NetworkBinding.BindToAllOnNet: PROCEDURE [
    predicate: NetworkBinding.PredicateRecord ← NetworkBinding.nullPredicate,
    responseDescription: Courier.Description ← NIL,
    net: System.NetworkNumber ← System.nullNetworkNumber,
    zone: UNCOUNTED ZONE ← NIL]
    RETURNS [responses: NetworkBinding.Responses];
```

NetworkBinding.**BindToAllOnNet** attempts to reliably locate all stations on the specified **net** that can answer affirmative to the **predicate**. All of the responses are collected into a single node pointed to by **responses** that must then be coerced into the client specific data type. The **responses** node is allocated from **zone**. The default value of **zone** implies that Heap.**systemZone** should be used.

The **responseDescription** is called for each response that is collected from **net**. Its default value implies that there is no client-specified response parameter, just the default responding station's network address.

The default value of **net** implies that the local network should be searched.

If no positive responses are received from the specified **net**, then this procedure returns **responses** = **NIL**.

NetworkBinding.BindToAllOnNet may take quite some time to complete. The actual time depends on the number of stations responding, the distance **net** is from the station calling **NetworkBinding.BindToAllOnNet**, and the size of the **predicate param** record. For example, the time to locate 200 stations on the local network using **NetworkBinding.nullPredicate** may exceed one minute.

```
NetworkBinding.BindToFirstNearby : PROCEDURE[
    predicate: NetworkBinding.PredicateRecord ← NetworkBinding.nullPredicate,
    responseDescription: Courier.Description ← NIL,
    maxHops: CARDINAL ← NetworkBinding.defaultHops,
    zone: UNCOUNTED ZONE ← NIL]
    RETURNS [responder: System.NetworkAddress, response: LONG POINTER];
```

NetworkBinding.BindToFirstNearby attempts to locate the first station within a specified radius, **maxHop**, of the requesting station that can answer affirmative to the **predicate**. The result is returned as the **responder**, which is the network address of the station responding, and a pointer to the client specified **response**. **response** should be **NARROW**ed into the proper data type. The **response** node is allocated from **zone**. The default value of **zone** implies that **Heap.systemZone** should be used.

The **responseDescription** is called for the first response that is collected. Its default value implies that there is no client-specified response parameter, just the default responding station's network address.

Note: **NetworkBinding.BindToFirstNearby** is implemented using an *expanding ring broadcast*. The time to locate an acceptable server varies, depending on how many networks must be searched in order to get a single positive response.

If there are no responses to the query, then **NetworkBinding.NoBinding** may be raised. If the **param** description routine specifies more data than can be contained in a single packet (approximately 250 bytes), then **NetworkBinding.DataTooLarge** may be raised.

```
NetworkBinding.BindToFirstOnNet: PROCEDURE[
    predicate: NetworkBinding.PredicateRecord ← NetworkBinding.nullPredicate,
    responseDescription: Courier.Description ← NIL,
    net: System.NetworkNumber ← System.nullNetworkNumber,
    zone: UNCOUNTED ZONE ← NIL]
    RETURNS [responder: System.NetworkAddress, response: LONG POINTER];
```

NetworkBinding.BindToFirstOnNet attempts to locate the *first* station on the specified **net** that can answer affirmative to the **predicate**. The result is returned as the **responder**, which is the network address of the station responding, and a pointer to the client specified **response**. **response** should be **NARROW**ed into the proper data type. The **response** node is allocated from **zone**. The default value of **zone** implies that **Heap.systemZone** should be used. The default value of **net** implies that the local network should be queried.

The **responseDescription** is called for the first response that is collected. Its default value implies that there is no client-specified response parameter, just the default responding station's network address.

If the **param** description routine specifies more data than can be contained in a single packet (approximately 250 bytes), then **NetworkBinding.DataTooLarge** may be raised.

```
NetworkBinding.VerifyBinding: PROCEDURE [
    predicate: NetworkBinding.PredicateRecord ← NetworkBinding.nullPredicate,
    responseDescription: Courier.Description ← NIL,
    host: System.NetworkAddress,
    zone: UNCOUNTED ZONE ← NIL]
    RETURNS[response: LONG POINTER];
```

NetworkBinding.VerifyBinding attempts to contact the station specified in **host**. The result is returned as pointer to the client-specified **response**. **response** should be **NARROW**ed into the proper data type. The **response** node is allocated from **zone**. The default value of **zone** implies that **Heap.systemZone** should be used.

The **responseDescription** is called for the response that is collected. Its default value implies that there is no client-specified response parameter. In such cases the value returned for response is always **NIL**. Since this is a directed request (rather than broadcasted), a **NIL** return is interpreted as a positive response and a negative or null response is always signaled as described below.

If there is no response to the query, then **NetworkBinding.NoBinding** may be raised. If the **param** description routine specifies more data than can be contained in a single packet (approximately 250 bytes), then **NetworkBinding.DataTooLarge** may be raised.

6.7.5 Server procedures

The following types and procedures are used only by clients that implement Network Binding servers.

```
NetworkBinding.PredicateProcedure: TYPE = PROCEDURE [
    pred: NetworkBinding.Predicate,
    args: LONG POINTER,
    response: NetworkBinding.ResponseProc];
```

NetworkBinding.PredicateProcedure defines the type of a procedure that the client must implement if the application protocol intends to process predicates with client-specified data. The NetworkBinding server calls the procedure whenever a binding request arrives at the station that satisfies **conjunct** and **program** portions of the **NetworkBinding.Predicate**. The data that makes up the client-specified portion of the binding request is pointed to by **args**. This pointer should be **NARROW**ed into the appropriate Mesa data structure.

If the client **PredicateProcedure** cannot answer in the affirmative to the bind request, then it must raise the error **NetworkBinding.NoBinding**, rather than returning from the predicate procedure.

```
NetworkBinding.ResponseProc: TYPE = PROCEDURE [
    response: NetworkBinding.ResponseRecord ← NetworkBinding.nullResponse];
```

If the client **PredicateProcedure** can answer in the affirmative, then it must call **response**. However, if the response is null, then the procedure can be called with the default argument, **NetworkBinding.nullResponse**. If the response is too large to fit into a single

packet (approximately 250 bytes), then the Network Binding server will raise NetworkBinding.DataTooLarge.

```
NetworkBinding.RegisterPredicate: PROCEDURE [
    programNumber: LONG CARDINAL,
    versionRange: Courier.VersionRange,
    conjunct: NetworkBinding.Conjunct,
    proc: NetworkBinding.PredicateProcedure,
    predicateDescription: Courier.Description,
    zone: UNCOUNTED ZONE ← NIL];
```

Before a station can respond to a binding request, the service client must register the information about the binding. Registration is done by calling NetworkBinding.RegisterPredicate. Once registered, the server responds appropriately as defined by the arguments of NetworkBinding.RegisterPredicate.

The client must provide a programNumber and the versionRange of the Courier program associated with the Network Binding application. It *must* be specified, but it may be ignored (see NetworkBinding.dontCare in §6.7.2). The value of conjunct specified can be any value, but if it is assigned the well known value NetworkBinding.cTRUE, then the client proc must be NIL or not return any response arguments via the NetworkBinding.ResponseProc. If proc is NIL, then predicateDescription should also be NIL; otherwise it is the Courier *description* procedure that describes the data present in the binding call's arguments.

```
NetworkBinding.DeregisterPredicate: PROCEDURE [
    programNumber: LONG CARDINAL,
    versionRange: Courier.VersionRange,
    conjunct: NetworkBinding.Conjunct];
```

NetworkBinding.DeregisterPredicate undoes the effect of NetworkBinding.RegisterPredicate. The values assigned to programNumber, versionRange and conjunct must match exactly with the values used in the registration or the call will be ignored.

6.8 XStream – Bulk Data Protocol

```
XStream: DEFINITIONS = ...
```

XStream is the programming interface to an implementation of the Bulk Data Transfer Protocol as described in Addenda to Standards, XSIS 138301. This implementation prescribes to the intent of the specification in that it "... is used in conjunction with Courier-based protocols whose remote procedures produce or consume bulk data."

6.8.1 Interface definition

```
XStream.Object: TYPE;
XStream.Handle: TYPE = LONG POINTER TO XStream.Object;
```

XStream uses an XStream.Object to cache information about the state of the current session. A pointer to that object, an XStream.Handle, can be acquired via XStream.Make.

```
XStream.Request: TYPE = RECORD[SELECT access: * FROM
    none => [],
    stream => [sH: Stream.Handle],
    proc => [proc: PROC[XStream.Handle]],
    deferred => [sink, source: Courier.SystemElement],
    ENDCASE];
```

The **XStream.Request** structure permits clients to define their desires about the mode of access that will be used during the session. The following four modes are permitted:

none A client should select a **none** variant when it wishes a *null* bulk data transfer to occur. When supplied a **none** *sink*, XStream discards the data that it would have otherwise sent. When supplied with a **none** *source*, XStream acts as if it has received no data.

stream If the client wishes to make an *immediate* bulk data transfer and is in possession of a well behaved stream, then it may use the **stream** variant of the request record. Once that information is recorded by XStream, the bulk data transfer will proceed with no direct client interaction (with the exception of having to describe the bulk data argument).

proc The **proc** variant defines an immediate bulk data transfer much like the **stream** variant. The difference is that when the transfer is initiated, the **proc** defined in the **XStream.Request** object is called. This is handy if the source or sink of the bulk data is not a well behaved stream; for example, the data may need filtering or even generation.

deferred The **deferred** variant defines that the bulk data transfer will be of the *third* party variety. The **System.NetworkAddress** of the machines involved in the transfer are included as **source** and **sink**.

Note: *Null* and *immediate* requests do not require that the remote's address be known. That information is not needed since it is available (and implicit) from the Courier connection.

```
XStream.Create: PROC[XStream.Handle] RETURNS[Stream.Handle];
```

XStream.Create gives the bulk data client a **Stream.Handle** on which to operate. The client is the owner of the representative stream until the stream is deleted (**sH.delete**). It is always the client's responsibility to delete the stream, or any other resource, that it created.

In cases where **XStream.Create** is called to create a stream to be used as a sink (i.e., **sH.put**), the deletion of the stream is taken as the indication of the end of the data. That action is translated into a **Stream.EndOfStream** at the source side of the stream.

```
XStream.Copy: PROC[sink, source: Stream.Handle];
```

XStream.Copy may be used to copy the source stream into the sink stream. In order for a stream to be used as an argument of **XStream.Copy**, that stream must be fairly well behaved. The definition of *well behaved* is included in §6.8.2. Needless to say, streams returned from **XStream.Create** meet the well behaved criteria.

sink and **source** are both provided by the caller. The data represented by the **source** stream will be the object of read operations. The data read from the **source** will be written to the **sink** stream. The transfer terminates when the source stream raises **Stream.EndOfStream**. A stream acquired from **XStream.Create** is an appropriate stream for either **source** or **sink**.

```
XStream.Destroy: PROC[XStream.Handle];

XStream.Make: PROC[request: XStream.Request]
    RETURNS[handle: XStream.Handle];
```

XStream.Destroy and XStream.Make are used by the Courier user client (at a level of abstraction below the bulk data client) to create the data structures needed to convey the information about the bulk data stream.

```
XStream.ServerCheckout: PROC[Courier.Handle, XStream.Request];
```

XStream.ServerCheckout is somewhat analogous to XStream.Create and XStream.UserCheckout. It associates the particular Courier.Handle to a client's XStream.Request. It is important to note that the client is simply making an association of the two state objects; it is not allocating resources. Therefore, no XStream.Handle is returned, and no XStream.Destroy call is required.

```
XStream.UserCheckout: PROC[Courier.Handle];
```

XStream.UserCheckout is used by the Courier user client to synchronize the use of Courier's transport and to complete the information required to initiate the transfer.

```
XStream.AbortTransfer: PROC[Stream.Handle];
```

In order to abort a transfer in a consistent manner, the client must call XStream.AbortTransfer. This operation is applicable for both the source and sink operation clients. After a stream is aborted, the only permissible operation is Stream.Delete (sH.delete). All other operations will result in ERROR ABORTED.

```
XStream.DescribeSink: Courier.Description;

XStream.DescribeSource: Courier.Description;
```

Objects are not completely formed when they are created. They are completed as access to them unfolds. In particular, an XStream.Object is not typed as a sink or source until it is (de)serialized. Consequently, only generic XStream.Handles are included as procedure call arguments. Before the object is (de)serialized, it cannot be mapped into a stream. The objects are translated to streams by the client-supplied procedures contained in the XStream.Request during the processing of the client's XStream.Create.

6.8.2 Additional semantics

XStream.Objects are created by the client ostensibly for the use with a single remote procedure call. An object may be used at most once and must be deleted after the remote procedure call.

The streams utilized for immediate transfers are variants of Pilot byte streams (i.e., Stream.Handle) with these restrictions:

- Only one stream per remote procedure call is allowed.
- The stream is simplex; that is, transfers data in only one direction.
- SSTs are not supported. The stream's SetSST and GetSST routines will result in ERROR Stream.InvalidOperation. XStream will never raise the signal Stream.SSTChange.

- Attentions are not supported. The stream's **SendAttention** and **WaitAttenion** routines will raise **ERROR Stream.InvalidOperation**. **XStream** will never raise the signal **Stream.Attention**.
- If a stream is created by a bulk data client within the **XStream.Request** callback procedure, then that stream is owned by that client. Normal termination of the transfer of data by the source client is signaled by deleting the stream (**sH.delete**). Normal termination is relayed to the sink client in the form of a **Stream.EndOfStream** status or signal.
- Clients may provide streams to be used as sources or sinks.
- Client sink streams are not required to process any signals. The status of the transfer is implied from the results of the remote procedure call. The client then has the responsibility for the stream.
- Client source streams must signal **Stream.EndOfStream** to indicate the end of the transfer.
- Source **XStreams** may be truncated, indicating some sort of transfer error. This is done by calling **XStream.AbortTransfer**, which results in **ERROR ABORTED** being raised at the sink **XStream**.

6.9 PhoneNet configuration

The PhoneNet configuration provides XNS network access to Pilot-based systems via a synchronous RS232C serial line, which may be either a leased line or a dialup line. The access may be half duplex (8010 machines only) at line speeds between 2400 to 9600 bits per second.

Note: 6085 machines do not operate reliably at speeds above 4800 bits per second because of the 6085's inability to handle simultaneous sending and receiving of data at the higher rates.

The PhoneNet configuration is a representation of a Data Link Layer in the ISO model. The data link protocol used is the Synchronous Point to Point Protocol (SPTP), XSIS 158412, dated December 1984, plus some extensions. The SPTP specification defines Version 3 of the data link protocol. The PhoneNet configuration also implements Version 2 to maintain backward compatibility with System Interface Units [SIU] and Version 4 (not yet formally specified) to permit fragmentation and reassembly of Network Layer packets in order to bypass current hardware limitations.

The support is implemented by `PhonenetConfig.bcd`. Two interfaces are exported by the configuration: **PhoneNet** and **PhoneAdoption**.

6.9.1 PhoneNet

```
PhoneNet: DEFINITIONS = ...
```

6.9.1.1 Types

```
PhoneNet.EntityClass: TYPE = MACHINE DEPENDENT {
    internetworkRouter(0), clusterRouter(1), siu(2), remoteHost(3)};
```

Each machine that supports the SPTP protocol must declare itself to be one of the classes defined by **PhoneNet.EntityClass**. During connection establishment negotiation, only

certain combinations are permitted (defined in SPTP). The class of the machine is selectable via the PhoneNet.Initialize procedure (defined below).

```
PhoneNet.Negotiation: TYPE = {active, passive};
```

Clients may also select whether they want to be active or passive parties in the establishment negotiation procedure. In active mode, once the physical medium becomes available, the machine will gratuitously transmit negotiating packets soliciting the connection. In passive mode, a machine will only respond to connection requests that it has received.

6.9.1.2 Errors

```
PhoneNet.Unsupported: ERROR;
```

Unsupported will be raised by the PhoneNet implementation in situations where a client has asked for a feature that may be implied by the interface but actually is not supported by the current configuration. At present, there only two such cases. The first case is an attempt to call PhoneNet.Initialize with the argument **lineNumber** anything other than zero. The second case is an attempt to transmit a packet larger than 600 bytes when the negotiated version of the data link protocol is not version 4.

```
PhoneNet.IllegalEntityClass: ERROR;
```

IllegalEntityClass will be raised by PhoneNet.Initialize if and only if the client specifies **ourEntityClass** as **siu**. A Pilot-based machine can never behave as a System Interface Unit.

```
PhoneNet.InvalidLineNumber: ERROR;
```

PhoneNet.**InvalidLineNumber** may be raised by procedures that take a **lineNumber** as their only argument; for example, PhoneNet.Destroy, PhoneAdoption.AdoptForNS and PhoneAdoption.DisownFromNS. In general, the error means that no driver associated with the specifed line number currently exists in the system. Since the current implementation only permits creating a driver with a **lineNumber** of zero, **InvalidLineNumber** indicates that there are no active data link drivers of type phonenet.

6.9.1.3 Procedures

```
PhoneNet.Initialize: PROCEDURE [
    lineNumber: CARDINAL, channel: RS232C.ChannelHandle,
    commParams: RS232C.CommParamHandle, negotiationMode: Negotiation,
    hardwareStatsAvailable: BOOLEAN, clientData: LONG UNSPECIFIED ← 0,
    ourEntityClass: EntityClass,
    clientHostNumber: System.HostNumber ← System.nullHostNumber];
```

Initialize is the main procedure to initialize the data link layer. The procedure will associate an RS232C line number (**lineNumber**) with that data link. Further references to the association are via the line number specified.

Returning from PhoneNet.Initialize does not imply that the data link is ready to support higher level protocols. It does link the device to the list of network supporting devices. When the physical medium becomes available and the establishment negotiation succeeds, the link will then become available for supporting higher level protocols.

lineNumber is the logical line number associated with the RS232C line. The current configuration will only support a single line, line number zero. Attempts to specify a line number other than zero will result in the error **Phonenet.Unsupported** being raised.

channel is the handle returned from the **RS232C.Create** call that was used to reserve the channel for exclusive use by the PhoneNet configuration (See §6.5.3.3). The channel must be reserved with a **preemptMe = never**.

commParams points to values in the record that define the characteristics of the RS232C line. Only the half/full duplex attribute and the line speed are applicable. The line speed is used only as an initial value and updated by empirical measurement after the connection is established.

negotiationMode specifies the active/passive attribute of the data link protocol negotiation. However, the parameter is ignored, since the norm is that both parties elect to be active, and if both parties elect to be passive, then nothing will happen.

hardwareStatsAvailable applies to direct contacts through the driver to a CIU. It is ignored in this implementation. **clientData** is also ignored.

ourEntityClass is the entity class of the client requesting the connection. If the entity class specified is **siu**, then the error **PhoneNet.IllegalEntityClass** is raised immediately. If the protocol negotiation discovers a remote entity that is in conflict with the client-specified value, then the connection will not be established.

clientHostNumber is ignored.

PhoneNet.Destroy: PROCEDURE [lineNumber: CARDINAL];

Destroy undoes the effect of the **PhoneNet.Create**. It causes the configuration to stop all processes and remove itself from the network driver chain maintained by the system. If higher level protocols are currently using the data link driver, then those protocol families will be notified via their state-changed procedures about the pending removal of the service.

lineNumber is the line number associated with the particular instance of the driver. Since **PhoneNet.Create** permits only line number zero, it is expected that the value of **lineNumber** will also be zero.

If no data link driver is associated with the line number specified, then **PhoneNet.InvalidLineNumber** is raised.

6.9.2 PhoneAdoption

PhoneAdoption: DEFINITIONS = ...

PhoneAdoption is used to "adopt" the XNS protocol family to a specified data link layer driver.

6.9.2.1 Errors

PhoneAdoption.InvalidLineNumber: ERROR;

InvalidLineNumber is raised if the **lineNumber** argument of either of the two procedures (described below) is invalid.

6.9.2.2 Procedures

PhoneAdoption.**AdoptForNS**: PROCEDURE [**lineNumber**: CARDINAL];

AdoptForNS establishes the linkage between the data link layer driver specified by the argument **lineNumber** and the XNS protocol family. Once called and as long as the data link is available, the referenced device will be usable for all networking functions. The procedure assumes that the XNS protocol family has been started.

If the line number specified does not represent an existing data link driver, then PhoneAdoption.**InvalidLineNumber** is raised.

PhoneAdoption.**DisownFromNS**: PROCEDURE [**lineNumber**: CARDINAL];

DisownFromNS undoes the effect of **AdoptForNS**. Once called, the particular data link driver will no longer process XNS network packets.

If the line number specified does not represent an existing data link driver, then PhoneAdoption.**InvalidLineNumber** is raised. Duplicate calls to this procedure are treated as no-ops.

7.

Editing and Formatting

7

Editing and Formatting

This chapter describes the facilities, usually Common Software packages, that are concerned primarily with formatting and editing. These facilities include an interface that defines some common ASCII characters, a package for converting between some common Mesa types and strings, string manipulation procedures, operations for converting between strings and Pilot's internal form of time, a Pilot byte stream implementation, and a sorting function.

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 when editing of the particular item has been completed. This procedure is 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:TYPE = 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, then 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 **char**.

```
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 edits the number parameter as follows: the number is edited in base **base** in a field **columns** wide (zero means use as many as needed). If **zerofill** is **TRUE**, then the extra columns are filled with zeros; otherwise spaces are used. If **unsigned** is **TRUE**, then 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 88 09:23:29 PDT (Wednesday)
full:	27-Jul-88 9:23:29 PDT
dateTime:	27-Jul-88 9:23:29
noSeconds:	27-Jul-88 9:23
dateOnly:	27-Jul-88

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 is 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 StringsImplA.bcd and StringsImplB.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 is then restarted as if **ns** had been the original argument. If **StringBoundsFault** is resumed with the value **NIL**, the procedure that raised the signal fills in the original string with as many characters as will fit.

7.3.3 String operations

```
String.WordsForString: PROCEDURE [nchars: CARDINAL] RETURNS [CARDINAL];
```

WordsForString 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 case, 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; **s.maxlength** is unchanged. If **s** = NIL, then **AppendChar** has no effect.

```
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. If either **to** or **from** is NIL, then **AppendString** has no effect.

```
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. If either **to** or **from** is NIL, then **AppendSubString** has no effect.

```
String.Copy: PROCEDURE [to, from: LONG STRING,];
```

Copy sets the length of **to** to zero and then appends **from** to **to**. If either **to** or **from** is NIL, then **Copy** has no effect.

```
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.maxlength** is unchanged.

```
String.Empty: PROCEDURE [s: LONG STRING,] RETURNS [BOOLEAN];
```

Empty returns TRUE if **s** is NIL or if **s.length** is 0 ; otherwise, FALSE is returned.

```
String.Equal: PROCEDURE [s1, s2: LONG STRING] RETURNS [BOOLEAN];
```

Equal returns TRUE if **s1** and **s2** contain exactly the same characters or if both **s1** and **s2** are NIL.

```
String.Equivalent: PROCEDURE [s1, s2: LONG STRING] RETURNS [BOOLEAN];
```

Equivalent returns TRUE if **s1** and **s2** contain the same characters except for case shifts or if both **s1** and **s2** are NIL. 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];
```

Length returns zero if **s** is NIL; otherwise, **s.length** is returned.

7.3.3.1 String operations that handle numbers

```
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. Note that *baseNumber* is the only portion of the number that must be provided to form a valid number.

The value returned is ± *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, then *radix* is 8; if the string has a 'D or 'd, then *radix* is 10; otherwise, *radix* is **radix**. The number *scaleFactor* is always expressed in radix 10.

- **String.InvalidNumber** is raised if **s** does not have a valid form, is equal to NIL, **s.length** = 0, values of radix are not in the range of 1 to 10, or basenumber is not provided. 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 ← 10]
  RETURNS [LONG UNSPECIFIED];
```

StringToLongNumber is analogous to **StringToNumber**, except that it returns a LONG UNSPECIFIED instead of an UNSPECIFIED.

```
String.AppendNumber: PROCEDURE [s: LONG STRING, n, radix: CARDINAL ← 10];
```

AppendNumber converts the value of **n** to text using **radix** and appends it to **s**. **radix** should be in the interval **[2..36]**. If **s** = NIL, then **AppendNumber** has no effect.

```
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 is supplied, as appropriate. If **s** = NIL, then **AppendDecimal** has no effect.

```
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. If **s** = NIL, then **AppendOctal** has no effect.

```
String.AppendLongNumber: PROCEDURE [s: LONG STRING, n: LONG UNSPECIFIED,
  radix: CARDINAL ← 10];
```

AppendLongNumber is analogous to **AppendNumber**.

```
String.AppendLongDecimal: PROCEDURE [s: LONG STRING, n: LONG INTEGER];
```

AppendLongDecimal is analogous to **AppendDecimal**.

7.3.3.2 String operations that allocate storage

```
String.MakeString: PROCEDURE [z: UNCOUNTED ZONE, maxLength: CARDINAL]
  RETURNS [LONG STRING];
```

MakeString returns a string large enough to contain **maxLength** characters, allocated from the zone **z**.

```
String.MakeMDSString: PROCEDURE [z: MDSZone, maxLength: CARDINAL] RETURNS [STRING];
```

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];
```

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];
```

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];
```

AppendCharAndGrow 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↑**. If **to↑** is **NIL**, then automatic expansion occurs.

```
String.AppendExtensionIfNeeded: PROCEDURE [
  to: LONG POINTER TO LONG STRING, extension: LONG STRING, z: UNCOUNTED ZONE]
  RETURNS [BOOLEAN];
```

AppendExtensionIfNeeded 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. **AppendExtensionIfNeeded** returns **FALSE** if the extension was not added or if **to↑** is **NIL**.

```
String.AppendStringAndGrow: PROCEDURE [to: LONG POINTER TO LONG STRING,
  from: LONG STRING, z: UNCOUNTED ZONE, extra: CARDINAL ← 0];
```

AppendStringAndGrow 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, then 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↑**. If **from** is **NIL**, then **AppendStringAndGrow** has no effect. If **to↑** is **NIL**, then automatic expansion occurs.

```
String.CopyToNewString:
  PROCEDURE [s: LONG STRING, z: UNCOUNTED ZONE, longer: CARDINAL ← 0]
  RETURNS [newS: LONG STRING];
```

CopyToNewString 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, then **newS** will be **NIL**.

```
String.ExpandString:
  PROCEDURE [s: LONG POINTER TO LONG STRING, longer: CARDINAL, z: UNCOUNTED ZONE];
```

ExpandString expands a string by **longer** characters. **s** must point to a **STRING** allocated from zone **z**. If **s↑** is **NIL**, then a new string will be allocated.

```
String.Replace:
  PROCEDURE [to: LONG POINTER TO LONG STRING, from: LONG STRING, z: UNCOUNTED ZONE];
```

Replace 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**, then **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

The Time 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.

The current time and date is kept is 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 is an absolute value, not a relative value; that is, **year** = 1968 means the year 1968, and **year** = 0 corresponds to year 0. For **month**, January is numbered 0, etc. Days of the month have their natural assignments. For **weekday**, Monday is numbered 0.

dst should be set to **TRUE** if daylight saving time is in effect; otherwise **dst** is set to **FALSE**. See **Time.Pack** and **Time.Append** for further clarification.

zone indicates time zones.

```
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, then **Time.Invalid** is raised. If the local time parameters are not available to Pilot and **ltp** is defaulted to **UseSystem**, then **System.LocalTimeParametersUnknown** is raised.

Caution: In UtilityPilot, the client must ensure that the processor clock is set correctly and that the local time parameters are set. **System.SetLocalTimeParameters** must be called before using **Unpack**.

```
Time.Pack: PROCEDURE [unpacked: Time.Unpacked,useSystemLTP: BOOLEAN ← TRUE]
  RETURNS [time: System.GreenwichMeanTime];
```

Pack converts an **Unpacked** into the Pilot-standard **GreenwichMeanTime**. If **UseSystemLTP** is set to **TRUE**, then Pilot's local time parameters are used; if **FALSE**, then **unpacked.zone** is used.

Pack uses **unpacked.dst** and the time parameters (**unpacked.zone** or **local**) to determine whether daylight saving time is in effect.

If the local time parameters are not available to Pilot, then **System.LocalTimeParametersUnknown** is raised.

```
Time.Append: PROCEDURE [s: LONG STRING, unpacked: Time.Unpacked,
  zone: BOOLEAN ← FALSE, zoneStandard: Time.TimeZoneStandard ← ANSI];
```

Append appends the time in human readable form to **s**. It adds the time zone if **zone** is **TRUE**. **Append** handles daylight saving time if both **zone** and **unpacked.dst** are **TRUE**.

```
Time.AppendCurrent: PROCEDURE [s: LONG STRING, zone: BOOLEAN ← FALSE,
  ltp:Time.LTP ← Time.useSystem, zoneStandard: TimeZoneStandard ← ANSI];
```

AppendCurrent is equivalent to Time.Append[s, Time.Unpack[Time.defaultTime, ltp], zone, zoneStandard].

7.5 Sorting

QuickSort provides one sorting function, Sort. This routine requires the client to provide a pointer to the data to be sorted. In addition, client implementation must provide the call-back procedures for comparing entries of the data and for swapping entries of the data.

```
QuickSort.Index: TYPE = INTEGER;

QuickSort.ClientData: TYPE = LONG POINTER;

QuickSort.Comparison: TYPE = {smaller, same, bigger};

QuickSort.CompareProc: TYPE = PROCEDURE [
  one, two: QuickSort.Index, data: QuickSort.ClientData]
  RETURNS [Comparison];

QuickSort.SwapProc: TYPE = PROCEDURE [
  one, two: QuickSort.Index, data: QuickSort.ClientData];

QuickSort.Sort: PROC [
  min, max: Index, compare: QuickSort.CompareProc,
  swap: QuickSort.SwapProc, data: QuickSort.ClientData];
```

QuickSort.Sort, upon completion, will have ordered the specified entries in data. data is a long pointer to an unspecified structure. The structure will be altered to contain the final sorted structure. min is the starting entry and max the ending entry. compare and swap are both call-back procedures the client must implement for comparing and swapping two entries in the structure being sorted.

Note: QuickSort.CompareProc assumes one is xxx than two, where xxx is smaller, same, or bigger.

8.

System Generation and Initialization

8

System Generation and Initialization

This section describes in general terms the organization of Pilot, Pilot-related components, and various aspects of system initialization. The following topics are addressed:

- the components of a Pilot release
- 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

Seven kinds of software components in a release of Pilot are 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 detail.)

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 Tajo, an executive and environment for general purpose programming; Othello, the Pilot disk and volume utility (not supported in Pilot 14.0), Installer, another disk and volume utility used for installing product software, and Sword, the debugger run in Tajo.

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: Pilot components may export a number of interfaces which are not documented in this manual. These interfaces 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.* Such interfaces 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 Loader.bcd. This is especially appropriate if the same boot file can load many 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 or its containing physical volume, as Pilot uses these 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 procedure is the only one imported by Pilot from the client system.

8.3 Volume initialization

```
FormatPilotDisk: DEFINITIONS...;

OthelloOps: DEFINITIONS...;
```

The steps in initializing a disk for use as a Pilot volume are as follows:

- 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.

Formatting is normally done by EIDisk, the disk diagnostic. All other initialization is done by Othello, the development disk utility (not supported in Pilot 14.0), or the Installer, the product installation and initialization utility. 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 action 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: A characteristic of rigid disks is that a disk is expected to have some unusable pages at the time of manufacture, but 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 may be copied onto it.

Finally, a disk may need to be prepared for booting. Typically, four kinds of files 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 resides 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 resides in a file on a logical volume, as do the germ and boot file. The microcode, germ, and boot file are not all in one file, or even necessarily on the same volume.

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 the 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,
  returnOnUserAbort: BOOLEAN ← FALSE,
  signalPassDone: BOOLEAN ← FALSE];

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];

FormatPilotDisk.PassesLeft: TYPE = CARDINAL;

FormatPilotDisk.PassDone: SIGNAL[passesLeft:PassesLeft];
```

```
FormatPilotDisk.SetUserAbort: PROCEDURE;
```

Format formats **count** pages of the disk **h** starting at page **firstPage**. If a problem occurs when verifying headers, labels, or data, then **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, then **BadPage** is raised. If **h** does not denote a Pilot disk drive, then **NotAPilotDisk** will be raised. If **h** denotes a drive other than the SA4000, then 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** is raised. If the volume is online (i.e., asserted to be a Pilot volume), then **PhysicalVolume.Error[alreadyAsserted]** is raised.

returnOnUserAbort indicates whether a user abort should be checked for early return during format.

signalPassDone indicates whether a **PassDone** should be raised on completion of each pass. **PassDone** must be **RESUME**d for proper cleanup.

FormatBootMicrocodeArea formats the area of the disk on **h** where microcode will reside. See the previous paragraph for description of other parameters and errors raised.

SetUserAbort sets an indicator that the user wishes to abort a **Format** in progress. **Format** checks the indicator if it was called with **returnOnUserAbort** set to **TRUE**. For formats to be abortable, **returnOnUserAbort** must be in effect. **SetUserAbort** has no effect if a **Format** is not in progress.

```
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, then 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 raised by any procedures that previously raised **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, then the error **NotAPilotDisk** is raised. If the volume is online, then **PhysicalVolume.Error[alreadyAsserted]** is raised.

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
- germs, which contain the bootstrap loader
- microcode files, which contain the Mesa emulator for a given machine

Boot files, germs, and microcode files must be *installed*; that is, made known to Pilot, the germ, and microcode. The **FormatPilotDisk** and **OthelloOps** interfaces provide facilities for dealing with boot files, germs, and microcode. The **TemporaryBooting** interface provides the means of actually invoking a boot file.

Note: Installing boot file, germ, and microcode files on a floppy disk is not directly supported by the current version of Pilot. They may be installed using the utility program `MakeDLionBootFloppyTool` or `MakeDoveBootFloppyTool`. Refer to Chapter 22 of the *XDE 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. Microcode 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, then **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, then **getPage** is called until **NIL** is returned but the data is ignored. **emptyFile** indicates that the microcode file was empty; that is, **getPage** returned **NIL** the first time that it was called.

If the first page of the microcode is bad, then **firstPageBad** is raised. If some page of the disk reserved for the boot microcode is found to be unusable, then **flakeyPageFound** is raised, indicating a problem with the disk. If an attempt is made to install too large a microcode file, then **microcodeTooBig** is raised. The error **other** is raised if the installation failed in some other way. If **h** does not denote a pilot disk drive, then the error **FormatPilotDisk.NotAPilotDisk** 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};
```

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. **pilot** 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`. **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;
```

MakeBootable 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 writeable and permanent and the logical volume that contains it must be open.

If **file** is unknown to Pilot, either **File.Unknown** or **Volume.Unknown** is raised. If the specified boot file is not compatible with the version of Pilot doing the **MakeBootable**, then **InvalidVersion** is 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**, **Volume.NotOnline**, **Volume.NeedsScavenging**, **Volume.ReadOnly**, and **File.MissingPages**.

Before changing the size of a file that has been made bootable, invoke the following operation:

```
OthelloOps.MakeUnbootable: PROCEDURE [file: File.File,
  type: OthelloOps.BootFileType, firstPage: File.PageNumber];
```

The same restrictions as for **MakeBootable** apply. **file** may be deleted without first invoking **MakeUnbootable**.

A default boot file of each type may be associated with every logical and physical volume. These boot files may be set and information about them obtained by invoking the following 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, then either **File.Unknown** or **Volume.Unknown** is raised.

The information set by the above 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; that is, contained on a physical volume that is known to Pilot. If the physical volume is only partially online, then **Volume.NotOnline** is raised. If the **lvID** is not open, then **Volume.NotOpen** will be raised. **Volume.NeedsScavenging** and **Volume.ReadOnly** may also be raised. If **lvID** is unknown to Pilot, then **Volume.Unknown** is

raised. If **pvID** is unknown to Pilot, then **PhysicalVolume.Error[physicalVolumeUnknown]** is raised.

Pilot can be told to remove a logical or physical volume's default boot file association for a file 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** is raised. Physical volume **pvID** must be on-line or **PhysicalVolume.Error[physicalVolumeUnknown]** is raised.

Every boot file of type **pilot** can have an explicit pointer to a debugger for that boot file; that is, 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,
  cantWriteBootFile, notInitialBootFile, cantFindStartListHeader,
  startListHeaderHasBadVersion, other, noDebugger,};
```

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 **debugger** is unknown to Pilot, then **Volume.Unknown** is raised.

If **nullBootFile** is returned, then **debuggeeFile** is either unknown or the volume on which it resides is unknown, not online, or not open. If Pilot is unable to modify the boot file denoted by **debuggeeFile**, then **cantWriteBootFile** is returned.

The boot file denoted by **debuggeeFile** must not be a restart file, since such files cannot have their debugger pointers set. If the file is a restart file, then **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, then 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), the operation returns **other**. If no installed debugger can be found on **debugger**, then **noDebugger** is returned.

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,
  firstLVPageNumber: 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];
```

GetNextSubVolume is a stateless enumerator and begins and ends with **nullSubVolume**. If **thisSv** cannot be found on **pvID**, then **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, then **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** is raised. **Volume.Unknown**, **Volume.ReadOnly**, **Volume.NotOnline**, and **Volume.NeedsScavenging** may be raised by this procedure.

The number of pages available on a storage device on a given drive is obtained by

```
OthelloOps.GetDriveSize: PROCEDURE [h: PhysicalVolume.Handle]
  RETURNS [nPages: LONG CARDINAL];
```

A character string denoting which switches should be down when booting a boot file can be converted into a **System.Switches** by the following operation.

```
OthelloOps.DecodeSwitches: PROCEDURE [switchString: LONG STRING]
  RETURNS [switches: System.Switches];
```

```
OthelloOps.BadSwitches: ERROR;
```

The semantics of the switch string passed to **DecodeSwitches** are 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, then **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.SetGetSwitchesSuccess, System.Switches];

OthelloOps.SetSwitches: PROCEDURE [file: File.File,
  firstPage: File.PageNumber, switches: System.Switches]
  RETURNS [OthelloOps.SetGetSwitchesSuccess];
```

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 cannot be set from the Ethernet, or is not set to a time less than or equal to the boot file's expiration date, then Pilot will refuse to boot and will hang with maintenance panel code 937. The logical volume on which the boot file resides must have been opened in order to invoke these procedures.

Note: The boot file may be booted with the \200 switch, **PilotSwitchesExtraExtraExtras.continueBootingIfNoTimeServer**. Files created during such a boot session will have undefined timestamps.

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 switches are set and retrieved by **SetSwitches** and **GetSwitches**. When a boot file is booted, Pilot sets the system switches to the value passed to it by the client booting program. If no switches are passed, or if they are equal to **System.defaultSwitches**, Pilot sets them to the boot file's default switches.

The following operations 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 call 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 networks can be interconnected 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, no special communication initialization is 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**.

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 volume 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, although 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, then the file must be installed using the operation **InstallVolumeBootFile**. A subsequent **BootFromVolume** operation applied to that volume (e.g., by the Installer, or by a client program analogous to the Installer) causes 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**).

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; for example, an Installer'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 following operation.

```
TemporaryBooting.MakeBootable: PROCEDURE [file: File.File,
  firstPage: File.PageNumber ← 0];

TemporaryBooting.InvalidParameters: ERROR;

TemporaryBooting.InvalidVersion: ERROR;
```

The parameter **firstPage** specifies the first page containing the information produced by MakeBoot; for example, the page following the client leader pages, or zero if no leader pages are present.

If the file does not contain a valid Pilot boot file starting at **firstPage**, then **InvalidParameters** is raised.

If the file being made bootable has an invalid version, then **InvalidVersion** is raised. The file is made bootable before this error is raised so that boot files which are incompatible with the current version of Pilot can be installed by the current version.

8.5.4 Installing a boot file

To establish a file as the boot file of a particular logical volume, use the following 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 following 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 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 a new version of a boot file must be installed 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 co-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, then 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 valid only 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** is 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** is 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: LONG POINTER ← 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; for example, 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; that is, the one outloaded in 1).

9.

The Backstop

9

The Backstop

A *backstop* is a system for recording information about ailing 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 VMMapLogImpl.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 is 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 debuggee system by calling **Backstop.Proceed**.

```
Backstop.Proceed: PROCEDURE [boot: Volume.ID];
```

boot specifies the volume to be restarted. If **boot** is **Volume.nullID**, then the physical volume is 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 situation could occur if a new version of a debuggee 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.

Note: The size of the volume containing the backstop bootfile can be calculated by adding the following numbers:

size of backstop bootfile +
size of debuggee's outload file +
size of backstop's data +
size of Pilot's backing file +
size of Pilot's internal data structures

Virtual memory size and real memory size alter the outload size, backing file size, and Pilot's internal structures.

9.1.3 Logging errors

Procedures described 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 debuggee 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 debuggee'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, then 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 then to 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** returns a handle for the local frame of the most recently called procedure of the process. Passing a handle so obtained returns a handle for the local frame of the next-most-recently-called procedure. Passing the handle of the root frame of the process returns **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 debuggee 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.NotErrorEntry: ERROR;

BackstopNub.Handle: LONG POINTER TO BackstopNub.ErrorEntry;

BackstopNub.ErrorEntry: TYPE = MACHINE DEPENDENT RECORD[
  globalFrame(0): BackstopNub.GlobalFrame,
  pc(2): BackstopNub.PC,
  time(3): System.GreenwichMeanTime,
  options(1): SELECT error(5): BackstopNub.ErrorType FROM
    signal => [signal(6): BackstopNub.Signal,
      msg(7): BackstopNub.SignalMsg,
      stk(8): ARRAY [0..stackSize) OF UNSPECIFIED],
    call => [msg(6): StringBody],
    unused => [],
    interrupt => [],
    addressfault => [faultedProcess(6): BackstopNub.PSBIndex],
    writeprotectfault => [faultedProcess(6): BackstopNub.PSBIndex],
    other => [reason(6): BackstopNub.SwapReason],
    bug => [bugType(6): CARDINAL],
  ENDCASE];

BackstopNub.GlobalFrame: TYPE [2];

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, then they raise **NotErrorEntry**.

No facilities are provided for reading process or frame entries.

10.

Online Diagnostics

10

Online Diagnostics

10.1 Communication diagnostics

CommOnlineDiagnostics: DEFINITIONS . . . ;

The **CommOnlineDiagnostics** interface is used by clients of communications online diagnostics. It includes procedures for running echo tests, gathering Ethernet statistics, and testing RS232C and dialer facilities. All tests may be run on any host machine exporting the communications online diagnostics server.

CommOnlineDiagnostics is implemented by two configurations: `CommDiagClient.bcd` and `CommDiagServer.bcd`. `CommDiagClient` provides the client access to the diagnostic functions which are implemented in `CommDiagServer.bcd`. Consequently, systems that only require remote access to another's diagnostic capability need not include `CommDiagServer.bcd`.

Note: Pilot releases preceding Pilot 14.0 included a backward-compatibility feature of **CommOnlineDiagnostics**. This feature has been removed; clients must now bind in `CommDiagClient.bcd` in place of `BackCompatibleDiag.bcd`.

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, then the error **CommError** is raised with a reason of **noSuchDiagnostic**.

CommOnlineDiagnostics.CommError: ERROR [reason:CommOnlineDiagnostics.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, as follows.

transmissionMediumProblem
: A problem with the physical device occurred.

noAnswerOrBusy
: The remote end did not answer or was already busy. Applies to circuit-oriented media only.

noRouteToSystemElement
: The network on which the diagnostic is to be run is not reachable at this time.

remoteSystemElementNotResponding
: The machine specified in the host parameter of the diagnostic is not responding.

tooManyConnections
: 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 Testing Ethernet echo

```
CommOnlineDiagnostics.EchoDiagHandle: TYPE = LONG POINTER TO
    CommOnlineDiagnostics.EchoDiagObject;

CommOnlineDiagnostics.EchoDiagObject: TYPE;

CommOnlineDiagnostics.StartEchoUser: PROC [
    targetSystemElement: System.NetworkAddress,
    echoParams: CommOnlineDiagnostics.EchoParams,
```

```
eventReporter: CommOnlineDiagnostics.EventReporter ← NIL,
host: System.NetworkAddress ← System.nullNetworkAddress RETURNS
[dH:CommOnlineDiagnostics.EchoDiagHandle]
```

StartEchoUser starts the echo test. Multiple echo tests may be run on the same host.

targetSystemElement is the machine that is to be the echo server. **echoParams** are the client-specified parameters for the test to be run.

eventReporter is the client-supplied procedure that is called whenever an interesting event occurs; for example, 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, then he should set the **eventReporter** to **NIL** or let it default to **NIL**.

host is the network address of the machine that is to be used as the echo user. The **dH** returned from **StartEchoUser** is the handle to be used to retrieve the echo test results.

```
CommOnlineDiagnostics.GetEchoResults: PROC [
    dH:CommOnlineDiagnostics.EchoDiagHandle;
    host: System.NetworkAddress,
    stopIt: BOOLEAN]
    RETURNS [totalsSinceStart: CommOnlineDiagnostics.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 is the handle that identifies the test from which to retrieve the results. If the procedure is called with **stopIt** equal to **TRUE**, then the test returns the results and then stops. If the client wishes to obtain intermediate results of an echo test, he may call **GetEchoResults** with **stopIt** equal to **FALSE**; the current counters are returned, and the test continues to run. This feature is useful for real-time feedback at time intervals chosen by the client. **host** is the network address of the machine that is the echo user. **totalsSinceStart** are the actual results of the echo user test. **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 indicates that the echo request/response exchange was successfully completed. **late** indicates the response to the echo request arrived late. **timeout** occurs when no response is received for the echo request sent; the test times out and sends the next echo request. **badDataGoodCRC** indicates that the echo response was received without a CRC error, but some data bytes of the packet do not match the expected pattern. If the test is varying the length the the data in the echo request, then an event of **sizeChange** occurs when the size goes from the maximum back to the minimum. **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): CommOnlineDiagnostics.WordsInPacket ← incrWords,
    constant(6): CARDINAL ← 125252B,
    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 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, then the test runs until stopped by **GetEchoResults** or by the "dead man's" switch.

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, either to reset the timeout and continue echoing or to stop the test and collect the results. If **GetEchoResults** is not called within this time, then the test enters an idle state. It must still be terminated via **GetEchoResults**.

minPacketSizeInBytes is used to specify the minimum number of data bytes to send in the echo request. **maxPacketSizeInBytes** is used to specify the maximum number of data bytes to send in the echo request. If the specified size is larger then the maximum data bytes allowed in an echo packet, then the packet is truncated to the maximum allowed. If **maxPacketSizeInBytes** is equal to **minPacketSizeInBytes**, then the test sends constant length.echo packets; otherwise, the size will range from the minimum specified to the maximum.

wordContents specifies what the data words in the packet will contain. The data word constant is set using the **constant** parameter. This parameter is used by the test only if the **wordContents** parameter is **allConstant**.

If **waitForResponse** is **TRUE**, then test does not send an echo request until the reply to the previous request is received or until a timeout occurs.

minMsecsBetweenPackets allows the client to set the approximate interval between echo requests by specifying **minMsecsBetweenPackets**.

checkContents allows the client to have the test verify each word in the echo response packet by specifying **checkContents**.

showMpCode 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 is the total number of echo packets that the echo user attempted to send, regardless of the number of valid responses received. **successes** is the total number of successful echo request/response exchanges. **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** is the number of echo responses that arrived at the echo user late. **unexpected** is the number of unexpected packets that were received on the echo socket. The **avgDelayInMsecs** is the average time between successful echo request/response exchanges.

okButDribble, badAlignmentButOkCrc, packetTooLong, overrun, idleInput, tooManyCollisions, lateCollisions, underrun, stuckOutput are the Ethernet statistics for the number of packets found with the specified problem.

Note: These last statistics are only valid for echo tests using Ethernets and should be ignored for other media.

```
CommOnlineDiagnostics.EtherDiagError:ERROR [
    reason:CommOnlineDiagnostics.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. If **GetEchoResults** is called when no echo test is running on the host machine, then an error is raised with a reason of **echoUserNotThere**. **noMoreNets** is raised by **GetEthernetStats** and indicates that no net exists with the **physicalOrder** specified. **invalidHandle** indicates that the client attempted to call **GetEchoResults** with a handle that is **NIL** or one that was deleted via setting the **stopIt** boolean.

```
CommOnlineDiagnostics.EventReporter: TYPE = PROC [
    event: CommOnlineDiagnostics.EchoEvent,
    dH:CommOnlineDiagnostics.EchoDiagHandle];
```

Clients wishing to be notified at every echo event can implement an **EventReporter** procedure and register it as an argument of **CommOnlineDiagnostics.StartEchoUser**. The procedure will be called at each interesting event, as defined by the enumerated type **CommOnlineDiagnostics.EchoEvent**. The type of interesting event is passed as an argument to the client-implemented procedure, as is **CommOnlineDiagnostics.EchoDiagHandle**. The latter

argument identifies the particular instance created by the call to CommOnlineDiagnostics.StartEchoUser.

```
CommOnlineDiagnostics.Histogram: TYPE =
    LONG DESCRIPTOR FOR ARRAY CARDINAL OFCommOnlineDiagnostics.Detail;

CommOnlineDiagnostics.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 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. 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** means the words in the packet will contain zeros. **all1s** means the words in the packet will contain ones. **incrWords** means each word of the packet will be incremented, starting with the first word equal to one. **allConstant** means the words in the packet will be a client-specified constant. **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 CommOnlineDiagnostics.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, as follows.

echoServerPkts
: The number of packets that the machine has echoed.

EchoServerBytes
: The number of bytes that the machine has echoed.

packetsRecv
: The total number of packets that have been successfully received, including echo packets.

wordsRecv
: The total number of words that have been successfully received, including words in echo packets.

packetsMissed
: The number of packets that have been dropped for lack of buffering.

badRecvStatus
The total number of packets that were not successfully received.

okButDribble
The number of packets that were successfully received, but had extra bits at the end.

badCrc The number of packets that were received with bad CRCs.

badAlignmentButOkCrc
The number of packets that were received with correct CRCs but did not end on byte boundries.

crcAndBadAlignment
The number of packets that were received with bad CRCs and did not end on byte boundries.

packetTooLong
The number of packets received that were longer than the maximum internet size of 576 bytes.

overrun
The microcode cannot take bits out of the input silo fast enough to keep up with the bits coming in off the wire.

idleInput
The number of times the machine did not receive input from the Ethernet for at least 40 seconds.

packetsSent
The total number of packets that have been successfully sent, including echo packets.

wordsSent
The total number of words sent, including those in echo packets.

badSendStatus
The total number of packets that were not successfully sent.

tooManyCollisions
The number of packets that were never sent after sixteen attempts failed because of collisions.

lateCollisions
The number of packets that have had collisions occurring in the later part of the packet (after bit 512).

underrun
The microcode cannot put bits into the output silo fast enough to maintain the 10 Mbit rate.

stuckOutput
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 item 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 is the number of the device on the device chain; the primary network has a physical order of one. **host** is the machine from which to obtain the statistics. **stats** returns the current Ethernet statistics.

time is the time when 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 is the network address of the machine from which to collect the echo counters. **packets** is the number of echoed packets. **bytes** is the total number of bytes the server has echoed.

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 Testing RS232C

RS232C testing consists of running a loopback test that exercises and verifies the data-transmission/reception features of the RS232C channel. As clients are required to set some of the channel characteristics, they should be familiar with the EIA RS232C standard.

```
CommOnlineDiagnostics.StartRS232CTest: PROC [
    rs232cParams: CommOnlineDiagnostics.RS232CParams,
    setDiagnosticLine: CommOnlineDiagnostics.SetDiagnosticLine ← NIL,
    writeMsg: CommOnlineDiagnostics.WriteMsg ← NIL,
    modemChange: CommOnlineDiagnostics.ModemChange ← NIL,
    host: System.NetworkAddress←System.nullNetworkAddress]
    RETURNS [dH: CommOnlineDiagnostics.RS232CHandle];
```

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 results in the error **RS232CDiagError** with the code **channelInUse**.

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 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 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 is the network address of the machine on which to run the diagnostic.

dH is handle returned from **StartRS232CTest** and passed to the **GetRS232CResults** procedure; it provides support for RS232C diagnostics on the multiport board. Clients must use the handle identifying a particular test to obtain the test results.

```
CommOnlineDiagnostics.GetRS232CResults: PROC [
    dH: CommOnlineDiagnostics.RS232CHandle,
    stopIt: BOOLEAN,
    host: System.NetworkAddress ← System.nullNetworkAddress]
    RETURNS [counters: CommOnlineDiagnostics.CountType];
```

After starting the loopback test, the client obtains the results of the test by calling **GetRS232CResults**. The test is implemented with a "dead man's 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.

If the procedure is called with **stopIt** equal to **TRUE**, then the test returns the results and terminates. If the client wishes to obtain intermediate results of an echo test, he may call **GetRS232CResults** with **stopIt** equal to **FALSE**; the current counters are returned, and the loopback test continues to run. **host** is the network address of the machine on which to run the loopback test. **counters** is the current results of the loopback test.

```
CommOnlineDiagnostics.RS232CDiagError: ERROR [
    reason: CommOnlineDiagnostics.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.

```
CommOnlineDiagnostics.RS232CErrorReason: TYPE = {aborted, noHardware, noSuchLine,
    channelInUse, unimplementedFeature, invalidParameter, invalidHandle};
```

RS232CErrorReason defines the reasons for the errors raised in **RS232CDiagError**.

aborted The channel has been aborted.

noHardware
No RS232C hardware is present or the RS232C channel code has not been started.

noSuchLine
A bad RS232C line number has been specified by the client.

channelInUse
Some other process was already using the RS232C port when the client attempted to start the RS232C test.

unimplementedFeature
Used internally and should never be observed by the client.

invalidParameter
An invalid parameter was passed to the RS232C test.

invalidHandle
The client called **GetRS232CResults** with a NIL handle or a handle that was deleted via setting the **stopIt** boolean.

```
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 **StartRS232CTest**. 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 is a counter that reflects the number of successfully sent frames. **bytesSent** is the current number of bytes that have been sent. **recOk** reflects the number of successfully received frames. **bytesRec** is the current number of bytes that have been sent.

deviceError indicates the number of times data was received when no receive operation was outstanding. **dataLost** indicates the number of times that a incoming frame was too large to fit in the input buffer. **xmitErrors** indicates the number of frames that have been received with some sort of transmission error; for example, checksum error, parity error. **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** indicates the number of times the receiver has detected a missing frame from looking at the sequence numbers. **sendErrors** indicates the total number of frames that have not been successfully sent. **recErrors** indicates the total number of frames that have not been successfully received.

```
CommOnlineDiagnostics.LengthRange: TYPE = RECORD [
    low, high: [0..CommOnlineDiagnostics.maxData)];
```

defines the range of data length (in bytes) in the frames.

```
CommOnlineDiagnostics.maxData: CARDINAL = 1000;
```

maxData is the maximum number of bytes of data in a frame.

```
CommOnlineDiagnostics.ModemChange: TYPE = PROC [
    modemSignal:CommOnlineDiagnostics.ModemSignal,
    state: BOOLEAN, dH: CommOnlineDiagnostics.RS232CDiagHandle];

CommOnlineDiagnostics.ModemSignal: TYPE = {dataSetReady, clearToSend,
    carrierDetect, ringIndicator, ringHeard};
```

ModemChange is a procedure type used by the client when he wishes to be notified of a change in the state of the signals defined in **ModemSignal**.

modemSignal is the signal of interest. The possible values of **modemSignal** correspond to the circuits described in EIA Standard RS232C. The state of that circuit is conveyed in the parameter **state**. A value of **FALSE** corresponds to a low EIA value, a value of **TRUE** to a high. **dH** is the handle identifying the particular test.

```
CommOnlineDiagnostics.PatternType: TYPE = {zero, ones, oneZeroes, constant, byteIncr};
```

The **PatternType** defines the contents of the data in the frames being sent. **zero** indicates the contents will be all zeros. **ones** indicates the contents will be all ones. **oneZeroes** indicates the contents will be an alternating bit pattern (1010...). **constant** indicates the test will use a client-supplied constant in each byte of data. **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): CommOnlineDiagnostics.PatternType,
    constant(10): CARDINAL ← 0,
    dataLengths(11): CommOnlineDiagnostics.LengthRange
    clockSource(13):RS232C.clockSource,
    waitForDSR(14):BOOLEAN ← TRUE];
```

RS232CParams defines the parameters passed to the RS232C test, as follows.

testCount
: The number of frames to send/receive. If this number is set to 0, then the test runs actively loopback until stopped by the **GetRS232CResults** or by the "dead man's" switch.

safetyTOInMsecs
: 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, either to reset the timeout and continue echoing or to stop the test and collect the results.

lineSpeed
: The speed of the line. Should agree with the setting of the modem.

correspondent
: The type of system the test is to "correspond" with. The line type determines what the correspondent should be. For a line type of asynchronous, use **ttyHost**. For bit synchronous, use **nsSystemElement**. For byte synchronous, use **system6**.

lineType The type of line the channel will use.

lineNumber
The number of the RS232C line. Should normally be set to 0. Other values apply only to processors with multiple RS232C lines.

parity The parity to use during the test.

charLength
The character data length, (excluding parity, stop and start bits). Should agree with the setting on the modem.

pattern The contents for each byte of data.

constant What the data constant should be if the client has specified a pattern of constant.

dataLengths
The range of data lengths to send in the frames. If the **low** and **high** are equal, then test sends constant length data. If they are not equal, then the test first sends the frame of the right length, decrementing the length with each subsequent send.

clockSource
Determines whether the clock will be provided by the DTE (internal) or by the modem (external).

waitForDSR
Some modems in the field do not raise DSR. **waitForDSR** enables users of such modems to tell the test to start even if DSR has not come up.

```
CommOnlineDiagnostics.RS232CTestMessage: TYPE = MACHINE DEPENDENT {
    looped, sendError, recError};
```

The **RS232CTestMessage** is passed to the client-supplied procedure that is called every time a frame is sent or received. The message indicates the status of the transfer.

looped reports a successful send/receive sequence. **sendError** reports an unsuccessful send. **recError** reports an unsuccessful receive.

```
CommOnlineDiagnostics.SetDiagnosticLine: TYPE = PROC [lineNumber: CARDINAL, dH:
    CommOnlineDiagnostics.RS232CDiagHandle]
    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. **dH** is the handle identifying the particular test.

```
CommOnlineDiagnostics.WriteMsg: TYPE = PROC [
    msg;CommOnlineDiagnostics.RS232CTestMessage,
    dH:CommOnlineDiagnostics.RS232CDiagHandle];
```

WriteMsg is a procedure type used by clients when they wish real-time feedback during the RS232C test.

msg indicates the type of event that just occurred. **dH** is the handle identifying the particular test.

10.1.4 Testing the Dialer

Dialer testing verifies 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: CommOnlineDiagnostics.DialupOutcome];
```

DialupTest is called to test the dialer. The test retries 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 specifies the line number to be used and should be set to 0. Other values apply only to processors with multiple RS232C lines.

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 is the type of dialing equipment being used. **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**, as follows.

success The dialing operation was successful; that is, all the digits in the number were dialed, and control was successfully transferred to the modem.

failure The dialing operation resulted in no answer, a busy signal, or the telephone was answered by something other than a compatible modem.

aborted Not implemented.

formatError
The parameter **phoneNumber** was formatted incorrectly.

transmissionError
The transfer of the dialing information to the dialing hardware did not succeed. This outcome indicates a hardware problem.

dataLineOccupied
The telephone line to which the dialing hardware is connected is off-hook.

dialerNotPresent
Working dialer hardware is lacking.

dialingTimeout
The dialer did not respond to a request during dialing. This outcome is a hardware problem.

transferTimeout
No meaningful reply was received from the dialer following dialing the last digit. This outcome indicates a hardware problem.

otherError
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...;
```

The **OnlineDiagnostics** 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};
```

Background defines the background on the bitmap display.

```
OnlineDiagnostics.CursorArray: TYPE = ARRAY [0..16] OF WORD;
```

CursorArray 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};
```

KeyboardType 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 and is equivalent to **UserTerminal.screenHeight**. **screenWidth** defines the number of horizontal dots across the bitmap display and is equivalent to **UserTerminal.screenWidth**.

SetBackground [background: ...] sets the bitmap display background to either white or black and is equivalent to **UserTerminal.SetBackground**. If the display has a border, then clients may set the pattern to be displayed in the border by calling **SetBorder**; the procedure is 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 and is equivalent to **UserTerminal.SetMousePosition**.

SetCursorPattern sets up the bit pattern of the cursor for display on the bitmap display and is equivalent to **UserTerminal.SetCursorPattern**. **SetCursorPosition** sets the position of the cursor on the bitmap display and is equivalent to **UserTerminal.SetCursorPosition**.

keyboard is equivalent to **UserTerminal.keyboard**.

Beep emits a tone from the speaker for the given duration of time. Duration is in milliseconds. **Beep** is 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};
```

NextAction defines the next action to be taken. The procedure is 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]];
```

LFDisplayTest displays test patterns on the display. The procedure can be used as a bitmap display alignment tool.

screenHeight defines the number of horizontal lines on the bitmap display and is equivalent. to **UserTerminal.screenHeight**. **screenWidth** defines the number of horizontal dots across the bitmap display and is equivalent to **UserTerminal.screenWidth**.

SetBackground sets the bitmap display background to either white or black and is equivalent to **UserTerminal.SetBackground**. If the display has a border, then clients may set

the pattern to be displayed in the border by calling **SetBorder**; the procedure is equivalent to **UserTerminal.SetBorder**.

GetNextAction gets the next action through keyboard input from the user. See **OnlineDiagnostics.NextAction** above.

ClearDisplay turns the entire screen 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 . . . ;

The **OnlineDiagnostics** 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, kCTLC, 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};

LSMessage 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]];
```

LSAdjust allows the user to adjust the Lear Siegler display. **cancelSignal** is raised when the user enters a 'Control C' on the keyboard and is equivalent to **NSCommand.Cancel**. **GetMesaChar** gets the character entered on the keyboard by the user and is equivalent to **NSCommand.GetMesaChar**. **PutCR** outputs a carriage return to the Lear Siegler display and is equivalent to **NSCommand.PutCR[TRUE]**. **PutMessage** displays the given message on the Lear Siegler display and is 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 and is equivalent to **NSCommand.PutMesaChar.**

```
OnlineDiagnostics.LSTest: PROCEDURE [
    cancelSignal: SIGNAL,
    GetMesaChar: PROC RETURNS [CHARACTER],
    PutMessage: PROC [message: OnlineDiagnostics. LSMessage, char: CHARACTER ← 0C]];
```

LSTest allows the user to test the Lear Siegler display or equivalent.

cancelSignal is raised when the user enters a 'Control C' on the keyboard and is equivalent to **NSCommand.Cancel**. **GetMesaChar** gets the character entered on the keyboard by the

user and is equivalent to **NSCommand.GetMesaChar**. **PutMessage** displays the given message on the Lear Siegler display and is equivalent to **NSCommand.PutLine**.

Note: For the diagnostic, the default for **char** is taken.

10.4 Floppy diagnostics

OnlineDiagnostics: DEFINITIONS . . . ;

The **OnlineDiagnostics** 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, hIllgIStat, hIncrtLngth, hObser1, hObser2, hReadHead, hReadSector, hReadStat, hReady, hRecal, hRecalErr, hSector, hSectorAddr, hSectorCntErr, hSectorLgth, hSeekErr, hTimeExc, hTrack, hTrack0, hTrackAddr, hTwoSide, hWriteDelSector, hWritePro, hWriteSector, hLast,

iFirst, iBadContext, iBadLabel, iBadSector, iBadTrack0, iCheckPanel, iCIERec, iCleanDone, iCleanProgress, iErrDet, iErrNoCRCErr, 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};

FloppyMessage defines the message keys used by the Floppy diagnostic.

```
OnlineDiagnostics.FloppyReturn: TYPE = {
    deviceNotReady, notDiagDiskette, floppyFailure,noErrorFound};
```

FloppyReturn 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];
```

Field is used for Floppy Diagnostics status display.

```
OnlineDiagnostics.FieldDataType: TYPE = {
    boolean, cardinal, character, hexadecimal, hexbyte, integer, octal, string};
```

FieldDataType defines the various types of data displayed by the Floppy Diagnostics .

```
OnlineDiagnostics.FloppyWhatToDoNext: TYPE = {
    continueToNextError, loopOnThisError, displayStuff, exit};
```

FloppyWhatToDoNext defines the operator options for running Floppy Diagnostics command files.

```
OnlineDiagnostics.SingleDouble: TYPE = {single, double};
```

SingleDouble defines the number of sides and data density of a floppy diskette.

```
OnlineDiagnostics.SectorLength: TYPE = {one28, two56, five12, one024};
```

SectorLength defines the number of bytes in a sector of a floppy diskette. It is used in Floppy Diagnostics command files.

```
OnlineDiagnostics.ErrorHandling: TYPE = {
    noChecking, stopOnError, loopOnError, continueOnError};
```

ErrorHandling 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; for 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.YesOrNo: TYPE = {yes, no};

OnlineDiagnostics.GetYesOrNoProc:PROCEDURE [
    msg: OnlineDiagnostics.FloppyMessage] RETURNS [OnlineDiagnostics.YesOrNo];
```

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.

10.5 FloppyTape diagnostics

```
OnlineDiagnosticsExtraExtras: DEFINITIONS . . . ;
```

The **OnlineDiagnosticsExtraExtras** interface is used by clients of the FloppyTape Online Diagnostics. It includes procedures for running the FloppyTape Diagnostic.

```
OnlineDiagnosticsExtraExtras.FlTapeMessage: TYPE = {
wishToContinue, stillSure, ensureDriveReady, readyAndWrEnable, successComplete,
removeCartridge, uncomplete, userTerminate, testTerminated, writeProtect,
mediaProblem, hardwareProblem, driveInUse, notReady, notFormat, warningFormat,
enterTapeLabel, formatTape, verifyFormat, createDataStruc, formatFailed,
alreadyFormat, tapeLabel, retentionTape, retentFailed, insertDiagTape, note,
verifyRead, verifyPass, verifyFail, tooManySoft, hardReadError, callSupport,
scavengeTape, scavengeFailed, unSuccessRepair, successRepair, sectorToEnter,
enterToTable, logSectorFailed, readTable, countIs, badSectorTable, emptyTable,
secTableFailed, badSector, stream, track, sector, badSectorFull, extra1, extra2, extra3,
extra4, extra5};
```

FlTapeMessage defines the message keys used by the FloppyTape Diagnostic.

```
OnlineDiagnosticsExtraExtras.YesOrNo: TYPE = {yes, no};
```

Call-back procedures are used as parameters in the procedures called by clients. They are used to post messages and data to the display and to obtain input from the user. The call-back procedures are of the same style as the ones used in the floppy online diagnostics. Nine call-back procedures are listed below.

```
OnlineDiagnosticsExtraExtras.GetTextProc: TYPE = PROCEDURE [
    msg: OnlineDiagnosticsExtraExtras.FlTapeMessage,
    inputText: LONG STRING];
```

GetTextProc gets a text string from the user. The procedure call passes a message key, **msg**, and a long string, **inputText**, as arguments. The client posts the message key which requests input from the user and fills the string body with the input from the user. The string must be a Mesa string and can be up to 100 characters long.

```
OnlineDiagnosticsExtraExtras.GetNumberProc: TYPE = PROCEDURE [
    msg: OnlineDiagnosticsExtraExtras.FlTapeMessage] RETURNS [inputNumber: LONG CARDINAL];
```

GetNumberProc gets a number from the user. The procedure call passes a message key, **msg**, and the client returns **inputNumber**, a long cardinal obtained from the user.

```
OnlineDiagnosticsExtraExtras.PutMessageProc: TYPE = PROCEDURE [
    msg: OnlineDiagnosticsExtraExtras.FlTapeMessage,
    moreFollows: BOOLEAN ← FALSE];
```

PutMessageProc displays the given message to the user. The parameters passed are a message key, **msg**, and a boolean, **moreFollows**, which is used to tell the client whether to terminate this display line with a carriage return.

```
OnlineDiagnosticsExtraExtras.PutNumberProc: TYPE = PROCEDURE [
    number: LONG CARDINAL,
    width: CARDINAL ← 5,
    moreFollows: BOOLEAN ← FALSE];
```

PutNumberProc displays the given number to the user. The parameters passed are a message key, **msg**, and a boolean, **moreFollows**, which is used to tell the client whether to terminate this display line with a carriage return.

```
OnlineDiagnosticsExtraExtras.PutTextProc: TYPE = PROCEDURE [
    outputText: LONG STRING,
    moreFollows: BOOLEAN ← FALSE];
```

PutTextProc displays the given text string to the user. The parameters are a long string, **outputText**, and a boolean, **moreFollows**, which is used to tell the client not to terminate this display line with a carriage return.

```
OnlineDiagnosticsExtraExtras.PutTimeStampMessageProc: TYPE = PROCEDURE [
    msg: OnlineDiagnosticsExtraExtras.FlTapeMessage];
```

PutTimeStampMessageProc posts a time stamp along with the given message, **msg**, to the display.

```
OnlineDiagnosticsExtraExtras.GetConfirmationProc: TYPE = PROCEDURE [
    msg: OnlineDiagnosticsExtraExtras.FlTapeMessage];
```

GetConfirmationProc displays the given message, **msg**, to the user and requests confirmation. The client procedure does not return until the user types any character.

```
OnlineDiagnosticsExtraExtras.GetYesOrNoProc: TYPE = PROCEDURE [
    msg: OnlineDiagnosticsExtraExtras.FlTapeMessage] RETURNS [YesOrNo];
```

GetYesOrNoProc displays a message, **msg**, to the user and requests a Y(es) or N(o) response. An enumerated type of yes or no, **YesOrNo**, is returned by this procedure.

```
OnlineDiagnosticsExtraExtras.LookForAbortProc: TYPE = PROCEDURE[]
    RETURNS [userAbort: BOOLEAN];
```

LookForAbortProc returns a boolean of TRUE if the user wishes to abort out of a utility. The client defines which key or keys are used to indicate an abort request. When this procedure is called, the client returns immediately with a true or false response, depending on whether the user has requested an abort prior to the procedure call.

The following six procedures are provided as floppytape online utilities. The procedures use the nine call-backs as parameters.

```
OnlineDiagnosticsExtraExtras.FlTapeFormat: PROCEDURE [
    getText: OnlineDiagnosticsExtraExtras.GetTextProc,
    putMessage: OnlineDiagnosticsExtraExtras.PutMessageProc,
    putText: OnlineDiagnosticsExtraExtras.PutTextProc,
    putTimeStampMessage: OnlineDiagnosticsExtraExtras.PutTimeStampMessageProc,
    getConfirmation: OnlineDiagnosticsExtraExtras.GetConfirmationProc,
    getYesOrNo: OnlineDiagnosticsExtraExtras.GetYesOrNoProc,
    lookForAbort: OnlineDiagnosticsExtraExtras.LookForAbortProc];
```

FlTapeFormat is used to format a cartridge tape. It does a retention pass, formats the entire tape, verifies the formatted data, creates a bad sector table, enters bad sectors found during the verify pass into the table, and puts the file system data structures on the tape.

```
OnlineDiagnosticsExtraExtras.FlTapeRetention: TYPE = PROCEDURE[
    putMessage:OnlineDiagnosticsExtraExtras.PutMessageProc,
    putTimeStampMessage: OnlineDiagnosticsExtraExtras.PutTimeStampMessageProc,
    getConfirmation: OnlineDiagnosticsExtraExtras.GetConfirmationProc];
```

FlTapeRetention is used to condition (also known as retention) the tape.

```
OnlineDiagnosticsExtraExtras.FlTapeVerifyRead: TYPE = PROCEDURE[
    putMessage: OnlineDiagnosticsExtraExtras.PutMessageProc,
    putText: OnlineDiagnosticsExtraExtras.PutTextProc,
    putNumber: OnlineDiagnosticsExtraExtras.PutNumberProc,
    putTimeStampMessage: OnlineDiagnosticsExtraExtras.PutTimeStampMessageProc,
    getConfirmation: OnlineDiagnosticsExtraExtras.GetConfirmationProc,
    lookForAbort: OnlineDiagnosticsExtraExtras.LookForAbortProc,
    debugSwitch: UNSPECIFIED ← 0];
```

FlTapeVerifyRead reads two streams of data from the Offline Diagnostic tape, collects data, and provides passed and failed information to the user.

```
OnlineDiagnosticsExtraExtras.FlTapeScavenge: TYPE = PROCEDURE[
    putMessage: OnlineDiagnosticsExtraExtras.PutMessageProc,
    putTimeStampMessage: OnlineDiagnosticsExtraExtras.PutTimeStampMessageProc,
    getConfirmation: OnlineDiagnosticsExtraExtras.GetConfirmationProc,
    lookForAbort: OnlineDiagnosticsExtraExtras.LookForAbortProc];
```

FlTapeScavenge repairs a malformed tape by restoring the Pilot data structures (root page) and repairing bad marker pages.

```
OnlineDiagnosticsExtraExtras.FlTapeLogBadSector: TYPE = PROCEDURE[
    getNumber: OnlineDiagnosticsExtraExtras.GetNumberProc,
    putMessage: OnlineDiagnosticsExtraExtras.PutMessageProc,
    putTimeStampMessage: OnlineDiagnosticsExtraExtras.PutTimeStampMessageProc,
    getConfirmation: OnlineDiagnosticsExtraExtras.GetConfirmationProc,
    getYesOrNo: OnlineDiagnosticsExtraExtras.GetYesOrNoProc];
```

FlTapeLogBadSector enters a sector specified by the user into the bad sector table.

```
OnlineDiagnosticsExtraExtras.FlTapeDisplayBadSectorTable: TYPE = PROCEDURE[
    putMessage: OnlineDiagnosticsExtraExtras.PutMessageProc,
    putNumber: OnlineDiagnosticsExtraExtras.PutNumberProc,
    putTimeStampMessage: OnlineDiagnosticsExtraExtras.PutTimeStampMessageProc,
    getConfirmation: OnlineDiagnosticsExtraExtras.GetConfirmationProc,];
```

FlTapeDisplayBadSectorTable displays the contents of the bad sector table.

Appendices

A

Performance Criteria

Appendix A 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 121 pages (30,976 words), allocated as follows: code – 62; data – 37; the Mesa runtime data structures – 6; and global frames – 16. 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 as 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.

Table A.1 summarizes the memory requirements. In the table, working set sizes are given in pages. They do not include the resident.

Table A.1. Pilot Physical Memory Requirements

Working Pilot Function	Set Size	Notes
Communications –XNS		
Idle	7	
Active – first connection	22 to 24	Does not include Idle.
Active – subsequent connections	22 + n*3 to 45 + n*3	Does not include first connection; n = number of streams.
Communications – Base		
Idle	21	
Active	21 to 173	
Communications – Courier		
Idle	5	
Active	18	
Active – bulk data	28	
File		
Create	29	
Delete	28	
SetSize (grow)	31	
SetSize (shrink)	30	
Floppy Channel	10	
Heap		
MakeNode (Heap Impl)	15	
FreeNode (Heap Impl)	15	
MakeNode (SOSP83 Heap Impl)	9	
FreeNode (SOSP83 Heap Impl)	9	
Signals	3	
Space		
Allocate	9	
Deallocate	10	
MapAt	29	
UnmapAt	5	
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.

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

Table A.2 indicates the dynamic cost of virtual memory in terms of disk accesses, CPU time, and real time for a particular disk unit.

Table A.2. Dynamic Cost of Virtual Memory

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

[1] Paging from the local Shugart 4008 disk. Real time per disk access = 100-200 milliseconds.

[2] No 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 maximum time exclusive of such situations.

A.2.2 File management

Table A.3 indicates the typical characteristics of the Pilot file system. In the 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 caches.

Table A.3. Typical Characteristics of Pilot File System

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]	—	$8 * 10^6$

[1] Limited 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	$7.5 * 10^5$ to $2.3 * 10^6$ bits/sec

A.2.4 Processes

Table A.4 provides data about the expected processing time on the Dandelion of each of the process structuring facilities.

Table A.4. Expected Processing Time of Process Structuring Facilities

Facility	Minimum	Typical	Maximum
Monitor entry or exit	3 μsec	< 4 μsec	5 μsec
Process switch time	25 μsec	30 μsec	40 μsec
Fork or Join[2]	0.7 msec	1 msec	1.5 msec
Wait[1,2]	10 μsec	< 60 μsec	100 μsec
Notify[1]	10 μsec	15 μsec	20 μsec

[1] Exclusive of process switching time.

[2] The wide range on this facility reflects a current lack of data about its operating time rather than a dynamic variation in the final product.

B

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 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. 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 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 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 MesaDEFileTypes, of the following form:

```
DIRECTORY
File: USING [Type],
FileTypes: USING [MesaDEFileType];
MesaDEFileTypes: 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 `MesaDEFileTypes.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 MesaDEFileTypes, and if `FileTypes.mesa` was updated after `MesaDEFileTypes.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; for example, 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; that is, 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; for example, 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.

C

Pilot's Interrupt Key Watcher

Appendix C 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, then 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 "panic interrupt." This is done at a priority level that precludes doing any interpret calls from the debugger.

D

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; that is, those on which the boot file is located.

UtilityPilot is commonly used to build special utility systems, such as disk initializers and diagnostics. 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; for example, Ethernet or 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 is provided by **Space.ScratchMap** and the system heaps come from real memory)
- Clients must validate/set local time parameters and the processor clock before calling any Pilot facility that needs them.
- Map logging is disabled.
- Run-time loading is not supported.
- If debugging of UtilityPilot-based clients is desired, then they must be built using a **.bootmesa** file that causes the bcd to be made resident. Without this bootmesa file, UtilityPilot might reclaim the real memory behind the bcd, resulting in an **InvalidLoadState** or bad module messages when the debugger is entered from a UtilityPilot-based client. The command line should be:

  MakeBoot *YourFavoriteUtilityPilotBootFile* [
  parm;**UtilityPilot**, parm:**DebugUtilityPilot**...]

E

Multi-national Considerations

The hardware and software described in this manual support serial communication via the RS232C controller in accordance with EIA standard RS232C. No support is provided for CCITT Recommendations V.24 and V.27, the equivalent prevailing standard in most of Europe.

F

References

F.1 Mandatory references

Study the following documents 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, consult the release documentation accompanying each release of Pilot before writing programs that use Pilot.

F.2 Informational references

The following documents provide additional useful information:

- *The Ethernet, A Local Area Network, Data Link Layer, and Physical Layer Specifications, Version 1.0,* September 30, 1980
- *Xerox Internet Transport Protocols.* XSIS 028112, December 1981.

G

Network Binding Example

```
-- File: NetworkBindingSample.mesa - last edit:
-- AOF          26-Jan-88 16:24:33

-- Copyright (C) 1988 by Xerox Corporation. All rights reserved.

DIRECTORY
    Courier USING [Description, Parameters, VersionRange],
    Heap USING [systemZone],
    Inline USING [LongCOPY],
    NetworkBinding USING [
    BindToAllOnNet, BindToFirstNearby, Conjunct, DeregisterPredicate,
    NoBinding, Predicate, PredicateProcedure, RegisterPredicate, Responses],
    System USING [NetworkAddress, nullNetworkAddress],
    Volume USING [GetAttributes, PageCount, systemID];

NetworkBindingSample: PROGRAM
    IMPORTS Heap, Inline, NetworkBinding, Volume =
    BEGIN

    program: LONG CARDINAL = ...;
    <<
    The program number is a Courier program number and globally administered.
    For details of how to attain a program number, reference the NetworkBinding
    Protocol Specification (in progress).
    >>

    conjunct: NetworkBinding.Conjunct = [1];
    <<
    The conjunct is relative to the program number and locally administered.
    As many conjuncts as desired may be associated with the relevant program number.
    >>

    version: Courier.VersionRange = [1, 1];
    predicate: NetworkBinding.Predicate = [
    program: [programNumber: program, version: version.high], conjunct: conjunct];

    SamplePredicate: TYPE = RECORD[pages: Volume.PageCount];
```

```
SequenceOfLargeFreeSpaces: TYPE = LONG POINTER TO LargeFreeSpaces;
LargeFreeSpaces: TYPE = RECORD[SEQUENCE count: NATURAL OF FreeSpaceTuples];
FreeSpaceTuples: TYPE = RECORD[
    networkAddress: System.NetworkAddress, free: Volume.PageCount];

DescribePages: Courier.Description =
    {notes.noteLongCardinal[notes.noteSize[SIZE[Volume.PageCount]]]};

FindLargeFreeSpace: PUBLIC PROC[pages: Volume.PageCount]
    RETURNS[him: System.NetworkAddress, howmuch: Volume.PageCount] =
    BEGIN

    <<
    This procedure will search all the networks three hops and less from
    this station for a machine that has more than 'pages' of disk space
    free on the system volume. The result from the procedure will be the
    address of the first machine that responds satisfying the criteria. It
    is by definition also the closest. In addition to the address of the
    responding machine, the result will also include the number of pages
    actually available on that machine's system volume.
    If there is no answer to the request, then the procedure will return a
    null System.NetworkAddress and a free count of zero.
    >>

    ENABLE NetworkBinding.NoBinding => 
    {him ← System.nullNetworkAddress; howmuch ← 0; CONTINUE};
    where: LONG POINTER;
    sample: SamplePredicate ← [pages: pages];
    param: Courier.Parameters ← [@sample, DescribePages];
    [him, where] ← NetworkBinding.BindToFirstNearby[
        predicate: [pred: predicate, param: param],
        responseDescription: DescribePages];
    howmuch ← NARROW[where, LONG POINTER TO SamplePredicate].pages;
    Heap.systemZone.FREE[@where]; --need to free the storage for the answer
    END; --FindLargeFreeSpace

FindAllLargeFreeSpaces: PUBLIC PROC[pages: Volume.PageCount]
    RETURNS[list: SequenceOfLargeFreeSpaces] =
    BEGIN

    <<
    This procedure will locate all the machines on the local network that
    have more than 'pages' of disk space available on the system volume.
    The result will be tuples of the network addresses that responded and
    the actual amount of space available.
    If there are no responses, as indicated by a NIL return from the call,
    a NIL will also be returned to the caller of this procedure.
    >>

    sample: SamplePredicate ← [pages: pages];
    param: Courier.Parameters ← [@sample, DescribePages];
    responses: NetworkBinding.Responses ← NetworkBinding.BindToAllOnNet[
```

```
        predicate: [pred: predicate, param: param],
        responseDescription: DescribePages];
    IF responses = NIL THEN RETURN[NIL];
    list ← Heap.systemZone.NEW[LargeFreeSpaces[responses.elementCount]];
    Inline.LongCOPY[ --copy answer to our record construct
        to: list, from: @responses.element[0],
        nwords: responses.elementSize * responses.elementCount];
    Heap.systemZone.FREE[@responses]; --free the original answer
    END; --FindAllLargeFreeSpaces

PredicateProcedure: NetworkBinding.PredicateProcedure =
    --[pred: Predicate, args: LONG POINTER, response: ResponseProc]
    BEGIN

    <<
    This procedure will be called when the NetworkBinding Server has fielded
    a request from the network. The server has already verified that the
    conjunct matches and has converted the predicate of request into a Mesa
    data structure.
    Since the semantics of the request require that in order to answer in the
    affirmative this machine must have more pages free on the system
    volume than specified in the request, a free count of less than that value
    requires that the predicate procedure signal ERROR NoBinding. If it does
    satisfy the criteria, then the actual number of free pages is returned.
    >>

    param: Courier.Parameters ← [@free, DescribePages];
    pages: Volume.PageCount = NARROW[args, LONG POINTER TO
    SamplePredicate].pages;
    free: Volume.PageCount ←
    Volume.GetAttributes[Volume.systemID].freePageCount;
    IF free > pages THEN response[response: param]
    ELSE RETURN WITH ERROR NetworkBinding.NoBinding;
    END; --PredicateProcedure

StartServer: PUBLIC PROC[] =
    BEGIN
    NetworkBinding.RegisterPredicate[
        programNumber: program, versionRange: version,
        conjunct: conjunct, proc: PredicateProcedure,
        predicateDescription: DescribePages];
    END; --StartServer

StopServer: PUBLIC PROC[] =
    BEGIN
    NetworkBinding.DeregisterPredicate[
        programNumber: program, versionRange: version, conjunct: conjunct];
        END; --StopServer

END... --NetworkBindingSample
```

APPENDIX H

TCP/IP Interfaces

APPENDIX H

TCP/IP Interfaces

Appendix H defines and describes the interfaces of TCP/IP communications. The description provides sufficient information to allow the programmer to understand the available facilities and to write procedure calls in the Mesa language to invoke them. For each of the facilities of TCP/IP, the section lists the procedure names, parameters, results, data types of each of the arguments, and possible signals which can be generated.

Relevant RFC references are provided at the beginning of each section.

An RFC can be copied from the <RFC> directory at SRI's machine:

SRI - NIC.ARPA

using FTP with username, ANONYMOUS, and password, GUEST.

1 ArpaConstants

ArpaConstants is the interface that defines the *well-known* ports described in RFC 923.

1.1 Types and constants

```
maxWellknownPort: ArpaRouter.Port = LOOPHOLE[255];

reservedPort: ArpaRouter.Port = LOOPHOLE[0];

rjePort: ArpaRouter.Port = LOOPHOLE[5];
echoPort:ArpaRouter.Port = LOOPHOLE[7];
discardPort: ArpaRouter.Port = LOOPHOLE[9];
activeUsersPort: ArpaRouter.Port = LOOPHOLE[11];
daytimePort: ArpaRouter.Port = LOOPHOLE[13];
netStatPort: ArpaRouter.Port = LOOPHOLE[15];
quotePort: ArpaRouter.Port = LOOPHOLE[17];
charGeneratorPort: ArpaRouter.Port = LOOPHOLE[19];
ftpDataPort: ArpaRouter.Port = LOOPHOLE[20];
ftpControlPort: ArpaRouter.Port = LOOPHOLE[21];
telnetPort: ArpaRouter.Port = LOOPHOLE[23];
smtpPort: ArpaRouter.Port = LOOPHOLE[25];
nwsUserSystemFEPort: ArpaRouter.Port = LOOPHOLE[27];
msgICPPort: ArpaRouter.Port = LOOPHOLE[29];
msgAuthPort: ArpaRouter.Port = LOOPHOLE[31];
anyPrinterServerPort: ArpaRouter.Port = LOOPHOLE[35];
timePort: ArpaRouter.Port = LOOPHOLE[37];
rlpPort: ArpaRouter.Port = LOOPHOLE[39];
graphicsPort: ArpaRouter.Port = LOOPHOLE[41];
nameServerPort: ArpaRouter.Port = LOOPHOLE[42];
nicNamePort: ArpaRouter.Port = LOOPHOLE[43];
mpmFlagsPort: ArpaRouter.Port = LOOPHOLE[44];
mpmReceivePort: ArpaRouter.Port = LOOPHOLE[45];
mpmSendPort: ArpaRouter.Port = LOOPHOLE[46];
niFtpPort: ArpaRouter.Port = LOOPHOLE[47];
loginPort: ArpaRouter.Port = LOOPHOLE[49];
impLogicalAddrMaintPort: ArpaRouter.Port = LOOPHOLE[51];
domainPort: ArpaRouter.Port = LOOPHOLE[53];
isiGlPort: ArpaRouter.Port = LOOPHOLE[55];
privateTermAccessPort: ArpaRouter.Port = LOOPHOLE[57];
privateFileServicePort: ArpaRouter.Port = LOOPHOLE[59];
niMailPort: ArpaRouter.Port = LOOPHOLE[61];
viaFtpPort: ArpaRouter.Port = LOOPHOLE[63];
tftpPort: ArpaRouter.Port = LOOPHOLE[69];
remoteJobService1Port: ArpaRouter.Port = LOOPHOLE[71];
remoteJobService2Port: ArpaRouter.Port = LOOPHOLE[72];
remoteJobService3Port: ArpaRouter.Port = LOOPHOLE[73];
remoteJobService4Port: ArpaRouter.Port = LOOPHOLE[74];
privateDialoutPort: ArpaRouter.Port = LOOPHOLE[75];
privateRJEPort: ArpaRouter.Port = LOOPHOLE[77];
fingerPort: ArpaRouter.Port = LOOPHOLE[79];
```

```
hostS2NameServerPort: ArpaRouter.Port = LOOPHOLE[81];
mitMlDevice1Port: ArpaRouter.Port = LOOPHOLE[83];
mitMlDevice2Port: ArpaRouter.Port = LOOPHOLE[85];
mitMlDevice3Port: ArpaRouter.Port = LOOPHOLE[87];
suMitTelnetGatewayPort: ArpaRouter.Port = LOOPHOLE[89];
mitDoverSpoolerPort: ArpaRouter.Port = LOOPHOLE[91];
dcpPort: ArpaRouter.Port = LOOPHOLE[93];
supdupPort: ArpaRouter.Port = LOOPHOLE[95];
swiftRvfPort: ArpaRouter.Port = LOOPHOLE[97];
metagramPort: ArpaRouter.Port = LOOPHOLE[99];
hostNamePort: ArpaRouter.Port = LOOPHOLE[103];
csnetNsPort: ArpaRouter.Port = LOOPHOLE[105];
remoteTelnetPort: ArpaRouter.Port = LOOPHOLE[107];
postOfficeProtocolPort: ArpaRouter.Port = LOOPHOLE[109];
sunRpcPort: ArpaRouter.Port = LOOPHOLE[111];
authPort: ArpaRouter.Port = LOOPHOLE[113];
sftpPort: ArpaRouter.Port = LOOPHOLE[115];
uucpPathPort: ArpaRouter.Port = LOOPHOLE[117];
surveyMeasurePort: ArpaRouter.Port = LOOPHOLE[243];
linkPort: ArpaRouter.Port = LOOPHOLE[245];
```

2 ArpaRouter

ArpaRouter is the interface for common, public types and procedures of the lower level ARPA Internet transport and internet facilities.

References: *RFC768 User Datagram Protocol, Postel, August, 1980.*

RFC791 Internet Protocol, Postel, September, 1981.

RFC793 Transmission Control Protocol, Postel, September, 1981.

2.1 Types and constants

ArpaRouter.Port: Type[1];

Port is a TCP or UDP port as defined in the TCP and UDP protocol specifications, RFC 793 and RFC 768. It is used for intra-machine multiplexing and is a parameter in many of the procedures in the interfaces to the Arpa protocol implementations. A **Port** can either have a well known value or a unique value. A well known value is one that is defined in **ArpaConstants**; a unique value is one that is known to both sides of a connection only for the duration of that connection.

ArpaRouter.InternetAddress: TYPE[2];

InternetAddress is an ARPA Internet address of any address class as defined in the IP protocol specification, RFC 791. **InternetAddress** is used by the low level communications as source and destination addresses for other hosts in the Internet. This type is a parameter in many of the procedures in the interfaces to the ARPA protocol implementations.

ArpaRouter.unknownInternetAddress: READONLY ArpaRouter.InternetAddress;

unknownInternetAddress can be used for initializing an **InternetAddress** and indicating an address that has not been set to a valid address. It is **not** a null address indicating the local machine or network. Such an address may not have 0 bits in the address class fields and should be obtained by other means.

2.2 Procedures

ArpaRouter.ArpaPackageMake: PROCEDURE ;

ArpaPackageMake starts and initializes the TCP/IP protocol family. This procedure must be called at least once during the loading of the protocol suite. Once started, subsequent calls cause a reference count to be incremented but take no other action.

ArpaRouter.ArpaPackageDestroy: PROCEDURE;

ArpaPackageDestroy stops and unregisters the TCP/IP protocol family. Operationally the procedure first decrements the reference count. If and only if the result is zero does it actually stop the protocol and delete the resources required to support it.

```
ArpaRouter.GetAddress: PROCEDURE RETURNS [ArpaRouter.InternetAddress];
```

GetAddress returns the **InternetAddress** of the local machine. If the internet address is not known, then **unknownInternetAddress** is returned.

3 ArpaRouterOps

ArpaRouterOps is the interface for procedures that are of use to many clients of the ARPA networking facility. This interface should be viewed as an extension to **ArpaRouter.** The facilities defined here are done so that they may evolve more freely.

Some of the procedures defined in **ArpaRouterOps** use a modified semantic definition of an internet address (**ArpaRouter.InternetAddress**). Internet addresses nominally define a unique machine address within an internet. The record includes fields that represent the network number, in cases where subnetting is used, a subnetwork number, and a network or subnetwork relative host number. In cases where the type **ArpaRouter.InternernetAddress** is used to identify a network or subnetwork, the bits representing the host have been stripped.

It is assumed, and this implementation relies on the assumption, that all subnets on a particular network will be using the same subnetting strategy. Consequently, in order to strip the host number field from an internet address record, the following rules are applied.

1) The network number is extracted from the internet address. This can be done mechanically by looking at the addressing class bits in the record.

2) If the network number extracted in 1) is the same as the network number of the directly attached network, the local subnet mask is applicable. By AND'ing that mask with the original record, the host field is cleared and the resultant value that includes both the network number and subnet number fields and becomes the value referred to as either the network number or the subnet number.

3) If the network number extracted in 1) is not equal to the local network, then nothing can be assumed about the subnetting strategey being used on the remote network. Therefore the result of 1) becomes the network number for comparison purposes. It is also assumed that there is some external gateway between the remote and the local subnet that knows about the remote's subnetting scheme if one should exist.

References: *RFC1058 Routing Information Protocol, Hendrick, June, 1988.*

3.1 Types

```
ArpaRouterOps.IsSame: TYPE = {yes, no, cantTell};
```

IsSame defines the type of returns possible by the routine **ArpaRouterOps.SameSubnet**. In addition to an absolute answer, it defines the possibility that there is not enough context locally to make the determination required.

3.2 Exported variables

```
ArpaRouterOps.startEnumeration: READONLY ArpaRouter.InternetAddress;
ArpaRouterOps.stopEnumeration: READONLY ArpaRouter.InternetAddress;
```

The two variables listed are used in conjunction with the stateless enumeration procedure, **ArpaRouterOps.Enumerate**. They have no valid application except within the context of the enumerate procedure.

```
ArpaRouterOps.allNetworks: READONLY ArpaRouter.InternetAddress;
```

ArpaRouterOps.allNetworks defines the value that may be used in conjunction with **ArpaRouterOps.Flush** to indicate that the client wishes to flush all entries that are currently in the routing table. The variable has no valid application except within the context of the flush procedure.

3.3 Procedures

```
ArpaRouterOps.Enumerate: PROCEDURE[
    last: ArpaRouter.InternetAddress, delay: CARDINAL]
    RETURNS[next,gateway: ArpaRouter.InternetAddress];
```

ArpaRouterOps.Enumerate is a stateless enumerator used to list the contents of the local IP routing table. Each call to **Enumerate** will return a network and the address of the immediate gateway used to reach the network. For each call, a unique network that has a delay characteristic specified (currently delay is measured in hops, roughly equivalent to the number of gateways that a packet must pass through to reach the remote network). To perform a complete enumeration at a specific delay, the first value passed as the argument **last** should be **ArpaRouterOps.startEnumeration**. If the value returned in **next** is not equal to **ArpaRouterOps.endEnumeration** it represents another remote net of the specified **delay**, and the second returned value, **gateway**, is the address of the immediate machine to which the packet will be sent in order to reach any destination on the netwok **next**. The value of **next** should then be used as the argument **last** in the next call to **Enumerate** and the process should continue until next is returned with a value equal to **ArpaRouterOps.endEnumeration**.

The following Mesa example shows a correct use of the enumerate procedure. It enumerates all networks that are currently in the cache with a delay between zero and 255. A delay of zero indicates that the network is directly attached. Enumerations should always indicate at least one network, the local network.

```
FOR delay: NATURAL IN[0..256) DO
    network: ArpaRouter.InternetAddress ← ArpaRouterOps.startEnumeration;
    DO
        gateway: ArpaRouter.InternetAddress;
        [network, gateway] ← ArpaRouterOps.Enumerate[network, delay];
        IF network = ArpaRouterOps.endEnumeration THEN EXIT;
        <<do something with network and gateway>>
        ENDLOOP;
    ENDLOOP;
```

```
ArpaRouterOps.Flush: PROCEDURE[
    network: ArpaRouter.InternetAddress ← ArpaRouterOps.allNetworks];
```

ArpaRouterOps.Flush may be used to flush a currently cached route to a remote network or all cached entries, other than the local network. The value specified for the argument **network** may be any valid internet address on the target network. The host number portion of the address will be stripped as discussed earlier in order to determine the correct entry to flush.

The default value of **ArpaRouterOps.allNetworks** indicates that the entire table is to be flushed.

Caution: The cache or routes to remote networks is global to the entire system. Clients should use this procedure with caution since they may have adverse affects on the performance of clients other than themselves.

Note: Under no circumstances is it possible to flush the entry representing directly attached networks. Attempts to do so are treated as no-ops.

```
ArpaRouterOps.GetDelay: PROCEDURE[network: ArpaRouter.InternetAddress]
    RETURNS[hops: CARDINAL, gateway: ArpaRouter.InternetAddress];
```

ArpaRouterOps.GetDelay may be used to retrieve a hint about the distance to a remote partner. This procedure is most meaningful if the network is using the Routing Information Protocol [RIP]. If RIP is being used, then the value of **hops** returned by the procedure call is roughly equivalent to the number of gateways a packet must pass through to get to the remote netork and will have values ranging from zero (directly attached) to 16 (not reachable).

If RIP is not being used, **hops** will have one of four values. A value of zero indicates that the remote network is one directly attached to host machine via a broadcast medium (e.g., ethernet). If the remote network is on the same network but a different subnet, the procedure will return a value of one. If the network does not have the same network number but still believed to be reachable, the value return in **hops** will be 15. If the network is unreachable, then the value returned will be identical to that returned if RIP is enabled, 16.

The value returned for **gateway** will be consistent whether RIP is being used or not. It is the internet address of the gateway to which outbound packets will be transmitted in order to deliver them to the remote network. If the remote network is directly attached (i.e., **hops** is zero) the value of **gateway** will be **Router.unknownInternetAddress**.

```
ArpaRouterOps.GetUseCount: PROCEDURE[] RETURNS[users: CARDINAL];
```

As noted in the specification of those procedures, **ArpaRouter.ArpaPackageMake** will simply increment a reference count if the protocol has already been started. And **ArpaRouter.ArpaPackageDestroy** will not stop the protocols if decrementing the reference count does not result in a zero value. **ArpaRouterOps.GetUseCount** may be used to interogate the number of times **ArpaRouter.ArpaPackageMake** and **ArpaRouter.ArpaPackageDestroy** have been called by returning the current reference count, or the current number of *users* of the protocol package, to the caller.

Caution: Unloading the configuration that includes the ARPA protocol engines (any configuration including ArpaSubConfg.bcd) with a reference count that is not zero will result in non-deterministic behavior.

```
ArpaRouterOps.LocalNetwork, LocalSubnet: PROCEDURE[
    network: ArpaRouter.InternetAddress] RETURNS[isLocal: BOOLEAN];
```

The procedure **ArpaRouterOps.LocalNetwork** may be used to determine if the address specified is included in the same network as the host machine. That is to say, the network number is the same. It does not guarantee that the remote address is directly accessible.

ArpaRouterOps.LocalSubnet may be used to determine if the specified address is on the same subnet as the host machine. If the result is true, this guarantees that the remote is on the same network (i.e., **LocalNetwork** would also return true with the same argument) and it also asserts that the remote address is directly accessible.

```
ArpaRouterOps.SameNetwork: PROCEDURE[netA, netB: ArpaRouter.InternetAddress]
    RETURNS[isSame: BOOLEAN];

ArpaRouterOps.SameSubnet: PROCEDURE[netA, netB: ArpaRouter.InternetAddress]
    RETURNS[isSame: IsSame];
```

The two procedures **ArpaRouterOps.SameNetwork** and **ArpaRouterOps.SameSubnet** are extensions of **ArpaRouterOps.LocalNetwork** and **ArpaRouterOps.LocalSubnet**. In the latter case the argument passed is compared to the values of local parameters. Thus **ArpaRuterOps.SameNetwork[someInternetAddress, ArpaRouter.GetAddress[]]** is identical to **ArpaRouterOps.LocalNetwork[someInternetAddress]**.

In the case of **ArpaRouterOps.SameSubnet**, it is possible that the two addresses specified are not on the network. Under those conditions, it is not possible to ascertain the subnet masks of one or both of the networks specified. That results in a returned value of **cantTell**.

4 ArpaSysParameters

ArpaSysParameters defines some of the options that may be included in IP and TCP headers, including *type of service* and *security* abstractions. The current package provides the following guarantees with respect to type of service and security options.

A client may globally set any of the type of service parameters. Those parameters will be transmitted in all outbound IP packets. The protocol package defined here is used in an end station environment only. Consequently, other than transmitting the information, there is no processing involved in supporting any of the type of service options.

The implementation described in this document is intended to be used in an *unclassified* environment only. Client setting of security levels other than unclassified (the default) is unsupported. As long as the global security setting remains unclassified, the IP options security options field will not be transmitted. Inbound packets may include security options. If those options indicate a security classification higher than unclassified, there is no guarantee that those packets will be discarded.

4.1 Types and constants

Many of the following types are used simply in the development of the top level types, **TypeOfService** and **SecurityObject**. Consequently the intermediate types will not be defined.

Fine point: However, it is left as an exercise to the reader to prove that they are correct.

```
ArpaSysParameters.AuthorityFlag: TYPE = MACHINE DEPENDENT RECORD[
    source(0:0..6): WORD[0..128), more(0:7..7): BOOLEAN];

ArpaSysParameters.nullAuthority: AuthorityFlag = [0, FALSE];

ArpaSysParameters.IpOption: TYPE = MACHINE DEPENDENT RECORD[
    option(0:0..7): OptionNumber,
    length(0:8..15): NATURAL[0..256)];

ArpaSysParameters.Precedence: TYPE = MACHINE DEPENDENT{
    routine(0), priority, immediate, flash, flashOverride, criticEcp,
    internetControl, networkControl(7)};

ArpaSysParameters.OptionClass: TYPE = MACHINE DEPENDENT{
    control(0), rsvd1(1), debugging(2), rsvd2(3)};

ArpaSysParameters.OptionNumber: TYPE = MACHINE DEPENDENT RECORD[
    cf(0:0..0): BOOLEAN,
    class(0:1..2): OptionClass,
    number(0:3..7): NATURAL[0..40B)];

ArpaSysParameters.endNumber: OptionNumber = [FALSE, control, 0];

ArpaSysParameters.noopNumber: OptionNumber = [FALSE, control, 1];
```

```
ArpaSysParameters.basicSecurityNumber: OptionNumber = [FALSE, control, 2];

ArpaSysParameters.looseSourceRoutingNumber: OptionNumber = [FALSE, control, 3];

ArpaSysParameters.extendedSecurityNumber: OptionNumber = [FALSE, control, 5];

ArpaSysParameters.structSourceRoutingNumber: OptionNumber = [FALSE, control, 9];

ArpaSysParameters.recordRouteNumber: OptionNumber = [FALSE, control, 7];

ArpaSysParameters.streamIDNumber: OptionNumber = [FALSE, control, 8];

ArpaSysParameters.timestampNumber: OptionNumber = [FALSE, debugging, 4];

ArpaSysParameters.basicSecurityLength: NATURAL[0..256) = 7;

ArpaSysParameters.SecurityType: TYPE = MACHINE DEPENDENT{type(130)};

ArpaSysParameters.SecurityLength: TYPE = MACHINE DEPENDENT{minimumLength(4)};

ArpaSysParameters.Security: TYPE = LONG POINTER TO READONLY SecurityOption;

ArpaSysParameters.SecurityObject: TYPE = MACHINE DEPENDENT RECORD[
    type(0:0..7): SecurityType ← type,
    length(0:8..15): SecurityLength ← minimumLength,
    level(1:0..7): SecurityLevel ← unclassified,
    protection(1:8..15): AuthorityFlag ← nullAuthority];

ArpaSysParameters.SecurityLevel: TYPE = MACHINE DEPENDENT{
    unclassified(85), confidential(122), secret(173), topSecret(222)};

ArpaSysParameters.SecurityOption: TYPE = MACHINE DEPENDENT RECORD[
    option(0:0..15): IpOption ← [basicSecurityNumber, basicSecurityLength],
    security(1:0..31): SecurityObject ← [],
    noop(3:0..7): OptionNumber ← noopNumber,
    end(3:8..15): OptionNumber ← endNumber];
```

The **SecurityOption** construct defines the *default* security implemented. Since this is a multiword construct, procedures requiring a security argument pass a reference to a **SecurityOptions** record. On those procedures that accept a security argument, a NIL is equivalent and preferrable to the default security options. Either setting will result in packets being transmitted with no security options included.

```
ArpaSysParameters.TypeOfService: TYPE = MACHINE DEPENDENT RECORD[
    precedence(0: 0..2): Precedence ← routine,
    delay(0: 3..3): BOOLEAN ← FALSE,
    throughput(0: 4..4): BOOLEAN ← FALSE,
    reliability(0: 5..5): BOOLEAN ← FALSE,
    reserved(0: 6..7): WORD[0..4) ← 0];
```

The **TypeOfService** construct defines the *default* type of service implemented and is the value used as a default in procedures that require a type of service argument.

```
ArpaSysParameters.TcpOptionType: TYPE = MACHINE DEPENDENT{
    eol(0), nop, maxSegment, (255)};
```

4.2 Procedures

The following procedures set the relevant parameters that are used as system wide defaults.

```
ArpaSysParameters.GetSecurity: PROC[] RETURNS[security: Security];
```

A client may query the current security options at any time by calling **GetSecurity**. The reference returned is *readonly* and may not be used to modify the value of the system's security object directly. In order to maintain a copy of the current settings the referent must be copied into a variable allocated by the client.

```
ArpaSysParameters.SetSecurity: PROC[security: Security];
```

A client may call **SetSecurity** and modify the system's current value of the security options. The referent passed as the **security** argument will be copied into the system's global data and applied to all transmitted and received packets from that point on.

Note: Setting any value of the security options other than the default is not supported.

```
GetTypeOfService: PROC[] RETURNS[tos: TypeOfService];
```

A client may query the current type of service options at any time by calling **GetTypeOfService**.

```
SetTypeOfService: PROC[tos: TypeOfService];
```

In order to modify the current assignments for type of service, a client may call **SetTypeOfService**. The value specified as **tos** will become the system's default value.

5 ArpaUtility

ArpaUtility is a collection of procedures through which higher level clients may specify configuration details that may be environment and/or installation specific.

5.1 Types

ArpaUtility.Option: TYPE = {
securityDiscard, tcpKeepAlive, useRip, spare2, spare3, spare4, spare5};

The current implementation permits the setting of several soft switches defined by **ArpaUtility.Option**. Each option has a specific function, a default value and global impact.

securityDiscard
This switch is used to control the disposition of Internet Protocol (IP) packets that arrive at the local machine with a security classification higher than is supported. If this switch is set to TRUE such packets will be discarded and the clients will not be notified. If it is FALSE, the packets will be delivered normally. The default value of the switch is TRUE, i.e., the packets will be discarded.

tcpKeepAlive
If the tcpKeepAlive switch is TRUE the system will transmit *keepalive packets* on all idle TCP connections at predetermined intervals.

useRip
Routing Information Protocol (RIP) is a Internal Gatway Protocol (IGP) that has gained some poularity in the ARPA community. It provides end stations with hints about routing topologies and delays to remote networks. As such it can be used by higher level clients as input into timeout algorithms and the like. If this option is set to TRUE the system will listen for RIP response packets that are gratuitously broadcasted by gateways supporting RIP to refresh the local machine's internal routing table. Having this switch set to TRUE will also cause the station to solicit information about routes to remote networks when the entries are first entered into the cache. The default value for this switch is FALSE. In that setting the system relies on the static information (see **ArpaUtility.SetGateway**) in conjunction with ICMP redirects to maintain the local cache.

spare2, spare3, spare4, spare5
These switches are reserved and currently unimplemented.

ArpaUtility.Options: TYPE = PACKED ARRAY Option OF BOOLEAN;

ArpaUtility.Options defines the type of the exported variable.

5.2 Exported variables

ArpaUtility.options: READONLY Options;

ArpaUtility.option exports the current setting of each of the switches. These switches are READONLY and may only be set during startup.

ArpaUtility.zone: <<READONLY>> UNCOUNTED ZONE;

ArpaUtility.zone exports a heap that is used by all internal state machines for incidental storage. This heap must be created before starting any of the protocol engines

(ArpaRouter.ArpaPackageMake) by calling ArpaUtility.CreateZone. The zone must not be deleted until after those protocol engines have been stopped (by calling ArpaRouter.ArpaPackageDestroy).

5.3 Procedures

```
ArpaUtility.CreateZone: PROC[pages, increment: CARDINAL ← 10];
```

ArpaUtility.CreateZone must be the first procedure called in the initialization process. This procedure creates the heap used by all the low level protocol machines for incidental storage. The client may tune the heap to expected requirements by adjusting the **pages** and **increment** arguments appropriately. Those arguments will be applied directly to a Heap.Create function. If the heap has already been created, this operation is a no-op.

```
ArpaUtility.DeleteZone: PROC[checkEmpty: BOOLEAN];
```

This procedure must not be called until after the client has first called ArpaRouter.ArpaPackageDestroy. It should then be called to delete the storage occupied by the heap during operation. If the package is to be unloaded, to not call this procedure at the proper time would cause a storage leak of the entire heap even if the heap is empty. When called, the client may test to see if the heap was indeed empty by setting the **checkEmpty** argument to TRUE. Calls the ArpaUtility.DeleteZone where the heap has already been deleted is treated as a no-op.

```
ArpaUtility.GetHost: PROC[] RETURNS[ArpaRouter.InternetAddress];
```

```
ArpaUtility.GetGateway: PROC[] RETURNS[ArpaRouter.InternetAddress];
```

```
ArpaUtility.GetSubnetMask: PROC[] RETURNS[ArpaRouter.InternetAddress];
```

```
ArpaUtility.GetNameServer: PROC[] RETURNS[ArpaRouter.InternetAddress];
```

```
ArpaUtility.GetDefaultDomain: PROC[LONG STRING];
```

Each of the above procedures returns, in internal format, the value of the relevant global variable that was set during initialization.

Caution: Due to a startup problem, it is not possible for the implementation to detect if the global variables have actually been set. It is the client's responsibility to insure that the proper *set* procedure has been called before any calls to any of the *get* procedures.

```
ArpaUtility.SetHost: PROC[LONG STRING];
```

ArpaUtility.SetHost defines the *home* internet address of the machine. That address is usually associated with the primary networking device, which is most likely to be the first ethernet controller. Configurations not having an ethernet as the primary networking controller are not supported. This caution is also applicable to ArpaUtility.SetGateway, ArpaUtility.SetNameServer and ArpaUtility.SetDefaultDomain if they are specified.

Caution: The LONG STRING passed in as the argument must not require any network access. It should be of the form *DD.DD.DD.DD* where the *'D* is a decimal digit. The numbers may be in octal or hexidecimal base providing the fields are suffixed with the proper character, 'B or 'H respectively, but this is not recommended.

```
ArpaUtility.SetGateway: PROC[LONG STRING];
```

If the system is being used in an internet or subnet environment, the IP address of the primary gateway must be specified. This need not be the only gateway available on the

local subnet. If others are available, they will be discovered either by ICMP redirect packets or by use of the RIP protocol.

ArpaUtility.SetSubnetMask: PROC[LONG STRING];

If subnetting (see RFC 950) is to be used on the local network, the subnet mask must be specified. The subnet mask is required by this implementation to be consistant for all subnets on the same network.

ArpaUtility.SetNameServer: PROC[LONG STRING];

ArpaUtility.SetNameServer provides the address of the default Domain server. It need not be the only domain server available on the local subnet or network or internet. Others may be discovered by the *resolver* by using the Domain protocol.

ArpaUtility.SetDefaultDomain: PROC[LONG STRING];

The LONG STRING specified in **ArpaUtility.SetDefaultDomain** will be appended to lookups that use the *resolver*.

ArpaUtility.SetSystemOptions: PROC[option: Option, setting: BOOLEAN];

This procedure must be called before **ArpaRouter.ArpaPackageMake** if it is to make changes to the default settings. It may be called as many times as there are options to set, once for each option. The value of setting may be interpreted as a pure **BOOLEAN** (i.e., **TRUE** or **FALSE**) or as a switch using **TRUE** as equivalent to *on* and **FALSE** as equivalent to *off*.

6 Resolve

The Resolve interface translates between human-readable strings and the internal representation of Internet addresses.

References: *RFC952. DoD Internet Host Table Specification. Harrenstien, October, 1985.*
RFC1034. Domain Names - Concepts and Facilities. Mockapetris, November, 1987.
RFC1035. Domain Names - Implementation and Specification. Mockapetris, November, 1987.

6.1 Signals and Errors

```
Resolve.CantAcquireHostsDotTxt: SIGNAL;
```

This signal is obsolete and will never be raised again.

```
Resolve.Error: ERROR [which: WhichError];

Resolve.WhichError: TYPE = {
    badQuery, serverFailure, nameError, notImplemented, refused, otherRCode,
    serviceUnavailable, serverUnreachable, networkProblem, cantFind, noAnswer,
    badReply, badSyntax, other};
```

Resolve.Error may be raised by several of the procedures defined below. The table below defines the reasons the error was raised and possible corrective actions when the error is raised.

badQuery	A name server answered that didn't recognize the question (parameters of the query were not understood). This is an internal error in either the local implementation or that of the server that responded. This operation should not be retried and the local system administrators should be notified.
serverFailure	A server failure notification was returned by a name server. The indication is that there is some problem with the particular server. Local system administration should be notified.
nameError	The server that responded claims that there is no such name. That server is supposed to be authoritive. If the problem persists after verifying the arguments of the procedure call, notify local system administration.
notImplemented	The feature requested is not provided by the implementation of a name server that responded. Verify that the operation is valid before notifying administrative personnel.
refused	A name server refused to answer. It may due to a security issue. Do not retry the query.
otherRCode	This error code will be returned if the server responded with an extended return code. Call a developer.

serviceUnavailable	The domain query resulted in an ICMP packet indicating that there was no listener at the well-known port or the wrong protocol was being used. Perhaps the service is temporarily unavailable. Try again in a few minutes and if the situation persists, notify administrative personnel.
serverUnreachable	There is no domain server currently reachable from the host machine, perhaps due to a temporary network failure. Try again in a few minutes and if the situation persists, notify administrative personnel.
networkProblem	Some (hopefully) temporary unexpected network failure has caused the operation to fail. Try again in a few minutes.
cantFind	No server could be located that knows about the existance or non-existance of the name specified.
noAnswer	No server responded in a timely fashion. Try again later.
badReply	The server responded with a reply that could not be processed locally. Either the server or the local implementation is in error. Notify the local system administrators.
badSyntax	The argument to the procedure call had a bad syntax. This apparently is a client problem and should not be retried until the data is verified.
other	The name exists but there is no data of the specified type or the truncation bit is set in the reply.

6.2 Procedures

```
Resolve.LocalStringToAddr: PROC [s: LONG STRING]
    RETURNS [addr: ArpaRouter.InternetAddress];
```

Resolve.LocalStringToAddr parses its argument string and returns the corresponding internet address. The argument must be in standard host number representation: four numeric literals separated by dots, representing the four bytes of the internet address in order from most significant to least significant. This procedure can raise **Resolve.Error[badSyntax]**.

```
Resolve.StringToAddr: PROC [s: LONG STRING]
    RETURNS [addr: ArpaRouter.InternetAddress];
```

Resolve.StringToAddr parses its argument string and returns the corresponding internet address. If the argument is in standard host number representation , then it is converted directly to an internet address as in **Resolve.LocalStringToAddr**, above. Otherwise, the argument is looked up first in the host table HOSTS.TXT (if available), then in the Domain Name Service if not found, and the resulting address is returned. This procedure can raise **Resolve.Error**.

```
Resolve.AppendNameFromAddr: PROC [
    to: LONG STRING, addr: ArpaRouter.InternetAddress, radix: CARDINAL ← 10];

Resolve.LocalAppendAddr: PROC [
    to: LONG STRING, addr: ArpaRouter.InternetAddress, radix: CARDINAL ← 10];
```

Resolve.LocalAppendAddr converts an **ArpaRouter.InternetAddress** to standard host number representation: four numeric literals separated by dots, representing the four bytes of the

internet address in order from most significant to least significant. The result is appended to the first argument, **to**. **String.StringBoundsFault** will be raised if the string described in **to** is too short. The format of the numeric literals is determined by **radix** as in **String.AppendNumber**.

Resolve.AppendNameFromAddr converts an **ArpaRouter.InternetAddress** to a human-readable name by looking up the address first in the host table HOSTS.TXT (if available), then in the Domain Name Service if not found. If the lookup succeeds, the result is appended to the first argument, **to**. **String.StringBoundsFault** will be raised if the string described in **to** is too short. If the lookup fails, **Resolve.Error** will be raised.

7 TcpStream

TcpStream is the client interface to the implementation of TCP in the ARPA family of protocols. TCP provides a sequenced, error-free stream across interconnected communication networks with duplicate suppression and flow control. It is assumed that the client is familiar with the TCP protocol specification, RFC 793.

References: *RFC793 Transmission Control Protocol, Postel, September, 1981.*

7.1 Types and constants

```
TcpStream.CompletionCode: TYPE = {normal, timeout, pushed, closing, endUrgent};
```

Returned from the **get** procedure, **CompletionCode** indicates the status of that **get**. Any status besides **normal** may have returned less data than requested by the client. The client should look at the **byteCount** return code from the **get** in these cases to determine the amount of data actually returned.

normal indicates that the **get** completed normally, returning to the client after retrieving the amount of data requested. **timeout** indicates that the TCP waited the amount of time specified in the **Make** or subsequent **setWaitTime**, and the requested amount of data did not arrive. **pushed** indicates that the remote end pushed the data, causing it to be transmitted immediately from the remote, and causing the local **get** to return at the point in the stream when it noticed the pushed data. **closing** implies a push, and indicates the remote end has no more data to send and issued a **close**, causing the local **get** to return upon receipt of the close. **endUrgent** indicates the client (who was previously notified of an urgent via **waitForUrgent**) has reached the last byte of the data marked urgent.

```
TcpStream.defaultWaitTime: TcpStream.WaitTime = 60000;
```

The default wait time of 60 seconds is taken from the maximum TCP packet lifetime.

```
TcpStream.WaitTime: Type = LONG CARDINAL;
```

```
TcpStream.Failed: SIGNAL [why: TcpStream.FailureReason];
```

```
TcpStream.FailureReason: TYPE = {
    timeout,
    noRouteToDestination,
    noServiceAtDestination,
    remoteReject,
    precedenceMismatch,
    securityMismatch,
    optionMismatch,
    noAnswerOrBusy,
    noTranslationForDestination,
    circuitInUse,
    circuitNotReady,
    noDialingHardware,
    dialerHardwareProblem};
```

Failed is the error that is raised when a connection could not be established. Since the connection was never established, the client should not attempt to **Close** or **Destroy** it after this error.

Failure reasons are described below.

timeout
: The connection could not be established within the amount of time the client specified in the timeout parameter to the **Make** procedure.

noRouteToDestination
: No route exists from the local socket to the target socket.

remoteReject
: The connection had to be reset for one of two reasons: either a TCP reset packet was received from the remote or a syn packet was received that acknowledged data was never sent. Though such a packet is probably a "stray," the integrity of the connection was jeopardized, and it was reset. The client may want to try the operation again, in case the condition was transient.

precedenceMismatch
: A connection attempt was made to the local machine with a precedence lower than that specified in the **Listen** or **Make** procedure.

securityMismatch
: A connection attempt was made to the local machine with a security level lower than that specified in the **Listen** or **Make** procedure.

```
TcpStream.Handle: TYPE = LONG POINTER TO TcpStream.Object;
TcpStream.Object: TYPE = RECORD [
    destroy: PROC[tsH: TcpStream.Handle],
    put: PROC [block: Environment.Block, push, urgent: BOOLEAN],
    get: PROC [block: Environment.Block]
        RETURNS [byteCount: CARDINAL, completionCode: TcpStream.CompletionCode],
    waitForUrgent: PROC [block: Environment.Block],
    close: PROC,
    setWaitTime: PROC [TcpStream.WaitTime],
    findAddresses: PROC RETURNS [
        localAddr, remoteAddr: ArpaRouter.InternetAddress,
        localPort, remotePort: ArpaRouter.Port]];
```

A **Handle** uniquely identifies a connection. Operations on this connection are executed by calls to the procedure fields in the **Object**.

destroy deletes the TCP stream. Except under error conditions, clients should call the **close** proc before deleting the stream in order to close gracefully and ensure proper delivery/reception of the last piece of data. A call to **destroy** on a non-suspended stream without executing the closing protocol causes a TCP reset message to be sent to the remote end. **destroy** flushes all input and output queues and destroys all stream state information. After calling **destroy**, the stream handle is invalid and cannot be used again.

put queues the block of client output data specified by **block** for transmission. If the client wishes to specify that the data be flushed out to the network instead of being buffered in the local TCP, then **push** should be set to **TRUE**. Indicating **push** not only prevents buffering at the local end, but also causes outstanding **gets** at the remote end to return immediately upon receiving the pushed data. Use **push** only when really needed, as it impacts the

efficiency of the connection. Setting **urgent** marks the last byte of the block as the end of urgent data.

get retrieves the specified amount of data from the stream and puts it in **block**. If the data is not pending, then **get** waits the amount of time specified in the **Make** call (or subsequent **setWaitTimes**) for the data to arrive from the remote. **byteCount** is the actual number of bytes transferred, as the client may not get the amount of data requested if the wait time is exceeded, if the data was pushed by the remote, or if the remote closed. The **completionCode** indicates the status of the completed **get** call.

waitForUrgent watches for a packet to arrive with the urgent bit set. This procedure returns as soon as TCP receives a packet with the urgent bit set. It is then the client's responsibility to issue **gets** to flush the stream to the end of the urgent data. Typically, a client has a separate process that is waiting in **waitForUrgent** and it notifies the data receiver when it receives notification of pending urgent data. As with all operations that block, **waitForUrgent** can be canceled.

close is the operation used to start gracefully closing down the stream when the client has no more data to send. Outstanding **puts** are transmitted until complete, as flow control permits. After calling this procedure, the client should issue **gets** to receive outstanding data until a **get** returns with the completion code of closed, indicating the remote end has also issued a **close**. It is the client's responsibility to continue the graceful close handshake by retrieving the **close** outcome from the remote with **get**. If the client destroys the stream immediately after issuing a **close** without waiting for the **close** from the other end, then the data will not be reliably transferred.

setWaitTime sets the current timeout value for the stream. The **timeout** is the amount of time in milliseconds that a **get** waits for the requested data before returning to the client.

findAddresses returns the sockets that identify the connection.

```
TcpStream.NotifyListenStartedProc: TYPE = PROCEDURE;
```

NotifyListenStartedProc is called by **Listen** or **Make** during a passive connection establishment. It is called when all resources have been allocated and started for the connection and the port is ready to accept a connection attempt.

```
TcpStream.SuspendReason: TYPE = {
    notSuspended, transmissionTimeout, noRouteToDestination,
    remoteServiceDisappeared, reset, securityMismatch, precedenceMismatch};
```

Suspended is raised if an already established connection is suspended for any reason. The only operation a client can (and must) do after receiving this error is **Delete**. It is also the client's responsibility to cause an **UNWIND** so that the TCP state can be properly cleaned up before calling **Delete**.

Suspend reasons are described below.

notSuspended
: Used for internal processing and should never be seen by the client.

transmissionTimeout
: The remote end has not acknowledged data sent to it in a long time. The local end concluded that the remote has disappeared.

noRouteToDestination
The route from the local socket to the remote socket has disappeared and another could not be found to use.

remoteServiceDisappeared – unused.

reset A TCP reset message was received from the other end.

securityMismatch
A connection attempt was made to the local machine with a security level lower than that specified in the **Listen** or **Make** procedure.

precedenceMismatch
A connection attempt was made to the local machine with a precedence lower than that specified in the **Listen** or **Make** procedure.

noAnswerOrBusy, noTranslationForDestination, circuitInUser, circuitNotReady, noDialingHardware, dialerHardwareProblem
These failures refer to problems on circuit-oriented networks and are not implemented.

WaitTime is used for establishing intervals for timeouts. It is always in milliseconds.

7.2 Exported variables

```
TcpStream.infiniteWaitTime: READONLY TcpStream.WaitTime;
```

infiniteWaitTime is used either to keep an operation from ever timing out or to declare no interest in processing timeouts. A client using **infiniteWaitTime** should be prepared to cancel the affected process at some time.

```
TcpStream.uniquePort: READONLY ArpaRouter.Port;
```

uniquePort is a unique port number that may be used in creating TCP streams when the client does not need a well known port number of the local end.

7.3 Signals and Errors

```
TcpStream.Suspended: ERROR [why: TcpStream.SuspendReason];
```

```
TcpStream.Closed: ERROR;
```

Closed indicates the client tried to issue a **put** or a **close** after the connection was closed.

```
TcpStream.ListenTimeout: SIGNAL;
```

ListenTimeout is raised by the **Listen** procedure if a connection request does not arrive within the interval specified by **listenTimeout** in the **Listen** procedure. The client may decide either to stop listening or to resume the signal to continue listening.

7.4 Procedures

```
TcpStream.Listen: PROC [
    localPort: ArpaRouter.Port,
    listenTimeout: TcpStream.WaitTime ← TcpStream.infiniteWaitTime,
    receiveTimeout: TcpStream.WaitTime ← TcpStream.defaultWaitTime,
    typeOfService: ArpaSysParameters.TypeOfService ← [],
    security: ArpaSysParameters.Security ← NIL,
    options: Environment.Block ← [Environment.nullBlock,
    notifyListenerStarted: TcpStream.NotifyListenStartedProc ← NIL]
    RETURNS[tsH: TcpStream.Handle];
```

The client must tell a passive TCP the port on which to listen and provide the process to do so. This requirement is accomplished with **Listen**.

localPort is the port on which to listen for a connection request.

receiveTimeout is the timeout that will be set on the stream **tsH** after the connection has been established.

typeOfService is the service desired in the network. This level of service will be requested in all packets transmitted on streams resulting from the listen command. The request will override the system's default as defined by the interface **ArpaSysParameters**.

security is the required security for a connection. When TCP (through **Listen**) receives a connection request, it checks **security** to determine if it is allowed to honor the request. The default value of **NIL** implies that the system's default value for security should be used. If the incoming connection request carries a security classification that is higher than those specified in the listen request (or the system's default), the request for connection will be suppressed.

options are the TCP options. It is the client's responsibility to put the options into the options block in the proper format. If options are not to be used, then this parameter should be a null block.

notifyListenerStarted is called when all resources have been allocated and started for the connection and the port is ready to accept a connection attempt.

If the connection is honored, then **Listen** creates the stream and returns it to the client in **tsH**. The client regains control only when **Listen** returns with a valid stream, or when **listenTimeout** is exceeded, raising the signal **ListenTimeout**.

```
TcpStream.Make: PROC [
    local, remote: ArpaRouter.InternetAddress,
    localPort, remotePort: ArpaRouter.Port,
    establishConnection: BOOLEAN ← TRUE,
    timeout: TcpStream.WaitTime ← TcpStream.infiniteWaitTime,
    typeOfService: ArpaSysParameters.TypeOfService ← [],
    security: TcpStream.Security ← NIL,
    options: Environment.Block ← Environment.nullBlock],
    notifyListenerStarted: TcpStream.NotifyListenStartedProc ← NIL]
    RETURNS [tsH: TcpStream.Handle];
```

Make is the procedure used to solicit a TCP connection.

local is the address of the local machine. Clients can set **local** to **ArpaRouter.unknownInternetAddress**, and the TCP implementation fills in the correct address. **remote** is the address of the machine to connect. It can be any machine in the Internet, including the local machine.

localPort is the port on the local machine to use for the connection. It is commonly set to **uniquePort**. **remotePort** is the port on the remote machine to connect. This value may be a well known port number for some well known service in the internet (see **ArpaConstants**), or it may be a port known privately between the communicating partners. **establishConnection** determines whether the connection is to be active or passive. If it is **TRUE**, the local end will actively solicit the connection, else the local end will passively listen for a connection request. Note that the **Listen** procedure is the usual method of creating a passive listener.

timeout is the amount of time to wait for a connection to happen. If the timeout is exceeded, then the **Failed** error is raised. **precedence** is the service precedence desired in the network.

typeOfService is the service desired in the network. This level of service will be requested in all packets transmitted on streams resulting from the **Make** procedure call. The request will override the system's default as defined by the interface **ArpaSysParameters**.

security is the required security for a connection. When TCP receives a packet, it checks **security** to determine if it is allowed to pass the data to the client. The default value of **NIL** implies that the system's default value for security should be used. If the incoming packet carries a security classification higher than those specified in the make request (or the system's default) the request for connection will be suppressed.

options are the TCP options to be used. It is the client's responsibility to put the options into the options block in the proper format. If options are not to be used, then this parameter should be a null block.

notifyListenerStarted is called when all resources have been allocated and started for the connection and the port is ready to accept a connection attempt. This will only be used when creating a passive listener.

If the connection is successfully established (the error **Failed** is not raised), then the return value **tsH** identifies a connection that is ready to be used by the client. Operations on this stream are executed through the procedures in the object pointed to by **tsH**.

7.5 Restrictions

The **options** parameter in the **Make** procedure is currently ignored, since only the maximum segment size option exists, and it is set by TCP during the connection handshake.

8 ArpaTelnetStream

Telnet is a virtual terminal protocol to be used with the TCP/IP protocols. The ArpaTelnetStream interface provides Pilot clients with an interface to the Telnet Protocol defined by RFC854 and telnet options defined by RFCs 855 to 861.

References: *RFC854 TELNET Protocol Specification, Postel, May, 1983.*

RFC855 TELNET Option Specification, Postel, May, 1983.

RFC856 TELNET Binary Transmission, Postel, May, 1983.

RFC857 TELNET Echo Option, Postel, May, 1983.

RFC858 TELNET Suppress Go Ahead Option, Postel, May, 1983.

RFC859 TELNET Status Option, Postel, May, 1983.

RFC860 TELNET Timing MarkOption, Postel, May, 1983.

RFC861 TELNET Extended Options - List, Postel, May, 1983.

RFC960 Assigned Numbers, Reynolds, December, 1985.

8.1 Types and constants

```
ArpaTelnetStream.Handle: TYPE = LONG POINTER TO ArpaTelnetStream.Object;

ArpaTelnetStream.Object: TYPE = RECORD [
  options: ArpaTelnetStream.Options,
  getByte: ArpaTelnetStream.GetByteProc,
  putByte: ArpaTelnetStream.PutByteProc,
  get: ArpaTelnetStream.GetProc,
  put: ArpaTelnetStream.PutProc,
  push: ArpaTelnetStream.PushProc,
  delete: ArpaTelnetStream.DeleteProc,
  getTimeout: ArpaTelnetStream.GetTimeoutProc,
  setTimeout: ArpaTelnetStream.SetTimeoutProc,
  setInputOptions: ArpaTelnetStream.SetInputOptionsProc,
  flushDataLine: ArpaTelnetStream.FlushDataLineProc,
  setTerminalType: ArpaTelnetStream.SetTerminalTypeProc,
  performAction: ArpaTelnetStream.ActionProc,
  setOption: ArpaTelnetStream.SetOptionProc];
```

A telnet **Handle** is modeled after the Pilot stream handle interface, and the procedures it contains are similar to the Pilot stream interface.

The **options** record contains settings of a variety of user parameters.

getByte returns the next byte of data in the data stream. If no data is pending, then **getByte** waits for an infinite amount of time if no time out was set or waits the amount of time specified in the **setTimeout** procedure. **getByte** also returns the reason that the procedure is returning in the field **code**. In most cases, this is set to **normal**, but if some event occurs that forces the procedure to return, this is noted in the **code** field.

putByte places one byte of data on the out going telnet connection. Setting the **push** flag to **TRUE** has the same effect as the **sendNow** procedure on a Pilot stream: the data is flushed

from the sending side to the receiving side. If the **push** flag is TRUE, then the TCP data buffers are flushed. This operation is expensive and should be done only when necessary.

The **get** and **put** procedures are similar to the **getbyte** and **putByte** procedures, except they operate on blocks rather than bytes. The same comments on the **push** boolean apply for **put** as well as for **putByte**.

push causes all buffered data to be sent to the telnet partner. This is the same operation that is done when **put** or **putByte** procedures are called with the **push** boolean set toTRUE.

getTimeout returns the **timeout** that is set on the telnet connection.

setTimeout sets the amount of time the telnet connection waits on a **get** operation before returning with a timeout reason or before raising the timeout signal. If this procedure is called with a value of 0, then the timeout in effect is infinite. The length of time the **get** process can wait is limited to about 16 minutes.

setInputOptions is used to set the various options described below.

flushDataLine flushes the incoming data stream of all pending data.

setTerminalType registers the client's desired terminal type for the current telnet connection. This procedure should be called before any terminal-type option negotiation with the remote host begin. The terminal type information then is used to answer remote terminal-type queries.

performAction sends telnet control information to the remote end.

setOption negotiates telnet options with the remote host.

```
ArpatelnetStream.Options: TYPE = RECORD [
    signalTimeOut: BOOLEAN ← TRUE,
    signalOnGoAhead: BOOLEAN ← FALSE,
    signalOnEraseLine: BOOLEAN ← FALSE,
    signalOnEraseChar: BOOLEAN ← FALSE,
    signalOnAbort: BOOLEAN ← FALSE,
    signalOnInterrupt: BOOLEAN ← FALSE,
    signalOnBreak: BOOLEAN ← FALSE,
    signalOnShortBlock: BOOLEAN ← FALSE,
    willEcho: BOOLEAN ← FALSE,
    willBinary: BOOLEAN ← FALSE,
    willStatus: BOOLEAN ← FALSE,
    willSupGA: BOOLEAN ← FALSE,
    willTimeMark: BOOLEAN ← FALSE,
    willTerminalType: BOOLEAN ← FALSE,
    willEOR: BOOLEAN ← FALSE,
    willEXOPL: BOOLEAN ← FALSE];
```

Options decides the way in which the telnet connection operates and how the client is notified about connection events. The options that decide how the user is notified about telnet events are **signalTimeOut**, **signalOnGoAhead**, **signalOnEraseLine**, **signalOnEraseChar**, **signalOnAbort**, **signalOnInterrupt**, **signalOnBreak**, and **signalOnShortBlock**. The options that govern the way a telnet connection responds to option requests from a connection partner are **willEcho**, **willBinary**, **willStatus**, **willSupGA**, **willTimeMark**, **willTerminalType**, **willEOR**, and **willEXOPL**.

The signal options, when true, cause the client to be signaled of events rather than notified in the return arguments of the **get** call.

The other booleans govern the telnet options exported to remote. Only the options supported by the client implementation should be set to **TRUE**. A **TRUE** option means the client is willing to negotiate that particular option with the remote, and the client supports that option in a bi-directional manner.

```
ArpatelnetStream.ReturnCode: TYPE = {
    abort, binary, break, doExtendedOptionsList, doStatus, doTerminalType, echo,
    endOfRecord, eraseChar, eraseLine, goAhead, interrupt, normal, sendStatus,
    sendTerminalType, shortBlock, status, supGA, terminalTypeIs, timeMark,
    willBinary, willEcho, willEndOfRecord, willExtendedOptionsList, willStatus,
    willSupGA, willTerminalType, willTimeMark};

ReturnRecord: TYPE = RECORD [
    returnCode: ReturnCode ← normal,
    argument: SELECT OVERLAID ReturnCode FROM
        binary, doExtendedOptionsList, doStatus, doTerminalType,
            echo, endOfRecord, normal, sendStatus, sendTerminalType,
            supGA,timeMark, willBinary, willEcho, willEndOfRecord,
            willExtendedOptionsList, willStatus, willSupGA,
            willTerminalType, willTimeMark => [on: BOOLEAN ← FALSE],
        terminalTypeIs, status => [hostStatus: ArpaTelnetStream.HostStatusRecord],
        ENDCASE];
```

The **ReturnRecord** is returned by all **get** operations, passing control information received from the remote connection to the client.

The return codes are described below.

abort The remote side of the telnet connection indicated that all the queued output should be suspended but that the currently running process should continue.

binary The telnet partner requests or acknowledges the local end to start treating both sending and receiving data in binary mode.

break Same as pressing the Break key (128 decimal).

doExtendedOptionsList
The telnet partner requests or acknowledges the local end to begin negotiating options that are on the "Extended Options Lists".

doStatus The telnet partner wishes to be able to send requests for status of option information or confirms that it is willing to honor such requests.

doTerminalType
The telnet partner is willing to receive terminal type information during terminal type sub-negotiation.

echo The telnet partner requestes the local end to begin echoing, or confirms that it is willing to be echoed by the local end.

endOfRecord is unused.

eraseChar The remote side of the telnet connection has sent an erase character code. The user should treat this as if an **eraseChar** character were typed to the local stream.

eraseLine The remote side of the telnet connection has sent an erase line code. The user should treat this as if an **eraseLine** character were typed to the local stream.

goAhead The telnet partner indicates that all the data has been sent and it is now waiting for data.

interrupt The remote side of the telnet connection indicated that the current process should be cancelled.

normal The procedure is returning because it has exhausted the space provided for the results of the **get** operation. The byte(s) received from the remote host is pure data with no control information.

sendStatus The telnet partner requests the local end to send its perception of the options used in the current connection.

sendTerminalType
The telnet partner requests the local end to send its terminal type.

shortBlock is unused.

status The telnet partner sends its perception of the options used in the current connection. The status information is contained in the return record.

supGA The telnet partner requests or acknowledges the local end to start supressing the telnet Go-Ahead (GA) character when transmitting data.

terminalTypeIs
The return record contains the terminal type sent by the telnet partner in response to the client's send terminal request. The terminal type is an Ascii string and should conform to RFC 940 – Assigned Numbers.

timeMark The telnet partner requests the local end to return a WILL-TIMING-MARK in the data stream at the appropriate place.

willBinary The telnet partner requests for permission or agrees to begin transmitting data in binary mode.

willEcho The telnet partner requests permission or agrees to begin echoing for the local end.

willEndOfRecord unused.

willExtendedOptionsList
The telnet partner requests permission or acknowledges that it will begin negotiating options that are on the "Extended Options List".

willStatus
The telnet partner agrees to send status information in response to sendStatus requests.

willSupGA The telnet partner requests permission or agrees to begin supressing GA character when transmitting.

willTerminalType
The telnet partner is willing to send terminal type information during terminal type sub-negotiation.

willTimeMark

The telnet partner assures the local server that it is doing time mark synchronization as requested.

```
ArpaTelnetStream.HostStatusRecord: TYPE = RECORD [
    remote: PACKED ARRAY ArpaTelnetStream.OptionsEnum OF BOOLEAN ← ALL[FALSE],
    local: PACKED ARRAY ArpaTelnetStream.OptionsEnum OF BOOLEAN ← ALL[FALSE],
    Terminal: LONG STRING ← NIL];
```

The **HostStatusRecord** is a structure that is used to hold option status of a telnet connection. The remote field contains information on options that the telnet partner currently supports. The local field contains information on options that the local end supports. The **Terminal** is an Ascii string specifying the terminal type which the connection is supporting. This field may be **NIL** if no terminal type is set.

```
ArpaTelnetStream.OptionsEnum: TYPE = {
    Binary, Echo, EOR, EXOPL, Status, SupGA,TerminalType, TimeMark};
```

The **OptionsEnum** is an enumeration of all the telnet options implemented by the **ArpaTelnetStream** interface.

```
ArpaTelnetStream.Action: TYPE = {
    abortOutput, areYouThere, break, eraseChar, eraseLine, GA, interruptProcess,
    sendStatusInfo, sendTerminalType, status};
```

Action is an enumeration of the control information the client can send.

```
ArpaTelnetStream.SetOptionType: TYPE = {
    binary, doStatus, doTerminalType, echo, EOR, extendedOptionsList, supGA,
    timeMark, willBinary, willEcho, willEOR, willExtendedOptionsList, willStatus,
    willSupGA, willTerminalType, willTimeMark};
```

SetOptionType is an enumeration of all the option negotiation types the client can use.

```
ArpaTelnetStream.ActionProc: TYPE = PROCEDURE [
    sH: ArpaTelnetStream.Handle, action: ArpaTelnetStream.Action];
```

ActionProc is a procedure type that allows the client to send control information to the telnet partner. **sH** is the current telnet session handle, and **action** contains the specific operation to perform.

```
ArpaTelnetStream.DeleteProc: TYPE = PROCEDURE [sH: ArpaTelnetStream.Handle];
```

DeleteProc is a procedure type that allows the client to delete the telnet session specified by parameter **sH**.

```
ArpaTelnetStream.FlushDataLineProc: TYPE = PROCEDURE [sH: ArpaTelnetStream.Handle];
```

FlushDataLineProc is a procedure type that allows client to flush the incoming stream of all pending data.

```
ArpaTelnetStream.GetByteProc: TYPE = PROCEDURE [sH: ArpaTelnetStream.Handle]
    RETURNS [byte: Environment.Byte, code: ArpaTelnetStream.ReturnCode];
```

GetByteProc is a procedure type that returns the next byte of data from the current telnet data stream along with the appropriate status code.

```
ArpaTelnetStream.GetProc: TYPE = PROCEDURE [
    sH: ArpaTelnetStream.Handle, block: Environment.Block]
    RETURNS [bytesTransferred: CARDINAL, code: ArpaTelnetStream.ReturnCode];
```

GetProc is a procedure type that allows the client to read data from the incoming telnet stream on a per block basis. The data is stored in **block**. The number of bytes transferred and the status code of the transfer are returned.

```
ArpaTelnetStream.GetTimeoutProc: TYPE = PROCEDURE [sH: ArpaTelnetStream.Handle]
    RETURNS [timeout: TcpStream.WaitTime];
```

GetTimeoutProc is a procedure type that returns the time out value that is set on the telnet stream specified by **sH**.

```
ArpaTelnetStream.PutByteProc: TYPE = PROCEDURE [
    sH: ArpaTelnetStream.Handle, byte: Environment.Byte, push: BOOLEAN];
```

PutByteProc is a procedure type that allows client to place one byte of data on the out going telnet stream. When **push** is set to **TRUE**, the byte is sent immediately.

```
ArpaTelnetStream.PutProc: TYPE = PROCEDURE [
    sH: ArpaTelnetStream.Handle, block: Environment.Block, push: BOOLEAN];
```

PutProc is a procedure type that allows client to send data to the telnet partner on a per block basis. When **push** is set to **TRUE**, the data are sent immediately.

```
ArpaTelnetStream.PushProc: TYPE = PROCEDURE [sH: ArpaTelnetStream.Handle];
```

PushProc is a procedure type that allows client to make sure all buffered data on the current telnet session is sent to the telnet partner.

```
ArpaTelnetStream.SetInputOptionsProc: TYPE = PROCEDURE [
    sH: ArpaTelnetStream.Handle, options: ArpaTelnetStream.Optioms];
```

SetInputOptionsProc is a procedure type that allows the client to modify local supported options after the telnet stream is created.

```
ArpaTelnetStream.SetOptionProc: TYPE = PROCEDURE [
    sH:ArpaTelnetStream.Handle,on:BOOLEAN,setOption:ArpaTelnetStream.SetOptionType];
```

SetOptionProc is a procedure type that allows the client to negotiate various options with the telnet partner. The **setOption** field contains the option to be negotiated, and the **on** field determines whether that option is to be turned on or off. This procedure raises **ArpaTelnetStream.Error** if the client negotiates an option that is not registered.

```
ArpaTelnetStream.SetTerminalTypeProc: TYPE = PROCEDURE [
    sH: ArpaTelnetStream.Handle, terminalType: LONG STRING]
    RETURNS [success: BOOLEAN];
```

SetTerminalTypeProc is a procedure type that allows the client to register its terminal type for the Telnet terminal type option negotiation.

```
ArpaTelnetStream.SetTimeoutProc: TYPE = PROCEDURE [
    sH: ArpaTelnetStream.Handle, timeout: TcpStream.WaitTime];
```

SetTimeoutProc is a procedure type that allows the client to set a time limit on data retrieval wait time.

8.2 Signals

```
ArpatelnetStream.Error: ERROR [reason: ArpatelnetStream.TelnetErrorReason];

ArpatelnetStream.TelnetErrorReason: TYPE = {
    doesntBinary, doesntEcho, doesntStatus, doesntTermType, timeout, doesntEOR,
    doesntTimeMark, doesntEXOPL, doesntSupGA};
```

The signal **Error** is raised either when the client tries to enable an option that is not supported by the telnet connection or when the timeout interval set by the client is reached on a **get** operation.

```
ArpatelnetStream.StreamAborted: ERROR [abortReason: ArpatelnetStream.AbortReason];

ArpatelnetStream.AbortReason: TYPE = {
    closing, timeout, noRouteToDestination, noServiceAtDestination, remoteReject,
    precedenceMismatch, securityMismatch, optionMismatch, transmissionTimeout,
    remoteServiceDisappeared, reset, other};

ArpatelnetStream.GoAhead: SIGNAL;

ArpatelnetStream.eraseLine: SIGNAL;

ArpatelnetStream.eraseChar: SIGNAL;

ArpatelnetStream.Abort: SIGNAL;

ArpatelnetStream.Interrupt: SIGNAL;

ArpatelnetStream.break: SIGNAL;
```

8.3 Procedures

```
ArpaTelnetStream.AbortOutput: PROCEDURE [sH: ArpaTelnetStream.Handle] = INLINE {
    sH.performAction[sH, abortOutput]};
```

AbortOutput cancels the output of a remote process if the connected system supports output abort; otherwise, the process continues to completion.

```
ArpatelnetStream.AreYouThere: PROCEDURE [sH: ArpatelnetStream.Handle] = INLINE {
    sH.performAction[sH, areYouThere]};
```

AreYouThere forces the remote host to send some visible signal (character or string) that the connection is still active. The character or string is seen on the **get** operation.

```
ArpaTelnetStream.Binary: PROCEDURE [sH: ArpaTelnetStream.Handle, on: BOOLEAN] = INLINE {
    sH.setOption[sH, on, binary]};
```

Binary sends a DO BINARY if **on** is TRUE , else it sends a WON'T BINARY (Binary option negotiation).

```
ArpaTelnetStream.Break: PROCEDURE [sH: ArpaTelnetStream.Handle] = INLINE {
    sH.performAction[sH, break]};
```

Break sends the telnet break character to the remote host.

```
ArpaTelnetStream.Create: PROCEDURE [
    input: TcpStream.Handle,
    options: ArpaTelnetStream.Options,
    addLFToCR: BOOLEAN ← TRUE]
    RETURNS [telnetStream: ArpaTelnetStream.Handle];
```

Create sets up a stream-like connection to a remote host. The following parameters are needed: a TcpStream.**Handle** to the connection which provides the data in the field **input**; the set of options which describes how the telnet stream appears to the user and to the telnet connection partner; the boolean **addLFToCR**, which defaults to TRUE if not supplied. This boolean should be set to FALSE when the client wishes to provide lines ending only in an Ascii carriage return. The telnet implementation adds the additional Ascii line feed (LF) to make the line a valid telnet line.

This procedure returns a **Handle** containing the procedures which are the telnet stream.

```
ArpaTelnetStream.Delete: PROCEDURE [sH: ArpaTelnetStream.Handle] = INLINE {sH.delete[sH]};
```

Delete is called before closing the telnet connection to free up local storage and destroy the **Handle** passed in by the **Create** procedure.

```
ArpaTelnetStream.DoStatus: PROCEDURE [sH: ArpaTelnetStream.Handle, on: BOOLEAN] =
    INLINE {sH.setOption[sH, on, doStatus]};
```

DoStatus sends a DO STATUS when **on** is TRUE , or else it sends a DON'T STATUS (Status option negotiation).

```
ArpaTelnetSteam.DoTerminalType:PROCEDURE[sH:ArpaTelnetStream.Handle, on: BOOLEAN] =
    INLINE {sH.setOption[sH, on. doTerminalType] };
```

DoTerminalType sends a DO TERMTYPE if **on** is TRUE , or else it sends a DON'T TERMTYPE (Terminal type option negotiation).

```
ArpaTelnetStream.Echo: PROCEDURE [sH: ArpaTelnetStream.Handle, on: BOOLEAN] = INLINE {
    sH.setOption[sH, on, echo]};
```

Echo sends a DO ECHO if **on** is TRUE, or else it sends a DON'T ECHO (Echo option negotiation).

```
ArpaTelnetStream.EOR: PROCEDURE [sH: ArpaTelnetStream.Handle, on: BOOLEAN] = INLINE {
    sH.setOption[sH, on, EOR]};
```

EOR sends a DO EOR when **on** is TRUE, or else it sends a DON'T EOR (End-OF-Record option negotiation).

```
ArpatelnetStream.EraseChar: PROCEDURE [sH: ArpatelnetStream.Handle] = INLINE {
    sH.performAction[sH, eraseChar]};
```

EraseChar is used instead of BS SP to do an erase of the last character. On many systems the character BS does the correct operation.

```
ArpatelnetStream.EraseLine: PROCEDURE [sH: ArpatelnetStream.Handle] = INLINE {
    sH.performAction[sH, eraseLine]};
```

EraseLine erases the last line typed (to the last CRLF).

```
ArpaTelnetStream.ExtendedOptionsList: PROCEDURE [
    sH: ArpaTelnetStream.Handle, on: BOOLEAN] =
    INLINE {sH.setOption[sH, on, extendedOptionsList]};
```

ExtendedOptionsList sends a DO EXOPL if **on** is TRUE, or else it sends a DON'T EXOPL (Extended-Options-List option negotiation).

```
ArpatelnetStream.FlushDataLine: PROCEDURE [sH: ArpatelnetStream.Handle] =
    INLINE {sH.flushDataLine[sH]};
```

FlushDataLine flushes the incoming data stream's internal buffers of all pending data.

```
ArpatelnetStream.GetByte: PROCEDURE [sH: ArpatelnetStream.Handle]
    RETURNS [byte: Environment.Byte, code: ArpatelnetStream.ReturnRecord] = INLINE {
    [byte, code] ← sH.getByte[sH]};
```

GetByte returns the next byte of data in the data stream. If no data is pending, then **GetByte** waits the amount of time set in the **SetTimeout** procedure, or an infinite amount of time if no timeout was set. **GetByte** also returns control information plus option negotiation requests and responses from the telnet partner. (Only client registered options are passed on to the client; unregistered ones are rejected automatically by the telnet implementation). The returned **byte** field is valid only when the **code** returned is **normal**.

```
ArpatelnetStream.GetBlock:PROCEDURE[sH:ArpatelnetStream.Handle, block: Environment.Block]
    RETURNS [bytesTransferred: CARDINAL, code: ArpatelnetStream.ReturnRecord];
```

GetBlock is similar to **GetByte** except that it reads data on a per block basis.

```
ArpatelnetStream.GetTimeout: PROCEDURE [sH: ArpatelnetStream.Handle]
    RETURNS [timeOut: TcpStream.WaitTime] =
    INLINE {RETURN[sH.getTimeout[sH]]};
```

GetTimeout returns the timeout that is set on the telnet connection.

```
ArpaTelnetStream.GA: PROCEDURE [sH: ArpaTelnetStream.Handle] = INLINE {
    sH.performAction[sH, GA]};
```

GA sends the go ahead signal on the telnet connection.

```
ArpaTelnetStream.InterruptProcess: PROCEDURE [sH: ArpaTelnetStream.Handle] = INLINE {
    sH.performAction[sH, interruptProcess]};
```

InterruptProcess interrupts a remote process if the connected system can interrupt the process.

```
ArpaTelnetStream.Push: PROCEDURE [sH: ArpaTelnetStream.Handle] = INLINE {sH.push[sH]};
```

Push causes all buffered data to be sent to the telnet partner. This is the same operation that is done when **PutBlock** or **PutByte** is called with the **push** boolean set to **TRUE**.

```
ArpaTelnetStream.PutByte: PROCEDURE [
    sH: ArpaTelnetStream.Handle, byte: Environment.Byte, push: BOOLEAN] =
    INLINE {sH.putByte[sH, byte, push]};
```

PutByte places one byte of data on the out-going telnet connection. Setting the **push** flag to **TRUE** has the same effect as the **sendNow** procedure on a Pilot stream, and the data is flushed from the sending side to the receiving side. The **push** flag generates a TCP Push flag. This operation is expensive and should be done only when neccesary.

```
ArpaTelnetStream.PutBlock: PROCEDURE [
    sH: ArpaTelnetStream.Handle, block: Environment.Block, push: BOOLEAN] =
    INLINE {sH.put[sH, block, push]};
```

PutBlock is similar to **PutByte**, except that it sends data on a per block basis.

```
ArpaTelnetStream.SendStatusInfo: PROCEDURE [sH: ArpaTelnetStream.Handle] = INLINE {
    sH.performAction[sH, sendStatusInfo]};
```

SendStatusInfo sends the local option status information to the telnet partner.

```
ArpaTelnetStream.SendTerminalType: PROCEDURE [sH: ArpaTelnetStream.Handle = INLINE {
    sH.performAction[sH, sendTerminalType]};
```

SendTerminalType allows the client to request the telnet partner to send terminal type information. (Terminal type sub-negotiation).

```
ArpaTelnetStream.SetInputOptions: PROCEDURE [
    sH: ArpaTelnetStream.Handle, options: ArpaTelnetStream.Options] =
    INLINE {sH.setInputOptions[sH, options]};
```

SetInputOptions is used to modify the registered options during create time.

```
ArpaTelnetStream.SetTerminalType: PROCEDURE [
    sH: ArpaTelnetStream.Handle, terminalType: LONG STRING]
RETURNS [success: BOOLEAN] = INLINE {RETURN[sH.setTerminalType[sH, terminalType]] };
```

SetTerminalType allows client to set the desired terminal type before starting terminal type negotiation with the telnet partner. The **terminalType** information is used as response to the telnet partner's send-terminal-type requests.

```
ArpaTelnetStream.SetTimeout: PROCEDURE [
    sH: ArpaTelnetStream.Handle, timeOut: TcpStream.WaitTime] =
    INLINE {sH.setTimeout[sH, timeOut]};
```

SetTimeout sets the amount of time the telnet connection waits on a **get** operation before returning with a timeout reason or raising the timeout signal. If this procedure is called with a value of 0, then the timeout, in effect, is infinite. The length of time the **get** process can wait is limited to about 16 minutes.

```
ArpaTelnetStream.SupGA: PROCEDURE [sH: ArpaTelnetStream.Handle, on: BOOLEAN] =
    INLINE {sH.setOption[sH, on, supGA]};
```

SupGA sends a DO SUPGA if **on** is TRUE, or else it sends a DON'T GUPGA (Supress-Go-Ahead option negotiation).

```
ArpaTelnetStream.Status: PROCEDURE [sH: ArpaTelnetStream.Handle]
    RETURNS [status: LONG POINTER TO HostStatusRecord] = INLINE {
    sH.performAction[sH, status]};
```

Status causes the remote site to send connection status information if this option is supported. This process is described by RFC 859.

```
ArpaTelnetStream.TimeMark: PROCEDURE [sH: ArpaTelnetStream.Handle, on: BOOLEAN] =
    INLINE {sH.setOption[sH, on, timeMark]};
```

TimeMark allows the client to insert time-mark characters in the telnet data stream. Depending on the value of the parameter **on**, it sends either a DO or DON'T TIMEMARK.

```
ArpaTelnetStream.WillBinary: PROCEDURE [sH: ArpaTelnetStream.Handle, on: BOOLEAN] = INLINE {
sH.setOption[sH, on, willBinary]};
```

WillBinary sends a WILL BINARY if **on** is TRUE, or else it sends a WON'T BINARY (Binary option negotiation).

```
ArpaTelnetStream.WillEcho: PROCEDURE [sH: ArpaTelnetStream.Handle, on: BOOLEAN] = INLINE {
sH.setOption[sH, on, willEcho]};
```

WillEcho sends a WILL ECHO if **on** is TRUE, or else it sends a WON'T ECHO (Echo option negotiation).

```
ArpaTelnetStream.WillEOR: PROCEDURE [sH: ArpaTelnetStream.Handle, on: BOOLEAN] = INLINE {
    sH.setOption[sH, on, willEOR]};
```

WillEOR sends a WILL EOR if **on** is TRUE, or else it sends a WON'T EOR (End-Of-Record option negotiation).

```
ArpaTelnetStream.WillExtendedOptionsList: PROCEDURE [
    sH: ArpaTelnetStream.Handle, on: BOOLEAN] = INLINE
    {sH.setOption[sH, on, willExtendedOptionsList] };
```

WillExtendedOptionsList sends a WILL EXOPL if **on** is TRUE, or else it sends a WON'T EXOPL (Extended Options List option negotiation).

```
ArpaTelnetStream.WillStatus: PROCEDURE [sH: ArpaTelnetStream.Handle, on: BOOLEAN] =
INLINE { sH.setOption[sH, on, willStatus] };
```

WillStatus sends a WILL STATUS if **on** is TRUE, or else it sends a WON'T STATUS (Status option negotiation.

```
ArpaTelnetStream.WillSupGA: PROCEDURE [sH:ArpaTelnetStream.Handle,on:BOOLEAN] = INLINE
{ sH.setOption[sH, on, willSupGA]};
```

WillSupGA sends a WILL SUPGA if **on** is TRUE, or else it sends a WON'T SUPGA (Supress-Go-Ahead option negotiation).

```
ArpaTelnetStream.WillTerminalType: PROCEDURE [sH: ArpaTelnetStream.Handle,
    on: BOOLEAN] = INLINE {sH.setOption[sH, on, willTerminalType]};
```

WillTerminalType sends a WILL TERMTYPE if **on** is TRUE, or else it sends a WON'T TERMTYPE (Terminal type option negotiation).

```
ArpaTelnetStream.WillTimeMark: PROCEDURE [sH: ArpaTelnetStream.Handle.
    on: BOOLEAN] = INLINE {sH.setOption[sH, on, willTimeMark]};
```

WillTimeMark allows the client to insert time-mark character in the appropriate place in the telnet data stream. Depending on the value of the parameter **on**, it sends either a WILL or WON'T TIMEMARK to the telnet partner.

9 TelnetListener

The TelnetListener interface provides Pilot clients with an interface to the Telnet Protocol defined by RFC854. The TelnetListener interface is used by clients needing to listen on a specified port for a telnet connection. Telnet is a virtual terminal protocol used with the TCP/IP protocols.

References: *RFC854 TELNET Protocol Specification, Postel, May, 1983.*

9.1 Types and constants

```
TelnetListener.ConnectProc: TYPE = PROCEDURE [
    sH: ArpaTelnetStream.Handle,
    underlyingStream: TcpStream.Handle,
    remoteAddr: ArpaRouter.InternetAddress];
```

ConnectProc is called by the telnet interface when a connection is received on the port specified in the **Listen** procedure.

```
TelnetListener.ConnectID: TYPE [2];
```

This ID is returned by the **Listen** procedure and is used to destroy a telnet listening connection.

9.2 Procedures

```
TelnetListener.Listen: PROCEDURE [
    connect: TelnetListener.ConnectProc,
    portNumber: ArpaRouter.Port,
    suppressLF: BOOLEAN ← FALSE]
    RETURNS [connectionID: TelnetListener.ConnectID];
```

Listen is called by the client to establish a telnet listening connection on the port specified in the field **portNumber**. The procedure that is called when a connection is received is passed in the field **connect**. If line feeds are to be suppressed every time a carriage return is seen (CRLF → CR), then the **suppressLF** boolean should be set to **TRUE**. This procedure returns the value **connectionID** to be used in destroying the telnet listener.

```
TelnetListener.StopListening: PROCEDURE [connectionID: TelnetListener.ConnectID];
```

StopListening destroys a listening connection started with the procedure **Listen**. **StopListening** is called with the **connectionID** returned by **Listen**.

10 ArpaFilingCommon

The **ArpaFilingCommon** interface provides types to be used by clients using the ARPA Filing interfaces. It defines a common set of filing definitions to be used by TFTP, ArpaFTP and ArpaFTPServer.

Types and constants are defined below.

```
ArpaFilingCommon.StatusCode: TYPE = {
    aborted, accessViolation, directoryNotFound, eof, exceedStorageAlloc,
    invalidFileName, fileBusy, fileNotFound, localFileError, mediumFull, ok, paramsError,
    undefined};
```

The **StatusCode** type is used to return information about the state of the local filing operation in a standard manner.

aborted The file action has been aborted by server.

accessViolation
The user did not have sufficient access rights to access the file.

directoryNotFound
The file action did not complete because the specified directory was not found in the current context.

eof The logical end of the file was reached successfully and there is no more data to retrieve.

exceedStorageAlloc
The file action failed because the user has exhausted his allocated space.

invalidFileName
The specified file name is not valid in the current context. (File name syntax error).

fileBusy The file action failed because the file is currently in use by another user. Try again later.

fileNotFound
The file action did not complete because the specified file was not found in the current context.

localFileError
The file action failed due to local file processing error.

mediumFull
The file action failed because the local filing medium is full.

ok The filing action completed successfully.

paramsError
The file failed due to syntax errors in the specified parameters.

undefined

A file error which does not fit into the above categories.

```
ArpaFilingCommon.PutProc: TYPE = PROCEDURE [
    fileStream: Stream.Handle,
    block: Environment.Block,
    eot: BOOLEAN ← FALSE,,
    clientData: LONG POINTER ← NIL]
    RETURNS [statusCode: ArpaFilingCommon.StatusCode];
```

The **PutProc** type is used as the callback procedure to a file storing operation. The field **fileStream** contains a stream to the currently active local file and is passed to the caller using some other protocol-specific operation. The field **block** contains the data to be stored. The field **eot** is set **TRUE** when the file transfer has ended. The field **clientData** is provided for client's own use. The field **statusCode** is returned with the appropriate code.

```
ArpaFilingCommon.GetProc: TYPE = PROCEDURE [
    fileStream: Stream.Handle, block: Environment.Block,
    clientData: LONG POINTER ← NIL]
    RETURNS [statusCode: ArpaFilingCommon.StatusCode,
    bytesTransferred: CARDINAL];
```

The **GetProc** type is used as the callback procedure to a file retrieval operation. The field **fileStream** contains a stream to the currently active local file and is passed to the caller using some other protocol-specific operation. The field **block** receives the data to be sent. The field **clientData** is provided for the client's own use. The field **statusCode** is returned with the appropriate code and the number of bytes transmitted is in the field **bytesTransferred**.

```
ArpaFilingCommon.CloseProc: TYPE = PROCEDURE [
    fileStream: Stream.Handle,
    deleteFile: BOOLEAN ← FALSE,
    fileName: LONG STRING ← NIL,
    clientData: LONG POINTER ← NIL];
```

The **CloseProc** type is used as a callback procedure in either a file retrieval or file storing operation. It is called when the operation has been completed. The field **fileStream** contains a stream to the specifed local file that was passed to the caller by some protocol-specific operation. The **deleteFile** field, when **TRUE** and when the file operation is storing, indicates that the file operation did not complete; the file stored may be incomplete and should be deleted. The field **fileName** contains the name of the file when the field **deleteFile** is **TRUE** and may provide a hint as to which file should be deleted. The **clientData** field is provided for the client's own use.

11 TFTP (Trivial File Transfer Protocol)

Trivial File Transfer Protocol (TFTP) is a simple file transfer protocol which is a client of the User Datagram Protocol (UDP). It can be used to transfer files between hosts implementing the Arpa protocols. See *RFC783* for a full description of this protocol.

References: *RFC764 Telnet Protocol, Postel, June, 1980.*

RFC783 The TFTP Protocol (Revision 2), Sollins, June, 1981.

11.1 Types and constants

```
TFTP.Modes: TYPE = {netascii, octet, mail};
```

TFTPModes is used to indicate the type of file being transferred.

netascii is Ascii as defined in *USA Standard Code for Information Interchange* with modifications specified in *RFC764*; it is 8-bit Ascii. **octet** is raw 8-bit bytes. **mail** is netascii characters sent to a user rather than a file.

```
TFTP.FileStreamProc: TYPE = PROCEDURE [
    fileName: LONG STRING,fileType: TFTP.TFTPModes]
    RETURNS [
        statusCode: ArpaFilingCommon.StatusCode,
        fileStream: Stream.Handle, put: ArpaFilingCommon.PutProc,
        get: ArpaFilingCommon.GetProc,
        closeProc: ArpaFilingCommon.CloseProc];
```

FileStreamProc is used by the server side of an TFTP connection to solicit information from the TFTP server client. The **put** and **get** callback procedures are used to store or retrieve file data from the client's file system. The **close** procedure is called when the file transfer is completed. For the store case, the **get** procedure need not be provided; for the retrieve case, the **put** procedure need not be provided. When the **FileStreamProc** is called, file type information is derived from the **fileType** field and the file name from the **fileName** field. Client filing errors are returned using the **statusCode** field.

```
TFTP.GetStreamProc: TYPE = PROCEDURE [fileName: LONG STRING]
    RETURNS [stream: Stream.Handle, fileError: BOOLEAN];
```

GetStreamProc is used by **Retrieve** for file stream creation.

11.2 Errors and signals

```
TFTP.TFTPError: ERROR [reason: TFTP.TFTPErrorReason, errorMsg: LONG STRING];

TFTP.TFTPErrorReason: TYPE = {aborted, undefined, fileNotFound, accessViolation,
    mediumFull, illegalOp, unknownTID, fileExists, noSuchUser, timeOut, hostError,
    localFileError};
```

Errors are defined as follows.

aborted	The current session is canceled.
undefined	Not defined; error message may help.

fileNotFound	File was not found at the remote location.
accessViolation	The remote file cannot be accessed.
mediumFull	The remote site's disk is full or allocation exceeded.
illegalOp	Received an illegal TFTP response.
unknownTID	Not used.
fileExists	File exists and cannot be overwritten.
noSuchUser	Not used.
timeOut	The TFTP session has timed out because the remote site has not responded.
hostError	Remote error.
localFileError	Error in acquiring the file for transmission.

The **errorMsg** is passed by the protocol and contains an English error message. The **errorMsg** is allocated from the zone passed into the interface by the client and should be freed by the client.

11.3 Procedures

```
TFTP.Send: PROCEDURE [
    toHost: ArpaRouter.InternetAddress,
    fileName: LONG STRING,
    fileStream: Stream.Handle,
    dataProc: ArpaFilingCommon.GetProc,
    zone: UNCOUNTED ZONE,
    rexmt: CARDINAL ← 5,
    timeOut: CARDINAL ← 25];
```

Send stores a file to a TFTP server. The **toHost** field has the address of the destination file. The **fileName** field has the name of the file on the remote server and should be in the file naming structure of the remote machine. **fileStream** is a stream to the local file to be stored. The callback procedure provided in **dataProc** is used to retrieve the file from the local file system.

This procedure may raise **TFTPError {... aborted, undefined, accessViolation, mediumFull, illegalOp, fileExists, timeOut...}**. Any error message strings returned by the signal **TFTPError** are allocated from **zone** and should be freed by the client.

The **rexmt** field gives the timeout between TFTP data packets and TFTP acknowledgements. The field **timeOut** gives the total timeout period for the TFTP connection.

```
TFTP.Retrieve: PROCEDURE [
    fromHost: ArpaRouter.InternetAddress,
    fileName, localName: LONG STRING,
    fileType: TFTP.TFTPModes,
    fileStreamProc: TFTP.GetStreamProc,
    zone: UNCOUNTED ZONE,
    rexmt: CARDINAL ← 5,
    timeOut: CARDINAL ← 25];
```

Retrieve retrieves a file from a TFTP server. The **fromHost** field has the address of the source of the file being retrieved. The **fileName** field has the name of the file on the remote server and should be in the file naming structure of the remote machine. The **fileStreamProc** is called when a connection is established to acquire the local filing stream handle.

If a local filing error is encountered when trying to acquire the local file, then the **fileStreamProc** should return a **NIL** stream handle and a value of **TRUE** in the **fileError** field. **Retrieve** may raise **TFTPError {... aborted, undefined, fileNotFound, accessViolation, illegalOp, timeOut...}**. Any error message strings returned by the signal **TFTPError** are allocated from **zone** and should be freed by the client.

The **rexmt** field gives the timeout between TFTP data packets and TFTP acknowledgements. The **timeOut** field gives the total timeout period for the TFTP connection.

```
TFTP.Register: PROCEDURE [
    storeFile: TFTP.FileStreamProc,
    retrieveFile: TFTP.FileStreamProc,
    zone: UNCOUNTED ZONE,
    rexmt: CARDINAL ← 5,
    timeOut: CARDINAL ← 25];
```

Register is used for server side filing implementation. The procedures registered are called in the following instances. **StoreFile** is called when a request to write is received by the server. **RetrieveFile** is called when a request to read is received by the server. The **rexmt** field gives the timeout between TFTP data packets and TFTP acknowledgements. The field **timeOut** gives the total timeout period for the TFTP connection.

Only one client should register procedures. Other clients who register procedures overwrite the previous procedures.

```
TFTP.UnRegister: PROCEDURE;
```

UnRegister is called to suspend TFTP server operations.

12 ArpaFTP

FTP is a file transfer protocol running on top of the TCP/IP protocols. The ArpaFTP interface provides Pilot clients with an interface to the File Transfer Protocol (FTP) defined by RFC959.

References: *RFC959 File Transfer Protocol, Postel, October, 1985.*

12.1 Types and constants

```
ArpaFTP.Handle: TYPE = LONG POINTER TO ArpaFTP.FTPObject;

ArpaFTP.FTPObject: TYPE;

ArpaFTP.FileTypeEnum: TYPE = {ascii, EBCDIC, image, local8, other};
```

The **FileTypeEnum** defines the file types understood by FTP.

ascii is the default file type, intended for transferring text files; **ascii** is defined in the telnet specification to be the lower half of an 8-bit code set (the most significant bit is zero). **EBCDIC** is not supported. **image** is used for the transfer of binary or compressed data. **local8** is used for the transfer of data that has a logical byte size of eight. **other** accommodates other data representations and is supported for 8-bit bytes.

```
ArpaFTP.FileFormatEnum: TYPE = {nonPrint, telnet, asa};
```

FileFormatEnum defines the set of format control options that can be used with the file types **Ascii** and **EBCDIC**.

nonPrint is the default formatting option and indicates there is no formatting in the file. **telnet** indicates that the file contains vertical format controls (such as >CR>, <LF>, <NL>, <VT>, <FF>). **asa** indicates that the file contains asa (FORTRAN) vertical control characters (see RFC 740 or Communications of the ACM, Vol. 7, No. 10, p. 606, October 1964).

```
ArpaFTP.FileStructureEnum: TYPE = {file, record, page};
```

FileStructureEnum defines the set of file structures that are known to FTP.

file is the defaulted file structure. **record** and **page** are not supported.

```
ArpaFTP.TransmissionModeEnum: TYPE = {stream, block, compressed};
```

TransmissionModeEnum defines the set of data transmission types known to FTP.

stream is the default transmission mode; the data is transmitted as a stream of bytes. **block** and **compressed** are not supported.

```
ArpaFTP.Options: TYPE = RECORD [
    fileType: ArpaFTP.FileTypeEnum ← ascii,
    fileFormat: ArpaFTP.FileFormatEnum ← nonPrint,
    fileByteSize: CARDINAL ← 8,
    fileStructure: ArpaFTP.FileStructureEnum ← file,
    transmissionMode: ArpaFTP.TransmissionModeEnum ← stream,
    modeChanged: BOOLEAN ← FALSE,
    fileTypeChanged: BOOLEAN ← FALSE,
    fileStructureChanged: BOOLEAN ← FALSE,
    optionsChanged: BOOLEAN ← FALSE];
```

The **Options** record sets various options allowed by the FTP protocol. Not all options are available on all hosts. When options are changed, the **optionschanged** field for the appropriate option should be set to TRUE (for example, when **transmissionMode** is changed, the **modeChanged** boolean should be set to TRUE) and the **optionsChanged** boolean should be set to TRUE. The **fileByteSize** is the size of the data bytes of a file. Only a byte size of eight is supported.

```
ArpaFTP.defaultOptions: ArpaFTP.Options = [ascii, nonPrint, 8, file, stream];
```

defaultOptions can be used to set the **options** field in **Store** and **Retrieve**.

```
ArpaFTP.ListStyle: TYPE = {verbose, terse};
```

The **ListStyle** type is used with the **List** procedure to indicate whether **verbose** (all information that can be displayed about the file) or **terse** (only the file name) is wanted.

```
ArpaFTP.OutputListProc: TYPE = PROCEDURE [output: Environment.Block];
```

The **OutputListProc** is used with the **List** command to provide a callback procedure for the listing return information.

12.2 Errors and Signals

```
ArpaFTP.FTPError: ERROR [
    reason: ArpaFTP.FTPErrorReason, errorNumber: CARDINAL, errorString: LONG STRING];
```

The error **FTPError** is raised for all error conditions that arise on the local or remote machine. The error reasons are described below. The **errorNumber** field is reserved for error conditions that are reported by the remote FTP site. Error numbers follow the error number definitions as outlined by RFC 959. The **errorString** is also reserved for remote errors and is the human readable text message that accompanies the FTP **errorNumber**. The string **errorString** is allocated from a private zone and is deallocated on unwinding this error.

```
ArpaFTP.FTPErrorReason: TYPE = {accountNeeded, badCommandSequence,
exceedStorageAlloc, fileAccessProblem, fileBusy, invalidFileName, invalidHandle,
localFileError, mediumFull, noDataConn, noRouteToDestination,
noServiceAtDestination, noSuchUser, optionMismatch, other, paramsError,
precedenceMismatch, remoteFileError, remoteReject, reset, sercurityMismatch,
serverCommandError, serviceBusy, serviceUnavailable, TcpError, timeout,
transferAborted, undefined, unimplemented, unimplementedForParam,
userNotLoggedIn};
```

Errors are defined as follows.

accountNeeded
The user must supply an account number to complete the operation.

badCommandSequence
The commands issued are out of order and do not conform to the FTP protocol.

exceedStorageAlloc
The user has exhausted the allocated storage space.

fileAccessProblem
User does not have appropriate access rights for this file operation.

fileBusy The specified file is temporary unavailable. Try again later.

invalidFileName
The file name specified was invalid. (Illegal file name syntax).

invalidHandle
The requested action failed because the session handle is invalid.

localFileError
Some local filing error was encountered.

mediumFull
The file action failed because the storage medium is full.

noDataConn
The file action failed because no data connection exits or a data connection can not be opened.

noRouteToDestination
TCP level error. The stream failed or is suspended because the remote site can not be reached.

noServiceAtDestination
TCP level error. The stream failed or is suspended because there is no service at the destination site.

noSuchUser
The user name could not be found on the remote host.

optionMismatch
TCP level error. The stream failed or is suspended due to option(s) mismatch with remote.

other TCP level error. The stream failed due to some undefined problem.

paramsError
The requested action failed due to errors in the parameter(s) of the command line.

precedenceMismatch
TCP level error. The stream failed or is suspended because the precedence negotiated is incompatible with remote.

remoteFileError
Some remote file error was encountered.

remoteReject
TCP level error. The remote host rejects the connection request.

reset Telnet level error. The control stream is aborted due to a reset from remote.

sercurityMismatch
TCP level error. The stream failed or is suspended due to security mismatch with remote.

serverCommandError
The requested action failed due to illegal command syntax.

serviceBusy
The file service is unavailable temporarily, try again later.

serviceUnavailable
The file service is not available.

tcpError TCP level error. Some undefined TCP error was encountered.

timeout The operation timed out. The remote host may no longer be responding.

transferAborted
The file transfer operation is aborted. Connection closed.

undefined
The requested action failed due to some undefined error.

unimplemented
The indicated action is not implemented by the remote host.

unimplementedForParam
Command not implemented for that parameter.

userNotLoggedIn
The requested action failed because the user is not logged in.

12.3 Procedures

```
ArpaFTP.Abort: PROCEDURE [connectionHandle: ArpaFTP.Handle];
```

Procedure **Abort** cancels or terminates any outstanding FTP command, including data transfers, on the connection specified by **connectionHandle**. **Abort** raises **FTPError**.

```
ArpaFTP.ChangeWorkDir: PROCEDURE [directory: LONG STRING,
    connectionHandle: ArpaFTP.Handle];
```

ChangeWorkDir allows the client to specify a working directory for all subsequent file operations. The **directory** field is a path name for the desired working directory. It must

conform to the remote host's directory name format. The **connectionHandle** is the current file session. This procedure raises **FTPError**.

```
ArpaFTP.Create: PROCEDURE [
    destHost: ArpaRouter.InternetAddress,]
    RETURNS [ftpHandle: ArpaFTP.Handle];
```

Create allows the client to open an FTP session. The **destHost** field is the internet address of the remote FTP server. The **ftpHandle** is returned once a session is sucessfully established. This handle must be used in all subsequent FTP calls for this session. **Create** raises **FTPError** (TCP level errors, serviceBusy, serviceUnavailable) in cases where a connection can not be made.

```
ArpaFTP.Delete: PROCEDURE
    filePathName: LONG STRING, connectionHandle: ArpaFTP.Handle];
```

Delete allows the client to delete the file specified by **filePathName** on the remote host. The **connectionHandle** is the current session. **FTPError** is raised in case of error.

```
ArpaFTP.Destroy: PROCEDURE [connectionHandle: ArpaFTP.Handle];
```

Destroy disconnects the user's current filing session from the remote FTP server. It should only be used when the session can no longer continue due to errors. The **connectionHandle** becomes invalid after this operation. **Destroy** raises **FTPError**.

```
ArpaFTP.List: PROCEDURE [
    filePathName: LONG STRING,
    outputProc: ArpaFTP.OutputListProc,
    options: ArpaFTP.Options,
    outputStyle: ArpaFTP.ListStyle ← terse,
    connectionHandle: ArpaFTP.Handle];
```

The **List** procedure can be used to request that the FTP server send a list of the current filing context to the user. The **filePathName** field specifies a system specific file path name. If the **outputStyle** is **verbose**, all current information on the file or file group specifed by the field **filePathName** will be returned using the procedure **outputProc**. The **options** field describes the attributes of the file information to be transfered. If **outputStyle** is **terse** only the file name of each of the files specified by the **filePathName** will be returned. In either case the file information should be separated by either a CRLF or a Null character. The **outputProc** is used to send the received information to the client process. This procedure may raise the error **Error**.

```
ArpaFTP.Login: PROCEDURE [
    userName, userPassword, userAccount: LONG STRING,
    connectionHandle: ArpaFTP.Handle]
    RETURNS [success: BOOLEAN ← FALSE];
```

Login performs the logging on sequence required by the remote host using the provided parameters. The information should be the user's name, password, and account on the remote system. Only paramter(s) required by the remote host need to be specified. The **connectionHandle** field is the current FTP session handle. The procedure returns **TRUE** when the user logged on the remote host sucessfully and **FALSE** otherwise. This procedure raises **FTPError**.

```
ArpaFTP.Passive: PROCEDURE [connectionHandle: ArpaFTP.Handle]
    RETURNS [port: ArpaRouter.Port];
```

Passive provides client with an interface to the FTP **PASV** command. The **PASV** command is mainly used for third party file transfers. It asks the remote server to listen on a data port (not the default data port) and to wait for a connection instead of initiating one upon the receipt of a transfer command. The returned **port** value (from the remote server) is the port address it is listening on. **Passive** raises **FTPError**.

```
ArpaFTP.Quit: PROCEDURE [connectionHandle: ArpaFTP.Handle]
    RETURNS [success: BOOLEAN ← FALSE];
```

Quit disconnects the user's current filing session from the remote FTP server. This procedure returns **TRUE** when the remote acknowleges the disconnection request and **FALSE** otherwise. The **connectionHandle** becomes invalid after this operation regardless of the remote host's response.

```
ArpaFTP.ReInit: PROCEDURE [connectionHandle: ArpaFTP.Handle];
```

Procedure **ReInit** reinitializes the current FTP connection specified by **connectionHandle**. This procedure allows the client to change FTP user without having to drop the current connection and create a new one. This procedure raises **FTPError**.

```
ArpaFTP.Rename: PROCEDURE [
    from, to: LONG STRING, connectionHandle: ArpaFTP.Handle];
```

Rename provides the client with the facility to rename files on the remote host. The remote file specified in the **from** field will be renamed to the name specified in the **to** field. This procedure raises **FTPError**.

```
ArpaFTP.Retrieve: PROCEDURE [
    remoteFileName: LONG STRING,
    fileStream: Stream.Handle,
    putProc: ArpaFilingCommon.PutProc,
    options: ArpaFTP.Options,
    connectionHandle: ArpaFTP.Handle,
    host: ArpaRouter.InternetAddress ← LOOPHOLE[LONG[0]]
    port: ArpaRoute.Port ← LOOPHOLE[0]];
```

Retrieve allows the client to retrieve a file from an FTP server. The **remoteFileName** field is the name of the desired file at the remote site. The **remoteFileName** string must conform to the remote host's file name format. The **fileStream** is the stream of the file to be stored on the local file system. Procedure **putProc** is a callback procedure provided by the client that stores the remote file on the local file system on a per block basis. The **options** field describes the file format, file type, and transmission mode for this file transfer. The **connectionHandle** field is the current session handle. Both the **host** and **port** fields are parameters used in the passive mode. The **host** field contains the address of the host that initiates the data transfer, and the **port** field contains the data port for the file action. (See procedure **Passive** for more details). This procedure raises **FTPError**.

```
ArpaFTP.Store: PROCEDURE [
    remoteFileName: LONG STRING,
    fileStream: Stream.Handle,
    getProc: ArpaFilingCommon.GetProc,
    options: ArpaFTP.Options,
    connectionHandle: ArpaFTP.Handle,
    host: ArpaRouter.InternetAddress ← LOOPHOLE[LONG[0]]
    port: ArpaRoute.Port ← LOOPHOLE[0]];
```

The **Store** procedure allows the client to store a file to an FTP Server. The **remoteFileName** field is the desired name for the file at the remote side. The **remoteFileName** string must conform to the remote host's file name format. The **fileStream** is the stream of the file to be stored. Procedure **getProc** is a callback procedure provided by the client that retrieves the local file for remote storing on a per block basis. the **options** field describes the file format, file type, and transmission mode for this file transfer. The **connectionHandle** field is the current session handle. Both the **host** and the **port** fields are parameters used in the passive mode. The **host** field contains the address of the host that initiates the data transfer, and the **port** field contains the data port for the file action. (See procedure **Passive** for more details). Thi s procedure raises **FTPError**.

13 ArpaFTPServer

FTP is a file transfer protocol running on top of the TCP/IP protocols (RFC793 and RFC792). The ArpaFTPServer interface provides Pilot clients with an interface to the server side of the File Transfer Protocol (FTP) defined by RFC959.

References: *RFC740 NETRJS Proctocol - Appendix C, Braden, November, 1977.*

RFC792 Internet Control Message Protocol, Postel, September, 1981.

RFC793 Transmission Control Protocol, Postel, September, 1981.

13.1 Types and constants

```
ArpaFTPServer.Options: TYPE = RECORD [
    fileType: ArpaFTPServer.FileTypeEnum ← ascii,
    fileFormat: ArpaFTPServer.FileFormatEnum ← nonPrint,
    fileByteSize: CARDINAL ← 8,
    fileStructure: ArpaFTPServer.FileStructureEnum ← file];
```

The **Options** type defines the record passed to the client to indicate the method by which a file is retrieved or stored. The **fileByteSize** is the logical byte size of the file transferred. All files are transferred as 8-bit files, regardless of their logical byte size.

```
ArpaFTPServer.FileTypeEnum: TYPE = {ascii, EBCDIC, image, local8, other};
```

FileTypeEnum defines the set of file types that can be understood by FTP. **ascii** is the default file type and is intended for transferring text files; **ascii** is defined in the Telnet specification to be the lower half of an 8-bit code set (the most significant bit is zero). **EBCDIC** is not supported. **image** is used for the transfer of binary or compressed data. **local8** is used for the transfer of data that has a logical byte size of eight. **other** is used to accommodate other data representations and is supported for 8-bit bytes.

```
ArpaFTPServer.FileFormatEnum: TYPE = {nonPrint, telnet, asa};
```

FileFormatEnum defines the set of format control options that can be used with the file types **ascii and EBCDIC**. **nonPrint** is the default formatting option and indicates there is no formatting in the file. **telnet** indicates that the file contains vertical format controls; for example, <CR>, <LF>, <NL>, <VT>, <FF>. **asa** indicates that the file contains asa (FORTRAN) vertical control characters. (See RFC 740 and Communications of the ACM, Vol. 7, No. 10, p. 606, October 1964).

```
ArpaFTPServer.FileStructureEnum: TYPE = {file, record, page};
```

FileStructureEnum defines the set of file structures known to FTP. **file** is the defaulted file structure. **record** and **page** are not supported.

```
ArpaFTPServer.LoginInfoNeeded: TYPE = {
    name, nameAndPassword, nameAndPasswordAndAcct};
```

The type **LoginInfoNeeded** describes the types of login information required by the authentication mechanism on the FTP server. **name** means that only the user's name is required to use the FTP server. **nameAndPassword** means that the user's name and

password are required to use the FTP server. **nameAndPasswordAndAcct** means that the user must specify name, password, and account information to use the FTP server.

```
ArpaFTPServer.ListStyle: TYPE = {verbose, terse};
```

The **ListStyle** type is used with the **List** procedure to indicate whether **verbose** (all information that can be displayed about the file) or **terse** (only the file name) is wanted.

```
ArpaFTPServer.FileStreamProc: TYPE = PROCEDURE [
    fileName: LONG STRING, options: ArpaFTPServer.Options,
    conversationHandle: LONG POINTER]
    RETURNS [
        statusCode: ArpaFilingCommon.StatusCode,
        fileStream: Stream.Handle,
        put: ArpaFilingCommon.PutProc,
        get: ArpaFilingCommon.GetProc,
        closeProc: ArpaFilingCommon.CloseProc,
        clientData: LONG POINTER ← NIL];
```

The **FileStreamProc** procedure is the type of procedure the client registers with the **ArpaFTPServer** so that its facility can be invoked in the case of store or retrieve operations.

The **fileName** field contains the name of the file for the operation. The **options** field contains the attributes of the file. The **conversationHandle** field contains client information that was passed to the server at user logon time.

The server client returns permission (**ok**) to perform the requested file operations, or reason of denial in the **statusCode** field. Data transfer will not start unless the returned **statusCode** is **ok**. The **fileStream** field is set to be the stream of the file specified in **fileName** when the operation request is granted. Both **put** and **get** are callback procedure provided by the client that do the actual storing and retrieving of the file on a per block basis. Procedure **get** need not be provided in a store operation and procedure **put** need not be provided in a retrieve operation.

Procedure **closeProc** is also a callback procedure provided by the client. It is called when the file transfer is completed. When **closeProc** is called, no other file operation is performed on the file described by **fileStream**.

clientData is provided by the client so the server can pass it along when calling **put**, **get**, or **closeProc**.

```
ArpaFTPServer.LogonProc: TYPE = PROCEDURE [
    userName, userPassword, userAccount: LONG STRING ← NIL,
    converstationHandle: LONG POINTER ← NIL]
    RETURNS [success: BOOLEAN, newConversationHandle: LONG POINTER];
```

A **LogonProc** procedure is the type of procedure a client registers with the **ArpaFTPServer** to login new users. The client returns **TRUE** when the specified information identifies an authenticated user. **LogonProc** is called at least once and at most three times depending on the client's registered **logonInfo** (name only, nameAndPassword, nameAndPasswordAndAcct). The **converstationHandle** contains client specified information that identifies a user session. **ArpaFTPServer** calls **LogonProc** with a NIL **converstationHandle** when the user and the session is newly established, and a valued **converationHandle** when the user is new but the session is reinitialized. In the case of

newly established session, the client creates a session, and returns the session handle in **newConversationHandle**. In the case of a reinitialized session, the client resets the state of the session for a new user, and sets **newConversationHandle** to the old **conversationHandle** provided by the **ArpaFTPServer**. The **newConversationHandle** is passed by the **ArpaFTPServer** in all subsequent calls so client can distinguish the different user sessions. The **ArpaFTPServer** rejects all filing calls that are made with an unauthenticated user.

```
ArpaFTPServer.QuitProc: TYPE = PROCEDURE [conversationHandle: LONG POINTER];
```

QuitProc is called at the end of a FTP session. The client may free any session information at this time. The field **conversationHandle** contains client information for the current session that was passed to the server at user logon time and should be invalidated by this call.

```
ArpaFTPServer.ReinitializeProc: TYPE = PROCEDURE [conversationHandle: LONG POINTER];
```

When **ReinitializeProc** is called by the FTP server, the client should consider the current session to be open but should set its state back to its initial values. This procedure is called when the remote user wishes to destroy the session but maintain the communication line as active. The field **conversationHandle** contains client information for the current session that was passed to the server at user logon time and should be invalidated by this call.

```
ArpaFTPServer.RenameProc: TYPE = PROCEDURE [
    from, to: LONG STRING,
    conversationHandle: LONG POINTER]
    RETURNS [statusCode: ArpaFilingCommon.StatusCode];
```

RenameProc is called by the FTP server when it receives a request to rename a file. The **from** field contains the current name of the file; the **to** field contains the new name of the file. The field **conversationHandle** contains client information that was passed to the server at user logon time. The client returns a code in **statusCode**.

```
ArpaFTPServer.AbortProc: TYPE = PROCEDURE [conversationHandle: LONG POINTER];
```

AbortProc is called by the FTP server when an abort is received from the FTP user. When **AbortProc** is called, the client should suspend and terminate all active processes for the specified session. The field **conversationHandle** contains client information that was passed to the server at user logon time.

```
ArpaFTPServer.ChangeWorkDirProc: TYPE = PROCEDURE [
    directory: LONG STRING ← NIL;
    conversationHandle: LONG POINTER ← NIL]
    RETURNS [statusCode: ArpaFilingCommon.StatusCode];
```

ChangeWorkDirProc is called by **ArpaFTPServer** when a CWD command is received from the user. The client should check the validity of the **directory** field and return the status in **statusCode**. The **conversationHandle** contains client specified information that has been passed to the **ArpaFTPServer** at user logon time. CWD allows both relative and absolute directory paths. If **directory** starts with either a "/", "(", "<", or "[", then the directory path with be reset, else the directory path will be set relative to the current working directory.

```
ArpaFTPServer.DeleteProc: TYPE = PROCEDURE [
    filePathName: LONG STRING,
    conversationHandle: LONG POINTER]
    RETURNS [statusCode: ArpaFilingCommon.StatusCode];
```

DeleteProc is called by the FTP server when it receives a request to delete a file. The field **filePathName** contains the name of the file to be deleted. The field **conversationHandle** contains client information that was passed to the server at user logon time. The client returns the termination status in the field **statusCode**.

```
ArpaFTPServer.ListProc: TYPE = PROCEDURE [
    filePathName: LONG STRING,
    outputProc: ArpaFTPServer.OutputListStringProc,
    outputStyle: ArpaFTPServer.ListStyle ← terse,
    conversationHandle: LONG POINTER]
    RETURNS [statusCode: ArpaFilingCommon.ReturnCode];
```

ListProc is called by the FTP server when it receives a request to list the contents of the files specified by the field **filePathName**. This field may contain wildcard and expansion symbols native to the local file system.

The type and amount of information returned is specified by the field **outputStyle**. When the value of this field is **terse**, the client returns the name of the files specified by the **filePathName** field. When the value of the field **outputStyle** is **verbose**, the client returns a complete list of information about the file or files specified by the **filePathName** field. The procedure specified by **outputProc** is used to return this information to the caller. Individual file information is separated by the Ascii character string CR and LF.

The field **conversationHandle** contains client information that was passed to the server at user logon time.

The status of the list operation is returned in **statusCode**.

```
OutputListStringProc: TYPE = PROCEDURE [output: LONG STRING];
```

OutputListStringProc is used with **ListProc** to send list data to the remote site.

```
ArpaFTPServer.FTPProcList: TYPE = RECORD [
    logon: ArpaFTPServer.LogonProc,
    quit: ArpaFTPServer.QuitProc,
    store: ArpaFTPServer.FileStreamProc,
    retrieve: ArpaFTPServer.FileStreamProc,
    reinitialize: ArpaFTPServer.ReinitializeProc,
    rename: ArpaFTPServer.RenameProc,
    abort: ArpaFTPServer.AbortProc,
    changeWorkDir: ArpaFTPServer.ChangeWorkDirProc,
    delete: ArpaFTPServer.DeleteProc,
    list: ArpaFTPServer.ListProc];
```

FTPProcList is used with the **Register** procedure to give the server a list of service commands to call when it receives services requests from a remote user. These procedures are described above.

13.2 Errors and Signals

```
ArpaFTPServer.FTPServerError: ERROR [
    reason: ArpaFTP.FTPErrorReason[serviceUnavailable..serviceUnavailable]];
```

FTPServerError is raised by clients to notify the **ArpaFTPServer** when service on the server side for a particular FTP session is unavailable due to errors or shortage of resources. Upon receiving this error signal, **ArpaFTPServer** ends the specified session by closing down the control connection in the case of errors, and rejects the request for a new FTP connection in the case of resource shortage.

13.3 Procedures

```
ArpaFTPServer.Register: PROCEDURE [
    ftpProcList: ArpaFTPServer.FTPProcList, logonInfo: ArpaFTPServer.LoginInfoNeeded];
```

Register initializes an FTP server process. Only one call to this procedure is valid without calling **UnRegister**. Multiple calls to this procedure without calling **UnRegister** may produce undefined results. Procedures passed in the field **ftpProcList** are used to satisfy service requests from remote users. The field **logonInfo** contains the value for the amount of information needed to authenticate remote users. A **NIL** value in the **FTPProcList** is considered as "command not implemented" and is reported as such to the FTP client on the remote host when that specified command is requested. Note that a minimum set of commands are required by the protocol.

```
ArpaFTPServer.UnRegister: PROCEDURE;
```

UnRegister terminates and unregisters the FTP server process initiated by a call to the procedure **Register**. **UnRegister** is currently not implemented.

14 ArpaFileName

The ArpaFileName interface is similar to the Mesa FileName interface. It provides a general data structure for dealing with file names.

14.1 Types

```
ArpaFileName.VirtualFilename,VFN: TYPE =
    LONG POINTER TO ArpaFileName.VirtualFilenameObject;

ArpaFileName.VirtualFilenameObject: TYPE = RECORD [
    host, directory, name, version: LONG STRING];
```

14.2 Signals

```
ArpaFileName.Error: SIGNAL;
```

Error is raised by ArpaFileName.AllocVFN and ArpaFileName.UnpackFilename when the client provides an invalid file name. A file name should have the following syntax, with all fields optional:

[host]dir1/dir2/../dirn/filename!version

">" can also be used as a directory delimiter.

14.3 Procedures

```
ArpaFileName.AllocVFN: PROCEDURE [LONG STRING] RETURNS [ArpaFileName.VirtualFilename];
```

The AllocVFN procedure allocates a new ArpaFileName.VirtualFilenameObject and parses its parameter into a ArpaFileName.VirtualFilename. Both the object and the strings in the object are allocated from the ArpaUtility.zone and must be deallocated by ArpaFileName.FreeVFN.

```
ArpaFileName.FreeFilename: PROCEDURE [s: LONG STRING];
```

The FreeFilename procedure frees the string allocated with ArpaFileName.PackFilename.

```
ArpaFileName.FreeVFN: PROCEDURE [ArpaFileName.VirtualFilename];
```

The FreeVFN procedure frees a ArpaFileName.VirtualFilenameObject. It also frees the component strings in the object to the ArpaUtility.zone. The VirtualFilenameObject must have been allocated by ArpaFileName.AllocVFN.

```
ArpaFileName.PackFilename: PROCEDURE [vfn: ArpaFileName.VFN,h,d,n,v: BOOLEAN ← FALSE]
    RETURNS [s: LONG STRING];
```

The PackFilename procedure converts the information in selected fields of a VirtualFilename into a string, adding appropriate delimiters when necessary. h, d, n, v indicate whether the host, directory, name, and version fields are to be included in the string returned. Hosts are delimited by [], directories are delimited by either > or /, and versions are preceded by !. If no version appears in vfn, enough room is left in s for a version at least six characters long. "<" receives no special treatment but is considered a normal

character in a file name field. s is allocated from the ArpaUtility.zone; it must be freed by the client with ArpaFileName.FreeFilename.

```
ArpaFilename.ResetVFN: PROCEDURE [
    vfn: ArpaFilename.VirtualFilename, h, d, n, v: BOOLEAN ← FALSE];
```

The ResetVFN procedure resets selected fields of a VirtualFilename to NIL and frees the associated storage to the ArpaUtility.zone. h, d, n, and v indicate whether the host, directory, name, and version fields are to be reset.

```
ArpaFileName.UnpackFilename: PROCEDURE [s: LONG STRING, vfn: ArpaFileName.VFN];
```

The UnpackFilename procedure parses a string into a VirtualFilename. If a directory is present in vfn and the directory in s does not begin with <, then the directory from s is appended, or else the directory is overwritten. UnpackFilename creates VirtualFilenames that no longer have a final > on the directory string. This procedure raises ArpaFileName.Error if the file name s cannot be parsed.

15 ArpaSMTP

The ArpaSMTP interface provides Pilot clients with an interface to the client side of the Simple Mail Transfer Protocol (SMTP) defined by RFC821.

References: *RFC821 Simple Mail Transfer Protocol, Postel, August, 1982.*

15.1 Types and constants

```
ArpaSMTP.Handle: TYPE = LONG POINTER TO ArpaSMTP.SMTPObject;

    ArpaSMTP.SMTPObject: TYPE;
```

Handle is a pointer to an **SMTPObject** representing a connection to a remote SMTP host.

```
ArpaSMTP.Recipients: TYPE = LONG POINTER TO ArpaSMTP.RecipientsSequence;

ArpaSMTP.RecipientsSequence: TYPE = RECORD [
    recipients: SEQUENCE length: CARDINAL OF LONG STRING];
```

The type **RecipientsSequence** is a sequence of the recipients to whom a particular message is addressed.

```
ArpaSMTP.InvalidRecipientList: TYPE = RECORD [
    invalidRecipients: SEQUENCE length: CARDINAL OF ArpaSMTP.InvalidRecipientRecord];
```

The type **InvalidRecipientList** is a sequence of all recipients to whom a post operation could not post.

```
ArpaSMTP.InvalidRecipientRecord: TYPE = RECORD [
    recipientName: LONG STRING ← NIL,
    errorReason: ArpaSMTP.SMTPErrorReason,
    errorString: LONG STRING ← NIL,
    errorNumber: CARDINAL];
```

An **InvalidRecipientRecord** is returned for each invalid recipient of a **Post** operation. The field **recipientName** is a pointer to the name string that was passed into the **Post** procedure.

The **errorReason** field is the translated error condition as received from the remote host. The **errorString** field is the error string as received from the remote host. The field **errorNumber** contains the error number reason of the remote reject. This number conforms to the error numbering scheme outlined in RFC821.

15.2 Signals

```
ArpaSMTP.SMTPError: SIGNAL [
    reason: ArpaSMTP.SMTPErrorReason,
    errorNumber: CARDINAL,
    errorstring: LONG STRING];
```

The error **SMTPError** is raised for all error conditions that arise on the local or remote machine. The **errorNumber** field is reserved for error conditions that are reported by the remote SMTP site. Error numbers follow the error number definitions as outlined by RFC 821. The **errorstring** is also reserved for remote errors and is the human readable text

message that accompanies the SMTP **errorNumber**. The string **errorstring** is allocated from a private zone and is deallocated on unwinding this error.

```
ArpaSMTP.SMTPErrorReason: TYPE = {addressTranslationError,
    insufficientSpaceOnRemote, invalidName, mailboxUnavailable, remoteError,
    remoteStorageAllocExceeded, serverCommandError, serviceUnavailable, tcpError,
    tcpTimeOut, transactionFailed, userNotLocal, transmissionTimeout,
    noRouteToDestination, remoteServiceDisappeared, reset, securityMismatch,
    precedenceMismatch};
```

Error reasons are described below.

addressTranslationError
: The remote host name passed is invalid.

insufficientSpaceOnRemote
: The remote site has insufficient space to process the mailing request.

invalidName
: The specified recipient name is invalid.

mailboxUnavailable
: The specified recipients mailbox is not available.

remoteError
: Some remote error.

remoteStorageAllocExceeded
: Remote mail storage allocation exceeded.

serverCommandError
: Error in the processing of the SMTP command.

serviceUnavailable
: The service must shut down.

tcpError Some TCP error on connection establishment.

tcpTimeOut
: TCP timeout on connection establishment; the remote server may no longer be responding.

transactionFailed
: Mail transaction failed.

userNotLocal
: User is not local to this remote machine; the accompanying error string may have an alternate path to the user.

transmissionTimeout
: The mail transfer connection timed out.

noRouteToDestination
: The route from the local socket to the remote socket has disappeared and another could not be found to use.

remoteServiceDisappeared
: The remote site does not support the FTP service.

securityMismatch
The connection attempt was made with a security level lower than that specified by the remote service.

precedenceMismatch
The connection attempt was made with a precedence level lower than that specified by the remote service.

15.3 Procedures

```
ArpaSMTP.Open: PROCEDURE [remoteHost, localHostName: LONG STRING]
    RETURNS [ArpaSMTP.Handle];
```

Open opens an SMTP connection with the host specified in **remoteHost**. The field **localHostName** contains the name that the local host has advertised to the remote server. This name is the common name of the sending machine. The procedure returns a connection handle to be used in all subsequent SMTP operations.

The procedure can raise the signal **SMTPError**.

```
ArpaSMTP.Post: PROCEDURE [
    smtpHandle: ArpaSMTP.Handle, returnPath: LONG STRING,
    recipients: ArpaSMTP.Recipients, message: Stream.Handle]
    RETURNS [success: BOOLEAN,
        badRecipientList: LONG POINTER TO ArpaSMTP.InvalidRecipientList];
```

Post sends a message to the host specified by the **smtpHandle** field. The field **returnPath** contains the common address of the sender of the message; that is, <userName>@<localHostName>. The **recipients** field contains a sequence of users to whom the message is addressed and who are believed to reside on the host specified by the **smtpHandle** field.

Post returns a boolean specifying success or failure in posting the message to the specified recipients. If posting was not successful, then the field **badRecipientList** contains a pointer to a sequence of invalid recipients. Free this field by using the procedure **FreeInvalidRecipients**.

```
ArpaSMTP.Verify: PROCEDURE [smtpHandle: ArpaSMTP.Handle,
    user, fullyQualUserName, mailBox: LONG STRING];
```

Verify confirms that the string **user** identifies a known user on the host specified by the field **smtpHandle**. If the argument **user** is a user on the remote host, then the full name of the user (if known) and the fully specified mailbox are returned.

```
ArpaSMTP.Expand: PROCEDURE [smtpHandle: ArpaSMTP.Handle,
    distributionList, expandedList: LONG STRING];
```

Expand asks the host specified by the field **smtpHandle** to confirm that the argument **distributionList** identifies a mailing list, and if so, to return the membership of that list. The full name of the users (if known) and the fully specified mailboxes are returned.

```
ArpaSMTP.Reset: PROCEDURE [smtpHandle: ArpaSMTP.Handle];
```

Reset sets the connection to the state it was in when the connection was first established. All current state information (recipients, send, etc.) will be reset.

```
ArpaSMTP.Close: PROCEDURE [smtpHandle: ArpaSMTP.Handle];
```

Close ends the existing SMTP connection identified by **smtpHandle**. The connection identified by **smtpHandle** is invalid after successful completion of this operation and should not be used for subsequent operations.

```
ArpaSMTP.FreeInvalidRecipients: PROCEDURE [smtpHandle: ArpaSMTP.Handle,
    invalidRecipients: LONG POINTER TO ArpaSMTP.InvalidRecipientList];
```

FreeInvalidRecipients frees invalid recipients returned by the procedure **Post**. The field **smtpHandle** is a handle to the SMTP connection; the field **invalidRecipients** is a pointer to the sequence returned by **Post**.

16 ArpaSMTPServer

Simple Mail Transfer Protocol (SMTP) is a mail transfer protocol to be used with the TCP/IP protocols. The **ArpaSMTPServer** interface provides Pilot clients with an interface to the SMTP defined by RFC821.

References: *RFC821 Simple Mail Transfer Protocol, Postel, August, 1982.*

16.1 Types and constants

```
ArpaSMTPServer.RcptList: TYPE = LONG POINTER TO ArpaSMTPServer.RcptRecord;

ArpaSMTPServer.RcptRecord: TYPE = RECORD [
    recipientName: LONG STRING ← NIL,
    nextRecipient: ArpaSMTPServer.RcptList ← NIL];

ArpaSMTPServer.ReturnReason: TYPE = {ok, insufficientSpaceOnVolume,
    storageAllocExceeded, transactionFailed, otherError};
```

Return reasons are described below.

ok The default return reason.

insufficientSpaceOnVolume
The local medium does not have enough space to accomodate the message.

storageAllocExceeded
The space allocated for this transaction has been exceeded.

transactionFailed
The transaction failed for some reason.

otherError
Catch-all error.

```
ArpaSMTPServer.PostProc: TYPE = PROCEDURE [
    message: Stream.Handle,
    recipientNames: ArpaSMTPServer.RcptList,
    returnPath: LONG STRING,
    messageLength: LONG CARDINAL,
    noOfRcptsHint: CARDINAL]
    RETURNS [returnReason: ArpaSMTPServer.ReturnReason];
```

PostProc is used by the SMTP server when a message is received for the users specified by the list **recipientNames**. The message can be received by reading from the stream provided in the field **message** until the signal **Stream.EndOfStream** is raised. The **returnPath** contains the return mail path to the sender of the message.

The fields **messageLength** and **noOfRcptsHint** can provide hints to the mail client as to how to grow data structures to accommodate the message. The mail client should return problems in the handling of the message in the **returnReason** field.

```
ArpaSMTPServer.ExpandProc: TYPE = PROCEDURE [
    dl: LONG STRING,
    dataProc: PROCEDURE [user, mBox: LONG STRING]];
```

ExpandProc is used by the SMTP server when a request for distribution list expansion is made on the server. The client process returns the distribution list contents identified in the field **dl** by calling the **dataProc** with each user's name in the field **user** and mailbox information in the field **mBox**.

```
ArpaSMTPServer.VerifyProc: TYPE = PROCEDURE [user: LONG STRING]
    RETURNS [fullyQualifiedUser, mailBox: LONG STRING];
```

VerifyProc is used by the SMTP server when a request for user name verification is made on the server. The client process returns the user's fully qualified name (if known) in the field **fullyQualifiedUser** and the user's mailbox identifier in the field **mailBox**.

```
ArpaSMTPServer.ValidateProc: TYPE = PROCEDURE [user: LONG STRING]
    RETURNS [accept: BOOLEAN];
```

ValidateProc is used by the SMTP server when a request to deposit mail for a particular user is made on the server. If the client process is receiving mail for the indicated **user**, then it returns **TRUE** in the field **accept**.

```
ArpaSMTPServer.SMTPProcList: TYPE = RECORD [
    post: ArpaSMTPServer.PostProc,
    expand: ArpaSMTPServer.ExpandProc,
    verify: ArpaSMTPServer.VerifyProc,
    validateUser: ArpaSMTPServer.ValidateProc];
```

SMTPProcList is used to pass a list of procedures to the SMTP server; the SMTP server uses these procedures to communicate with client processes.

16.2 Procedures

```
ArpaSMTPServer.Register: PROCEDURE [
    smtpProcs: ArpaSMTPServer.SMTPProcList, serverName: LONG STRING];
```

Register starts an SMTP server session. Only one session is started no matter how many calls are made to **Register**. The fields **smtpProcs** contains a list of procedures that the SMTP server uses to communicate with the client process. The **serveName** field contains the commonly known name of the server.

```
ArpaSMTPServer.UnRegister: PROCEDURE;
```

UnRegister stops the SMTP server from receiving additional connections and releases all resources used by the SMTP server.

17 ArpaMailParse

ArpaMailParse parses the headers of messages formatted according to RFC822. Syntactic entities from RFC822, such as *atom*, are indicated by italics in this section.

To parse a message, call **Initialize** loop calling **GetFieldName**, call either **GetFieldList** or **NameList** (depending on the semantics of the field name returned by **GetFieldName**), and call **Finalize**. **NameList** is the main procedure to deal with lists of recipients in the many syntactic forms defined by RFC822. Most of the remaining procedures in the interface support special cases of these forms and are used infrequently.

ArpaMailParse is implemented by the program `ArpaMailParserImpl.bcd`.

References: *RFC822 Standard for the Format of ARPA - Internet Text Messages, Crocker, August, 1982*

17.1 Types

```
ArpaMailParse.BracketType: TYPE = RECORD [
    group: BOOLEAN ← FALSE,
    routeAddr: BOOLEAN ← FALSE];
```

BracketType is passed to a **ProcessProc** as part of its **NameInfo** argument and describes the context of a name in a name list.

group is **TRUE** if the name appears in the context "*phrase*: ...;"; that is, *phrase* is the name of a *group*. This *phrase* is not treated as part of any recipient name.

routeAddr is **TRUE** if the name appears in the context "*phrase* < ... >"; that is, *phrase* is the initial part of a *route-addr* describing a recipient.

```
ArpaMailParse.Handle: TYPE = LONG POINTER TO ArpaMailParse.Object
ArpaMailParse.Object: TYPE;
```

A **Handle** is a pointer to an **Object**, representing an instance of a parse.

```
ArpaMailParse.NameInfo: TYPE = RECORD [
    nesting: ArpaMailParse.BracketType,
    type: ArpaMailParse.NameType];
```

NameInfo is used exclusively with the **NameList** procedure; it provides the client-supplied **process** procedure with information about its parameters. **nesting** describes the context of this name in the name list being parsed. If **nesting.group** or **nesting.routeAddr** is **TRUE**, then procedures **GetGroupPhrase** or **GetRouteAddrPhrase** may be called from the **process** procedure to obtain the *phrase* for that nesting property.

```
ArpaMailParse.NameType: TYPE = {normal, file};
```

NameType is passed to a **ProcessProc** as part of its **NameInfo** argument; it describes how the **local** name is interpreted. **normal** means the name is a single recipient (neither a file name nor a public distribution list). **file** means the name occurs as the tag portion of an empty group list and should be treated as the name of a file containing a list of recipient names.

```
ArpaMailParse.ProcessProc: PROCEDURE [
    h: ArpaMailParse.Handle,
    local, registry, domain: LONG STRING,
    info: ArpaMailParse.NameInfo];
```

For each recipient encountered, **NameList** calls the client's **ProcessProc**, passing it the simple name, registry, and Arpanet host.

h is provided so the client may call **GetGroupPhrase** or **GetRouteAddrPhrase**.

local is always non-empty. The string parameters are free from leading, trailing, and excess internal white space. If **domain** is absent, then a string of length zero (not **NIL**) is passed. Each is guaranteed to contain room for **ArpaMailParse.minLength** characters. **domain** (but not **local**) may be changed in limited ways by a **ProcessProc**. It is permissible either to change the length to zero or (if the length is zero) to append a value to alter the qualification of the name that is to be passed to the write argument of **NameList**. **registry** is not used.

info provides additional information about the name being supplied. See the description of **NameInfo** for above.

```
ArpaMailParse.WriteProc: PROCEDURE [string: LONG STRING];
```

Each time the client's **ProcessProc** returns **TRUE**, **NameList** outputs the complete name (with possibly altered qualification) by calling **WriteProc** with fragments of the recipient name. **NameList** keeps the original format of the name as much as possible, including bracketing, comments, and the location of white space. Successive white space characters (outside of quoted strings) are replaced by a single space. **NameList** assumes responsibility for outputting appropriate separators (commas) and brackets, based on the values returned by successive invocations of **process**.

17.2 Constants and data objects

```
ArpaMailParse.endOfInput: CARDINAL = ...;
```

endOfInput should be returned by the client's **next** procedure (see **Initialize**) when the end of the input is reached.

```
ArpaMailParse.endOfList: CARDINAL = ...;
```

endOfList may be used as a delimiter terminating a list of names. It has no other effect.

```
ArpaMailParse.minLength: CARDINAL = 40;
```

The registry and domain **STRING**s passed to the client's **ProcessProc** will be at least this long.

17.3 Signals and errors

```
ArpaMailParse.Error: ERROR [code: ArpaMailParse.ErrorCode, position: CARDINAL];
```

Error is raised when the parse of the mail message fails. **code** describes the reason for the failure. **position** is the number of characters parsed when the error was detected.

```
ArpaMailParse.ErrorCode: TYPE = {
    illegalCharacter, unclosedBracket, bracketNesting, implementationBug,
```

phraseExpected, domainExpected, atomExpected, commaOrColon Expected,
at Expected, spaceInLocalName, mailBox Expected, missingSemiColon, nestedGroup,
endOfInput, commaExpected, fieldsAreAtoms, colonExpected, lessThanExpected,
greaterThanExpected, noFromField};

The error conditions that cause a failure are largely self-explanatory. **noFromField** is not raised by **ArpaMailParse**, but is provided for clients who cannot succeed if the message is either unparseable or contains no "From:" field.

17.4 Procedures

ArpaMailParse.Finalize: PROCEDURE [h: ArpaMailParse.Handle];

Finalize finalizes the parse. This procedure must be called when the client has finished parsing, after either normal completion or an error has occurred. **Finalize** modifies **h**, so it should not be reused.

Note: **Finalize** may not be called from within the **process** procedure invoked by **NameList** or from within the catch phrase of **Error**.

ArpaMailParse.GetFieldBody: PROCEDURE [
h: ArpaMailParse.Handle, string: LONG STRING, suppressWhiteSpace: BOOLEAN ← FALSE];

GetFieldBody reads the remainder of the current field body using **next** (see **Initialize**) and puts the characters consumed into **string**. If the field body is too long, then overflow characters are discarded. If the field body terminates before a **CR** is seen, then **Error[endOfInput]** is raised. Upon return, **string** has no initial or terminal white space (blanks and tabs) and, if **suppressWhiteSpace** is **TRUE**, each internal run of white space is replaced by a single blank. RFC822 line-folding conventions are also observed.

ArpaMailParse.GetFieldName: PROCEDURE [h: ArpaMailParse.Handle, field: LONG STRING]
RETURNS [found: BOOLEAN];

GetFieldName presumes that **next** (see **Initialize**) is positioned to read the first character of a field name and it returns the field name, without the terminating colon, in **field**. It leaves **next** ready to return the first character following the colon (or, if the end of the message header has been reached, the character (if any) after the two **CR**s that normally terminate the header). If the field name is too long, overflow characters are discarded. Upon return, **found** is **FALSE** if no field names remain in the header.

If the header field ends prematurely or illegal header characters are encountered, then **Error[fieldsAreAtoms]** is raised. **Error[colonExpected]** is raised if there are embedded spaces in the field name.

ArpaMailParse.GetGroupPhrase: PROCEDURE [h: ArpaMailParse.Handle, phrase: LONG STRING];

GetGroupPhrase can only reasonably be called from inside the **process** procedure passed to **NameList**. The *phrase* that introduces the current group is appended to **phrase**. If the phrase is too long, overflow characters are discarded. Upon return, **phrase** has no initial or terminal white space (blanks and tabs, and each internal run of white space is replaced by a single blank. If **GetGroupPhrase** is called at an inappropriate time (for example, when **NameInfo.nesting.group** = **FALSE**), no changes are made to **phrase**.

```
ArpaMailParse.GetRouteAddrPhrase: PROCEDURE [h: ArpaMailParse.Handle, name: LONG STRING];
```

GetRouteAddrPhrase can only reasonably be called from inside the **process** procedure passed to **NameList**. The *phrase* that describes the current recipient is appended to **name**. If the phrase is too long, then overflow characters are discarded. Upon return, **name** has no initial or terminal white space (blanks and tabs), and each internal run of white space has been replaced by a single blank. If **GetRouteAddrPhrase** is called at an inappropriate time (for example, when **NameInfo.nesting.routeAddr** = FALSE), then no changes will be made to **name**.

```
ArpaMailParse.Initialize: PROCEDURE [next: PROCEDURE RETURNS [CHARACTER]]
  RETURNS [ArpaMailParse.Handle];
```

Initialize creates an instance of the header parser and returns a **Handle** to be passed to other procedures of this interface. Subsequent invocations of **GetFieldName**, **GetFieldBody**, and **NameList** obtain their input using **next**.

```
ArpaMailParse.NameList: PROCEDURE [h: ArpaMailParse.Handle,
process: ArpaMailParse.ProcessProc, write: ArpaMailParse.WriteProc ← NIL];
```

NameList expects to read characters using **next** (see **Initialize**) for a structured field body consisting of a list of recipient names. For each name encountered, it calls **process**. If **process** returns TRUE and **write** is not NIL, then **NameList** outputs the complete name, with potentially altered qualification, by calling **write**. If syntax errors are detected during parsing, then **Error** is raised. It is legitimate for the **process** procedure to raise a signal that causes **NameList** to be unwound.

```
ArpaMailParse.StringForErrorCode: PROCEDURE [code: ArpaMailParse.ErrorCode, s: LONG STRING];
```

StringForErrorCode appends a user-sensible **error** message onto the string **s**. If the error message is too long, then overflow characters are discarded.

18 ArpaVersion

ArpaVersion allows for the acquiring of version number information.

The procedure is defined below.

```
ArpaVersion.Append: PROC [s: LONG STRING];
```

Append appends the version number of the software onto the string **s**.

Glossary

Glossary

abstract machine
A set of functions provided by hardware or software that forms the underpinnings of a system sitting above. For example, Pilot is an abstract machine that runs on a variety of physical machines.

address fault
A system error that occurs when an attempt is made to reference an illegal address.

adjective
An identifier constant from an enumerated type, used to select one of the alternatives in a variant record. SEE *tag*.

alignment
An I/O interface constant that constrains alignment of data in memory; that is, each I/O buffer must be aligned on a virtual memory address that is a multiple of alignment. SEE ALSO *granularity, truncation.*

asynchronous
Returning control to the caller before completing an operation; e.g., after queuing the operation. SEE *synchronous*.

atom A Mesa primitive that provides a unique identifier in a global naming space. An atom has an associated property list.

backing storage
A sequence of pages from a file or real memory to which a part of virtual memory is mapped. Any part of virtual memory currently in use is mapped to a backing store. SEE *Map*.

backstop
A system for recording information about malfunctioning software and hardware. For product systems, installed instead of a debugger.

bank A block of 256 pages aligned on a 64K word boundary (Mesa terminology).

bcd Binary Configuration Description.

Binary Configuration Description
A compiled and possibly bound Mesa module, sometimes called an object file. SEE *configuration.*

bind To combine object modules into a larger unit (called a configuration) by resolving intermodule references.

BITBLT
A complex Mesa processor instruction to do a very fast bit block transfer. Originally designed to move and possible modify a rectangular piece of the display bitmap, but has found other uses. (Pronounced "bit blit.")

bitmap A representation of a rectangular image as a sequence of bits, each of which represents the intensity of a point in the image. The display hardware and microcode convert a bit-map to a displayed image.

block 1. The description of a section of memory which is the source or sink of transmitted bytes. This memory section is a sequence of bytes (not necessarily word aligned) that must lie entirely within a mapped space.
2. A Mesa construct that is used as a placeholder to represent a region of storage of indeterminant size.

boot To load and start a system on a machine whose main memory has essentially undefined contents. ("Boot" is short for "bootstrap," which in turn is short for "bootstrap load.")

boot file
1. A file that contains a ready-to-run Pilot-based system that can be loaded by a germ for execution.
2. A file that contains all of the system routines and tools that are automatically provided on power-up, as well as bootstrap code. The type of boot file determines the environment of the machine; for example, Tajo.

boot switch
A Runtime software-selectable option that transmits operational information from the booting agent to the running boot file.

build 1. To combine separate modules into a single unit by assembling them into object modules, binding (linking) to produce a .lnk module, locating (assigning an absolute location in main memory) to produce a .loc module, and finally, running MakeMicroBoot to produce a .db file. SEE *file extension* for module definitions.
2. To generate a bootable file from individual components on a working directory. Picks up named components from hierarchical DFs; last step is to run MakeMicroboot or MakeBoot to create a boot file.

byte code
A one-byte opcode instruction compiled by Mesa into a directly executable, stack-oriented language. Interpreted by special microcode on each of the various machines.

call stack
A Mesa processor data structure containing a frame for each procedure invocation that has not yet returned. The call stack is ordered with the most recent invocation first.

cardinal
A non-negative integer.

catch phrase
A Mesa construct that establishes code to catch one or more signals.

channel
A Mesa interface for accessing and driving I/O devices. Specifies the device-specific data and control information needed by a client to operate the device. SYNONYM: *software channel.*

client A program that uses the services of another program or system.

code 1. The statements in a software program.
2. To write a list of instructions to cause the product/system to perform specified operations.

Common Software
A collection of modules, written in Mesa and based on Pilot, that provide frequently used functions. Not included in PilotKernel.bcd.

Communication Package
The code allowing Pilot clients to perform inter- and intra-processor communication.

compile To translate a source file into object file (.bcd).

condition variable
A data structure that allows a process to wait for some event either until notified by another process that the event has occurred or until a specified period of time passes without occurrence of the event.

configuration
A set of modules and/or sub-configurations assembled into a new conglomerate entity which has the characteristics of a single module.

configuration description
A C/Mesa source file that tells the Binder how to combine modules into a configuration. Called *config.*

consistent
Having the characteristic that in a set of object modules referenced directly or indirectly there is no case of more than one version of a particular object module.

CONTINUE
To resume program execution at the statement following the one to which the catch phrase belongs. Thus, control is resumed in the procedure where a signal was caught, not in the procedure that raised the signal.

Courier
The Xerox Network Systems protocol that permits the initiation and control of remote processes, including the transfer of information and control parameters associated with such processes. SEE *remote procedure calling.*

dangling pointer
A pointer to an invalid memory location, usually the result of deallocating storage while a pointer to it remains. For example, a pointer allocated from a local frame whose procedure has returned.

data stream
A device- and format-independent interface used to move sequential data between a device and a process or between two processes. Two different computers may be involved.

definitions module
1. In Mesa, an interface unit that serves as a blueprint or specification for how the parts of a system will fit together. Does not contain executable code. During compilation the modules provide definitions which can be referenced by

other modules being compiled.
2. Describes an interface to a function by providing a bundle of procedure and data declarations which can be referenced by client programs. SEE *module*.

Description File (DF)
A file that identifies the files needed to build a system component. A fundamental Pilot concept, it is a text file listing all components that make up a module, where they are located, and when they were created. Almost a blueprint for building a module.

detach To condition a process so that when it returns from its root procedure, it will immediately be deleted.

device A peripheral unit, almost always hardware, that is separately accessible through its own channel.

device driver
1. A set of programs that implement and export a software channel. Includes interrupt routines, interfaces with microprograms, control of hardware registers, and other functions required to service the device.
2. A program that translates the channel requests for a device into physical device actions.

df (also **DF**)
Description File.

drive An I/O device capable of containing a Pilot physical volume, typically a fixed disk or a removable disk pack. SYNONYM: *disk*.

event A value that names a particular occurrence which may interest some subsystems.

exception
An unusual event which programs must be prepared to handle, such as I/O error.

export To enable all or part of an interface to be used by other modules. SEE *import, interface*.

face A Mesa interface that embodies some aspects of the processor and of its I/O devices. Implemented by a combination of Mesa code (head), lower-level machine code (microcode), and the underlying hardware.

fault A process failure that suspends running execution of the process (possibly only temporarily). SEE *page fault, frame fault, write fault*.

file 1. The basic unit of long-term information storage. Consists of a sequence of pages, the contents of which can be preserved across system restarts.
2. A sequence of data pages located on some physical device and containing some common grouping of information. Files may be local or remote.

file extension
A part of a file name, separated from it by a period (.) that indicates the file type. In Pilot, some file extensions are:

.boot – boot file
.db – microcode boot file
.df – description file
.com – command file that runs on the VMS operating system on a VAX.
.bat – command (batch) file that runs on an IBM-PC running PC-DOS.
.tds – tool driver script for use with the XDE Tool Driver program.
.hex – PROM-image form of executable assembly codes, as made from *loc* file.
.loc – located version of assembly code, as made from *lnk* file.

.lnk – linked version of assembly code, as made from other *lnk* files and *obj* files.
.obj – object (assembled) version of assembly code, as made from *asm* file.
.asm – assembly (e.g., 80186) source code
.mp1 – output from the linker--sizes, unresolveds, included files, etc.
.mp2 – output from the locater--absolute addresses (gigantic !)

file id A unique identifier issued to a file by Pilot for use by higher level software.

fileType
A functional designation that enables Pilot, scavenging programs, and clients to recognize a file's purpose. Assigned by programmer according to definitions in the **fileTypes** interface file.

file window
A consecutive (but not necessarily contiguous) group of pages within a file into which a map unit is mapped.

filter A mechanism for modifying a data stream; that is, transforming, buffering, or otherwise manipulating data before passing it on to another stream.

fork 1. To create a new process that will run concurrently with the process that created it. The process that is forked is called a live process.
2. To create a copy of a system followed by development of that copy in a fashion inconsistent with the continuing development of the original.

frame A PrincOps data structure allocated for the variables and internal data structures of an executing module or procedure. Module frames are called global frames; procedure frames are called local frames. Since Mesa supports multiple concurrent execution of a procedure, a given procedure may have several frames.

frame fault
A fault that occurs when a process is unable to allocate a local frame from the frame heap, typically during a procedure call.

frame pack
A swap unit produced by the Packager that contains the global frames for a collection of modules.

frame segment header
A machine-dependent record located in resident MDS memory that keeps track of sets of actual local frames.

friends interface
An interface (possibly undocumented) that can be used with care by clients outside the group owning the components.

gateway
1. A hardware/software system capable of exchanging messages between two dissimilar networks.
2. A processor serving as a forwarding link between Ethernets. SEE *Router*.

germ A bootstrap loader that can load a Pilot bootfile into a Mesa processor and place it into execution. Each kind of processor has one or more germs.

granularity
An I/O interface constant that constrains alignment of data in memory; i.e., each I/O buffer in virtual memory must have a length that is an integral multiple of granularity. SEE ALSO *alignment, truncation.*

handle
An identifier associating the use of a hardware or software facility with the process that obtains the handle.

handler
Device-dependent microcode associated with a given I/O device on the IOP board. Each type of I/O device has its own handler.

head 1. One or more files containing the object code of the modules that export the device faces.
2. An implementation of a face for a processor or device. A collection of heads provides a processor-independent environment in which Pilot and its clients execute.

heap A system-designated area of virtual memory used for dynamic allocation of storage. SEE *zone*.

implementation module
SYNONYM: program module.

initialize
To set to a known value.

initial microcode
1. Lowest level microcode, typically residing in a special place on the disk, outside any logical volume. Invoked by hardware booting logic of the machine.
2. The program that reads the Pilot microcode and the germ from the disk.

interface
A formal contract between pieces of a system that describes the services to be provided. A provider of these services is said to implement the interface; a consumer of them is called a client of the interface.

interface module
A module that defines types, variables, constants, procedures, and signals, thus specifying the services to be provided by its implementation modules. SYNONYM: Definitions module.

internet
A collection of networks mutually accessible via internet routing services.

Internet Datagram Protocol
A connectionless protocol which provides for the addressing, routing, and delivery of standard internet packets.

Internet Transport Protocols
The set of protocols which provide for the transport of data across an interconnection of networks.

interval
A sequence of pages in the virtual address space, described by a pointer to the first page and a count of the number of pages.

jam To deactivate or remove from a queue.

join 1. To return results to the process doing the join.
2. To delete the process that was spawned by the parent process' fork.

kernel For Pilot, a file containing the object code of the fundamental parts of the Pilot operating system.

link An absolute disk address.

local boot file
A boot file that is specific to the machine and circumstances in which it is created. SEE *universal bootfile*.

log file A file containing a history of program actions.

logical volume
The unit of storage for client files and the system data structures for manipulating them. Contained in a physical volume.

machine
A hardware configuration consisting of a processor, main memory, and peripheral devices; for example, workstations and servers.

Main Data Space
A subspace of virtual memory that provides the local execution environment for Mesa programs and holds the implicit Mesa data structures.

maintenance release
A small release that fixes problems in the previous major release. Only problems too serious to wait for the next major release are addressed.

MakeBoot
A Mesa program that transforms an object file containing Pilot and its client into a memory image which can be run on any machine conforming to the Mesa Processor Principles of Operation. The resulting bootfile is later boot-loaded for operation.

MakeMicroBoot
A program that transforms an object file containing emulator and IOP code into a memory image which can be run on any machine conforming to the Mesa Processor Principles of Operation. The resulting bootfile is later boot-loaded for operation.

map To associate a region of virtual memory with a file window so that the contents of the file window appear to be the contents of the region.

mapping
The act of associating an area of virtual memory with a sequence of file pages in main memory (window).

map unit
A consecutive group of virtual memory pages that is the principal unit for allocating, mapping, and swapping virtual memory.

MDS Main Data Space.

Mesa A Pascal-like, strongly typed, system programming language that forms the basis of the Xerox Development Environment. Pilot is written in Mesa.

Mesa.db
A boot file that contains 8010-specific code, including initialization code, the Mesa emulator, and IOP handlers not in ROM.

MesaDaybreak.db
A boot file that contains 6085-specific code, including initialization code, the Mesa emulator, and IOP handlers not in ROM.

Mesa Processor Principles of Operation (PrincOps)
A document that describes the Mesa abstract machine. A variety of machines implement the PrincOps architecture in hardware and microcode; Pilot runs on PrincOps machine.

Mesa run-time
A set of procedures used as a base upon which to build experimental systems.

microcode
The code that, together with the heads, implements the Mesa processor on a given kind of hardware.

mode A special state of a system in which user actions have special meaning.

mode-less
Free of modes, so that a given action produces the same result. For example, in a mode-less user interface, pressing a particular key always has essentially the same effect.

module In Mesa, the smallest self-contained executable program unit; the basic unit of compilation. The two types are: definitions or interface modules and program modules.

monitor A Mesa module that controls access to shared resources, thus synchronizing interactions among processes.

monitor lock
A data structure that contains the interlocks sufficient to guarantee that only one process at a time may gain access to shared resources.

network
A communication medium, such as an Ethernet, known to routers by a unique identifying number.

network address
1. The source or destination of processes which reside on different machines. Unique between system restarts, but reused each time the system is restarted.
2. An identifier that consists of a network number, host number, and socket number. The network number identifies a network anywhere in the world. The host number uniquely identifies a machinwithin the designated network. A socket number identifies a particular socket on that host.

network stream
A byte stream that provides sequenced, duplicate-suppressed, error-free, flow-controlled communication over arbitrarily interconnected communication networks, as defined in Xerox Internet Transport Protocols.

node A block of allocated storage, often with a record structure. SYNONYM: *storage node.*

Othello A Mesa program used to manage Pilot physical and logical disk volumes. Does not provide programming facilities.

outload file
A snapshot of the volatile state of a system (essentially the contents of real memory and registers). Outload files are used by the debugger. SEE *World-swap.*

overlay The technique of repeatedly using the same area of internal memory for bringing routines into memory from bulk storage. Used when the available main memory is smaller than the total storage requirements necessary for all program instructions.

package
To group components of modules into swap units to try to improve use of real memory.

packet A unit of information in the internet.

page A block of 256 words (512 bytes) of information in either virtual memory or a file. The page is the basic addressable unit of a file.

page fault
A fault that occurs when a process attempts to reference a page of virtual memory which is not (at that moment) backed by a page of real memory.

physical volume
The basic unit of physical availability for random access file storage; for example, rigid disks. A storage medium whose availability is intrinsically independent of that of other instances of such media. SEE *logical volume*.

Pilot The operating system that manages the hardware resources of, and provides the run-time support for, all Mesa programs on a machine. A nucleus of software which interfaces between a Mesa processor and all other software.

pipeline
A sequence of concatenated filters that perform a series of transformations on the contents and properties of a stream.

pointer 1. A data item containing the location of a value.
2. The exact location of desired information.

private interface
An interface available only to clients of closely related components, whose implementors are typically in the same group. SEE *public interface*.

process The fundamental architectural concept in all Mesa software. A procedure activation that runs concurrently with its caller, allowing asynchronous activities.

program module
1. A Mesa unit that contains actual data and executable code that implements the interfaces. Program modules can be loaded and bound together to form complete systems.
2. A binary object file that contains the procedural description of one or more of the functions defined in some definitions module. Also called implementation module. SEE *module*.

public interface
An interface that can be used by all clients. SEE *private interface*.

pulse A processor-dependent unit of time, the resolution and accuracy of which is determined by the internal clocks of the processor. Typically, resolution is in the range of 1-1000 microseconds, and accuracy is 10% or better.

real memory
The physical memory that holds software and data during processing, as opposed to secondary or virtual memory.

release
An official, consistent version of software produced and maintained by its developers. The term used to identify and categorize a software product provided by Engineering to any other functional group; releases are separated into internal releases and external releases. SEE ALSO *maintenance release*.

remote procedure call
The invoking of a procedure from one machine to be executed in another machine over a network.. SEE *Courier*.

router 1. A software package that manages the transmission and reception of information from one Pilot client to another Pilot client on the same or another system element.

2. A software package that sends packets between sockets. If the destination socket is on another network, then the path chosen by a router includes intermediate stops. A router that sends packets between networks is called an *internet router*.

scavenge
To check for damaged file structures and to attempt to repair them.

server 1. A combination of hardware and software capable of performing some particular set of services. Includes appropriate peripheral devices; for example, a large disk storage device in a file server.
2. A machine dedicated to performing one or more services. SEE *user*.

signal A Mesa language construct used to help handle exceptional conditions encountered during program execution. Signals are like procedures except that the code to be executed is determined at run-time.

Signaller
The program that gets control when a signal is raised, attempts to find an associated catch phrase, and executes the code in the catch phrase.

snapshot
1. A record in a separate file that contains the contents of another file or device at a given moment.
2. To make such a record.

socket 1. A source or destination of packets on a given machine. Uniquely identified by a 16-bit socket number. Accessed through a channel interface and thus is a logical input/output device. Several streams of packets may share a single socket.
2. An abstract location in a host that can originate or receive communication.
3. A source or destination of packets on a given machine.

Space The Pilot interface for managing virtual memory. Space often refers more generally to virtual memory.

stateless enumerator
A type of function that enumerates items when successively called. The input parameter for each call is the returned value of the previous call.

state vector
A statically allocated (during MakeBoot) location for storage of the state of a running process that has been suspended by an interrupt or fault. Cleared or released when the interrupted process is restarted.

stream 1. A sequence of bytes, possibly marked by attention flags and possibly partitioned into identifiable subsequences.
2. An abstraction for device- and format-independent sequential access to a collection of data. Some streams also provide random access to the data.

stream component manager
The software entity that implements a stream components; that is, a transducer, filter, or pipeline.

Stream Facility
An interface that provides Pilot clients with a convenient, efficient, device- and format-independent sequential access to a stream of data.

stream handle
A pointer to a stream object that indentifies the particular stream being accessed. Contains the data and procedures of operations on the stream.

stream interface
The set of procedures and data types by which a client controls the transmission of a stream of information.

Stream Package
Software that provides a basic set of transducers and filters and a way of assembling them sequentially into processing and transmitting pipelines. A mechanism for transporting sequential data between Pilot client and I/O device or other Pilot client that is device- and format-independent.

Supervisor
An interface that manages the orderly acquisition and release of shared resources; for example, files, removable volumes, or even the entire processor.

swap To transfer data between memory and files, either in response to hints from the client program or upon demand. To swap in is to copy from a file window into real memory; to swap out is to copy from real memory to a file window.

swap unit
1. A subdivision of an interval that allows more efficient management of swapping.
2. A portion of a space to be swapped.

switch A modifier to a command or subcommand, often preceded by /.

synchronous
Completing an operation before returning control to the caller. Most Pilot operations are synchronous. SEE *asynchronous*.

Synchronous Point-to-Point Protocol
The Xerox bit-synchronous protocol for point-to-point data links.

system An organized combination of hardware components and software working together to perform some logical process in a systematic manner.

system architecture
The design/configurator for attaching the various hardware/components of the system to interact with each other for the purposes for which the system was designed.

system volume
The logical volume on which the bootfile resides.

Tajo A collection of functions designed to facilitate the implementation and execution of software development tools. (internal Xerox)

timeout
To fail to complete an operation within a specified amount of time.

transducer
1. A mechanism for transcribing a sequential stream of data on or off an I/O device.
2. A software entity that implements a stream, such as MStream, connected to a specific device or medium through a Pilot channel.

truncation
An I/O interface constant that constrains alignment of data in memory; that is, each physical record on the device must have a length that is a multiple of truncation. SEE ALSO *alignment, granularity*.

uncaught signal
A signal that is not handled by any module in the call stack. If a signal is uncaught, then the Signaller transfers control to the debugger.

universal boot file
A boot file that can be transported to any machine (of the right configuration) and executed there. COMPARE TO *local bootfile.*

universal identifier
A 5-word Mesa object that uniquely names network resources and other permanent objects in a network.

user In the context of the user/server model, the active entity which uses a service provided by a server. SEE *server.*

UtilityPilot
A version of Pilot that provides most Pilot facilities but does not require that a disk be present on the machine. Used primarily to build special utility systems; for example, disk initializers, diagnostics.

valid memory location
A currently allocated memory location. A location that has been freed is invalid and should not be referenced.

version A complete and internally consistent set of modules that implement a system. A fully debugged version with needed features becomes a release. SEE *release.*

virtual memory
A large, linearly addressed, word-organized area common to all processes and devices. All software, including Pilot, common software, and applications, resides and executes here.

volume SEE *physical volume, logical volume.*

window A data structure representing a sequence of pages from a file.

working set
Memory requirement for a Pilot function; that is, the virtual pages (code and data) that, when in memory, provide a local minimum of page faults to service the function.

world-swap
To write out the complete state of a logical volume onto a disk file and then to read in a different state. CoPilot normally works by world-swaps between the debugger and the program being debugged. SEE *outload file.*

write fault
A fault that occurs when a process attempts to store into a page of virtual memory which is backed by a read-only page of real memory.

Xerox Development Environment (XDE)
1. A set of robust and sophisticated software development tools, which use the resources of Xerox Network Systems.
2. A set of basic tools for manipulating programs, including the Tajo user interface and a variety of built-in tools, but not including language-dependent tools such as the compiler and debugger.

Xerox Network Systems (XNS)
A combination of hardware and software that unites specialized devices into a network where the capabilities of a variety of workstations are enhanced by distributed services.

zone A client-designated area of virtual memory used to allocate and free arbitrary-sized storage nodes. SEE *heap.*

Index

Index

A

B

C

D

E

F

G

H

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

CHANGE HISTORY

Update to Pilot Programmer's Manual of December 1986

The Pilot Programmer's Manual has been updated to reflect the state of the Pilot operating system as implemented in Pilot 14.0. Changes made since publication of Document #610E00160 are summarized below.

Chapter 1: Introduction

Page	Change
1-6	§1.3.2. Revised MDS description.
1-8	§1.5. Revised pseudo-Mesa declarations

Chapter 2: Environment

Page	Change
2-4 to 2-8	§2.1.2. Added more device type interfaces.
2-25 to 2-34	§2.3.5. Map logging changes: Client-side map logging; switch \375 has no effect. Added table of switch names, values, and meaning. Expanded description of switches.

Chapter 3: Streams

Page	Change
3-3	§3.2. Expanded **stream.Delete** description.
3-5	Last ¶. Revised end-of-stream discussion.
3-6	§3.2.1.1. Added Note.
3-19	§3.6. Moved "Memory stream" from Chapter 7.

Chapter 4: File Storage and Memory

Page	Change
4-18	§4.3.3. Added **FileTypeExtrasExtras**
4-47 to 4-49	§4.6.2. Added **HeapExtras**: **NewCreate**, **NewCreateUniform**, **NewCreateMDS**, **NewGetAttributes**, and **NewGetAttributesMDS**.

Chapter 5: I/O Devices

Page	Change
5-5	§5.2. Corrected **LevelV** description.
5-11, 5-15	§5.3. Added **UserTerminalExtras2**.
5-19	§5.5. Added **FloppyExtras, FloppyExtrasExtras, FloppyExtras3,** and **FloppyExtras4**.
5-23	§5.5.2. Added Note.
5-24	§5.5.3. Added Note.
5-26	§5.5.3. Added description of **FloppyExtras4** additions to **NewScavenge**.
5-38 - end	§5.8 and §5.9. New sections for FloppyTape.

Chapter 6: Communications

Page	Change
6-2	Socket number range is now 3000
6-4	§6.2. **XNS.bcd** replaces **Communication.bcd**. (§6.4. Same.)
6-24	§6.4.3. Added caveat about routing table.
6-25	§6.4.3. Replaced description of **SetNetworkID**.
6-28	§6.5.2. **FlowControl** is implemented and described.
6-31	§6.5.3.1 . Added status code descriptions; correc ted device error description.
6-43	§6.5.5.2. Added **dialupExtras**.
6-56	§6.6.4.1. Corrected error in **RemoteErrorSignalled** statement.
6-59	§6.6.6.1.2. Corrected error in Mesa data type.
6-66	Added new sections: Network Binding (also added example in Appendix G), XStream - Bulk data protocol, and PhoneNet driver

Chapter 7: Editing and Formatting

Page	Change
7-2	§7.2.2. Corrected **Format.StringProc**.
7-6, 7-7	§7.3.3. Added **NIL** description for most operations.
7-7	§7.3.3.1. Corrected **StringToNumber** discussion. Added **NIL** description where applicable.
7-9, 7-10	§7.3.3.2. Added **NIL** description where applicable.
7-10	§7.4.2. Redefined **year**; added **dst** description to **Time.Packed**.
7-11	§7.4.2. Expanded **Pack** description. §7.5. Moved "Memory streams" to Chapter 3.
7-12	§7.5 (new). Added new section, Sorting.

Chapter 8: System Generation and Initialization

Page	Change
	Global: Othello is not supported in Pilot 14.0.
8-2	§8.2. Changed `RuntimeLoader.bcd` to `Loader.bcd`.
8-3	§8.2. Deleted description of possible local boot file facility.
8-4	§8.3.1. Added to **FormatPilotDisk**.
8-5	§8.3.1. Expanded **SetUserAbort** description.
8-7	§8.3.3. Added other signals raised by **MakeBootable**.
8-13	§8.5.3. Added errors to **TemporaryBooting.MakeBootable**.

Chapter 9: The Backstop

Page	Change
9-2	§9.1.2. Added **Note:** how to calculate the size of the volume containing the backstop boot file.

Chapter 10: Online Diagnostics

Page	Change
10-1	§10.1. Added implementation bcds; added note about backward compatibility.
10.5	§10.1.1. Modified **EventReporter** description.
10-11	§10.1.3. Modified **modemSignal** description.

Appendices A - G

Page	Change
A-2	§A.1.1. Updated physical memory requirements of Pilot. §A.2.3 Revised communication performance estimate for system elements connected to the same Ethernet.
D-1	Added: Clients must set processor clock.
D-1	Added: debugging UtilityPilot-based clients.
G	New: Example of network binding.

Appendix H: TCP/IP (old Section 11)

Because the interfaces described in the original Section 11 are non-Pilot related, the section was moved to this appendix.

Page	Change
H-3	§1.2. **ArpaPackageMake** and **ArpaPackageDestroy** procedures added.
H-4	§2.1. Suspend reasons **securityMismatch** and **precedenceMismatch** added.
H-6, H-7	§2.1. **NotifyListenStartedProc** added. Called by **Listen** and **Make**.

Appendix H: TCP/IP (old Section 11) - continued

Page	Change
H-13	§5. **GetArpaInitInfo** procedure deleted.
H-20	§6.2. **StreamAborted** error added.
H-28, ff	§10.2 Several error reasons added.
H-30	§10.3 **Destroy** procedure added.
H-30, H-31	§10.3 **host** and **port** parameters added to **Store** and **Retrieve** procedures.
H-32	§10.3 **Passive** procedure added.
H-37	§12.1 **errorString** field added to **InvalidRecipientRecord**.
H-38	§12.2 Several error reasons added.
H-40	§12.3 **Reset** procedure added.
H-41	§13.1 **RcptList** and **RcptRecord** and return reasons added.
H-41	§13.1 **messageLength** and **noOfRcptsHint** fields added to **PostProc**.
H-48	§15. New section: **ArpaInit**.
H-49	§16. New section: **ArpaVersion**.