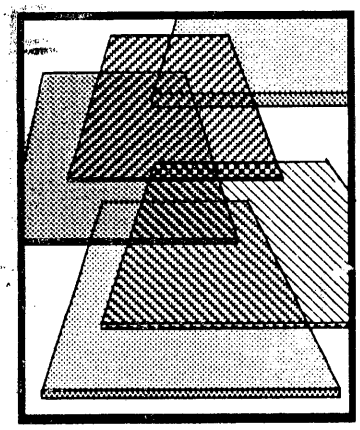
XEROX

Xerox Development Environment



Xerox Development Environment: Concepts and Principles

XDE3.0-1001
Version 3.0
November 1984

PRELIMINARY

Office Systems Division
Xerox Corporation
3450 Hillview Avenue
Palo Alto, California 94304

Notice

This manual is the current release of the Xerox Development Environment (XDE) and may be revised by Xerox without notice. No representations or warranties of any kind are made relative to this manual and use thereof, including implied warranties of merchantability and fitness for a particular purpose or that any utilization thereof will be free from the proprietary rights of a third party. Xerox does not assume any responsibility or liability for any errors or inaccuracies that may be contained in the manual or have any liabilities or obligations for any damages, including but not limited to special, indirect or consequential damages, arising out of or in connection with the use of this manual or products or programs developed from its use. No part of this manual, either in whole or part, may be reproduced or transmitted mechanically or electronically without the written permission of Xerox Corporation.

Preface

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

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

Xerox Corporation
Office Systems Division
XDE Technical Documentation, M/S 37-18
3450 Hillview Avenue
Palo Alto, California 94304

Table of contents

Introduction

The purpose of the Xerox Development Environment (XDE) is to facilitate program development. Specifically, its goals are to reduce the cost of programming (in terms of people time and processor time), and increase the quality of the code produced (in terms of efficiency, reliability, and ease of maintenance). The foundation of the XDE is therefore an integrated set of cooperating programs that aid in developing and maintaining large programs. This tool set is used in a *distributed* environment; that is, a group of powerful personal workstations loosely tied together by a local area network (Ethernet).

Because each machine has only one user, the XDE can rely on the principle of cooperation, rather than competition. Thus, there is no need for a centralized resource manager, and individual programs can directly share resources such as memory and compute power ("personal timesharing"). Similarly, distinct machines can cooperate via the Ethernet. Expensive resources such as electronic printers can be shared among many distributed users. Incorporating such resources into each workstation would be utterly impractical: by assuming that workstations are cooperative rather than antagonistic, it is possible to share such a resource among many users. Cooperation thus allows the full exploitation of the environment's resources and capabilities, since little effort is expended in centralized control or protection.

The XDE provides a standard set of *tools*, or applications programs, that support and simplify many common programming tasks. However, this tool kit is completely open-ended; you are not limited to the existing tools. Rather, the tools are built from a large library of extensively layered system routines and primitives. You have access to all the system routines, both simple and complex, and are free to use them to custom tailor the existing tools or to create your own specialized tools. Because you are the only user on your machine, you can modify the environment as much as you like; tools that you create have exactly the same status as the built-in ones. The software environment is thus layered, fully extensible, and extremely flexible.

Like the software architecture, the hardware architecture also emphasizes cooperation and tailorability. The XDE is *distributed* both geographically and in terms of control: all machines on a given network have equal control and equal access to shared resources. Individual workstations cooperate via the Ethernet in order to share a resource, provide one another with files, or enable a conversation between users. The facilities for

communication among workstations are also layered, allowing for different degrees of reliability and specialization.

The distributed architecture also means that the hardware, like the software, is easily extensible. It is easy to add or remove new workstations or resources, or to modify the topology of the network to fit the needs of a particular group of users. Because you can access any resource from any workstation, the distributed environment also increases reliability. For example, if one printer is busy or not functioning, there may be another working somewhere on the network.

The goal of the Xerox Development Environment, then, was to create a cooperative, extensible environment to support software development in a networked system. With this goal in mind, the designers of the XDE began the development of the Mesa programming language, the Pilot operating system, and the Mesa processor architecture. The three pieces were designed expressly to support the goals of the XDE (rather than the other way around).

The Mesa programming language is a high-level language with strict type-checking and strong support for modularity and concurrent processes. Pilot is a specialized operating system that provides a uniform basis for the development of applications software. The Mesa processor emphasizes efficient execution of high-level language constructs and extremely compact program representation.

The first part of this document discusses the advantages and implications of the tools environment, and some of the ways in which it facilitates cooperation among programs. The second part describes the communication network linking individual machines, and the additional services that it makes available. Part three discusses the support provided by the language, the operating system, and the processor architecture. There is a glossary following the third chapter which includes both terms from this document and other terms that you are likely to encounter during your first exposure to the Xerox Development Environment. A selected bibliography is also included.

1

Tajo

The Xerox Development Environment tools environment, called *Tajo*, is a set of integrated tools designed for creating, debugging, and maintaining large programs. The primary goal of the Tajo design is to provide maximum flexibility and functionality. Thus, you, and not the system, should be in control at all times: you should be free to interact with any tool at any time, without having to wait until another tool finishes execution. Tajo thus allows tools to run in parallel with other tools, including other instances of the same tool.

To simplify the problems of interacting with several tools at once, Tajo provides a highly-interactive window-based user interface. Each tool is represented on the display screen by one or more *windows*, which are just rectangular partitions of the screen.

The user interface, which is consistent across all windows, is largely based on visual imagery. The emphasis is on minimal typing and maximum assistance to the user. For example, a window can be "scrolled" up and down to view different parts of a file; there is no need to calculate character position numbers or line numbers. Similarly, operations are performed by pointing at words and pressing a mouse button, or by choosing an item from a menu. In general, you can see the available commands and options and can just select them with your mouse without having to type a complicated command line.

An integral part of the user interface is the concept of the global "current selection", which can be either a text string or a graphics icon. You choose the current selection with your mouse; doing so highlights your selection on the screen. Many commands use the current selection as an argument. For example, if you want to move text from one window to another, you select the desired location, push the **MOVE** key, and then select the text that you want to move. Among other things, the current selection mechanism allows you to use all (or part) of one tool's output as the input (argument) to another tool in a visual way.

1.1 Tools and windows

A *window* is a rectangular portion of the screen in which text or graphics can be displayed. There can be any number of potentially overlapping windows on the display at any time, and you can change their shape and size at will. Figure 1.1 is an illustration of a sample screen with several overlapping windows.

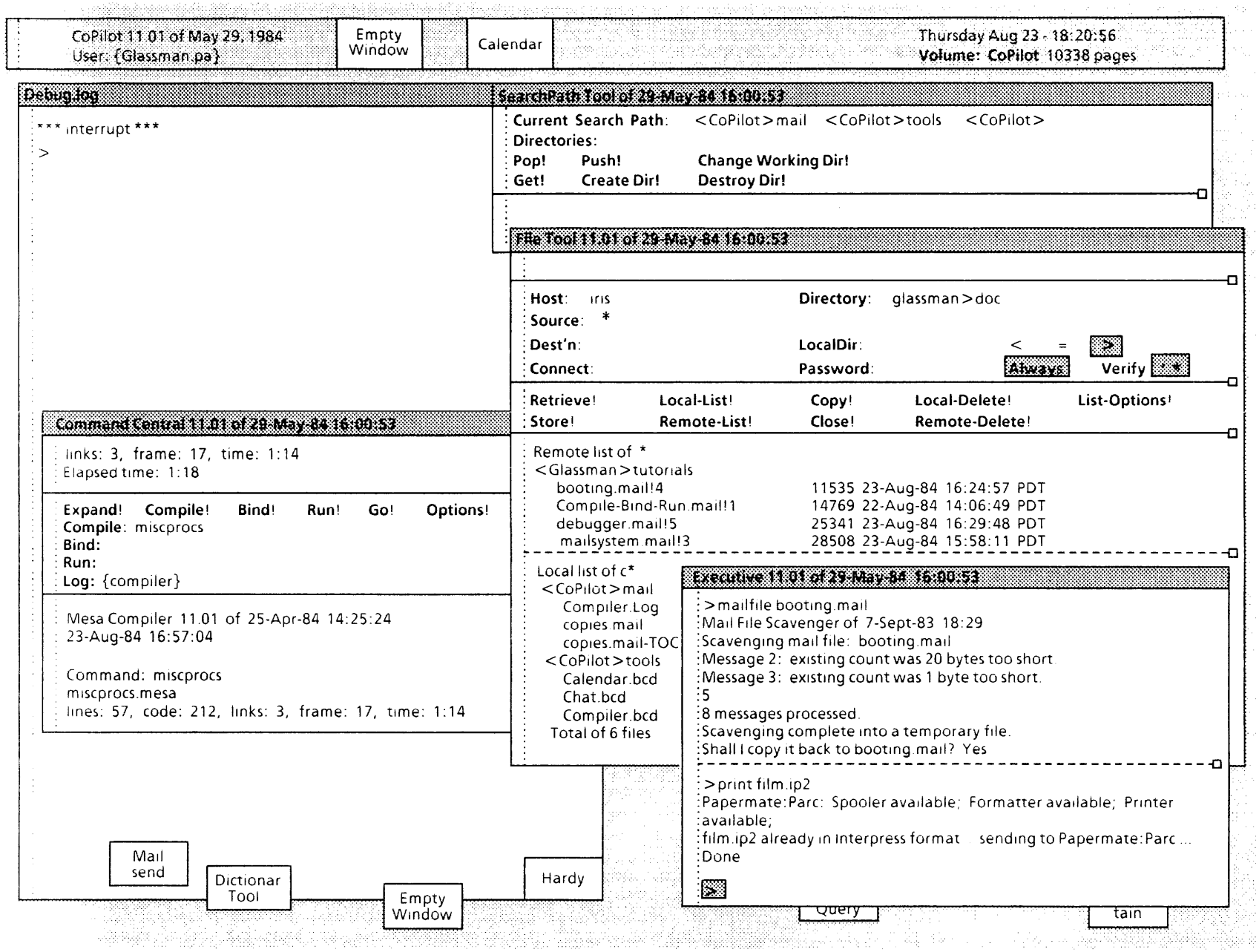

CoPilot 11.01 of May 29, 1984
User: {Glassman.pa}
Empty Window
Calendar
Thursday Aug 23 - 18:20:56
Volume: CoPilot 10338 pages
Debug.log
*** interrupt ***
>
SearchPath Tool of 29-May-84 16:00:53
Current Search Path: <CoPilot>mail <CoPilot>tools <CoPilot>
Directories:
Pop! Push! Change Working Dir!
Get! Create Dir! Destroy Dir!
File Tool 11.01 of 29-May-84 16:00:53
Host: iris
Directory: glassman>doc
Source: *
Dest'n:
LocalDir:
Connect:
Password:
Always
Verify
Retrieve! Local-List! Copy! Local-Delete! List-Options!
Store! Remote-List! Close! Remote-Delete!
Command Central 11.01 of 29-May-84 16:00:53
links: 3, frame: 17, time: 1:14
Elapsed time: 1:18
Expand! Compile! Bind! Run! Go! Options!
Compile: miscprocs
Bind:
Run:
Log: {compiler}
Mesa Compiler 11.01 of 25-Apr-84 14:25:24
23-Aug-84 16:57:04
Command: miscprocs
miscprocs.mesa
lines: 57, code: 212, links: 3, frame: 17, time: 1:14
Remote list of *
<Glassman>tutorials
booting.mail!4 11535 23-Aug-84 16:24:57 PDT
Compile-Bind-Run.mail!1 14769 22-Aug-84 14:06:49 PDT
debugger.mail!5 25341 23-Aug-84 16:29:48 PDT
mailsystem.mail!3 28508 23-Aug-84 15:58:11 PDT
Local list of c*
<CoPilot>mail
Compiler.Log
copies.mail
copies.mail-TOC
<CoPilot>tools
Calendar.bcd
Chat.bcd
Compiler.bcd
Total of 6 files
Executive 11.01 of 29-May-84 16:00:53
>mailfile booting.mail
Mail File Scavenger of 7-Sept-83 18:29
Scavenging mail file: booting.mail
Message 2: existing count was 20 bytes too short
Message 3: existing count was 1 byte too short.
5
8 messages processed.
Scavenging complete into a temporary file.
Shall I copy it back to booting.mail? Yes
>print film.ip2
Papermate:Parc: Spooler available; Formatter available; Printer available;
film.ip2 already in Interpress format sending to Papermate:Parc ...
Done
>
Mail send
Dictionar Tool
Empty Window
Hardy
Query
tain

Figure 1.1

Tools own windows: each tool usually owns one window, although a tool is not required to have a window, and may have several windows. You communicate with a tool through its window by choosing commands or values from menus, invoking commands, and typing or editing text. Windows exist so that tools can communicate with users: if a tool is not going to be used by a human, there is no need for it to have a window.

Windows are also the mechanism by which the environment isolates tools from one another on the display. Thus, you alternate your attention among the various windows on the screen as you alternate your attention among the corresponding tools. When you are temporarily through with a tool, you can conceptually "put it away" for a while by making its window tiny (that is, by putting the window into its iconic form. Notice the tiny windows at the bottom of Figure 1.1.) A tiny window retains all window state, such as parameters that you have given the tool, options that you have turned on, or messages that the tool has posted to you. You can also *deactivate* a tool. Deactivating a tool is like putting it away in a tool box: the window is destroyed and all window state is lost, but the tool itself is still available for future use.

Tools are unobtrusive; you can use them when you need them, and ignore them when you don't need them. Consistent with the principle that you are in control, a tool will not make itself large, obscure other windows, or otherwise attempt to attract your attention. Thus, a tool can make no assumptions about window layout. In particular, a tool cannot make any assumptions about the size or shape of its window. For instance, if a tool attempts to produce output based on the current width of its window, the window may change shape between the time that the tool checked the width and the time that the output is sent to the window.

1.2 User interface

For the approach of multiple windows representing multiple tools to be most effective (for you to be able to change activities quickly and easily), the user interface must be consistent across all tools. The perceptions, models, and conjectures that a user accumulates about a system are referred to as the "user illusion". The intent underlying Tajo is that you should see the tools only in relation to your needs and purposes, and that you should not have to concern yourself with their internal workings. Thus, Tajo tries to create a consistent user illusion that enables you to predict instinctively how to use any tool, regardless of whether or not you have had any previous exposure to it. The principle is that similar actions, in different contexts, should have predictable results. (This is sometimes called the Law of Least Astonishment.)

Tajo therefore provides extensive routines for performing basic display and input/output management, manipulation of windows and subwindows, scrollbars, current selection management, menus, and cursor control. For example, there is a window package that provides general window manipulation functions, such as creating and destroying windows, painting text, curves, shades, and bitmaps, and moving windows.

The built-in tools all use these user interface routines, and the individual tool builder is encouraged to use them as well. This approach has two advantages. It presents the user with a consistent, easy-to-use interface that frees him from the operating system details inherent in switching from one activity to the next. Similarly, it allows the programmer to concentrate on the central aspects of his program rather than on I/O management.

1.2.1 Window structure: subwindows

Windows are visually separated into *subwindows*. There are various classes of subwindows provided by the environment, each of which defines a different style of user interface to serve a specific purpose. When you want to write a tool, you can create a window for it from the existing subwindow classes, or you can develop your own subwindow class if the existing ones do not fill your needs.

One system-supplied subwindow type is the *message subwindow*. A message subwindow is used for displaying messages from a tool to the user; it is a read-only subwindow.

A second kind of subwindow, the *text subwindow*, provides a place where streams of text can be input, output, or manipulated. Text displayed in a text subwindow is managed by a common display and user input package, which provides a consistent way to select and modify text. A text subwindow allows you to do such things as scrolling, searching, positioning to a character index, and editing. For example, the File Tool has a text subwindow that contains a log of recent filing actions, and the mail tool (Hardy) has two text subwindows, one for the Table of Contents, and the other for display of messages. The main subwindow of an Empty (file) window is also a text subwindow.

A third kind of subwindow, the *form subwindow*, is used for gathering and modifying parameters. Sometimes a form subwindow contains both commands and parameters; sometimes there are two separate form subwindows, one for commands and one for parameters. The form subwindow is discussed in more detail in the next section. Figure 1.2 is a picture of the Sample Tool window, which has three subwindows: a message subwindow, a form subwindow, and a text subwindow.

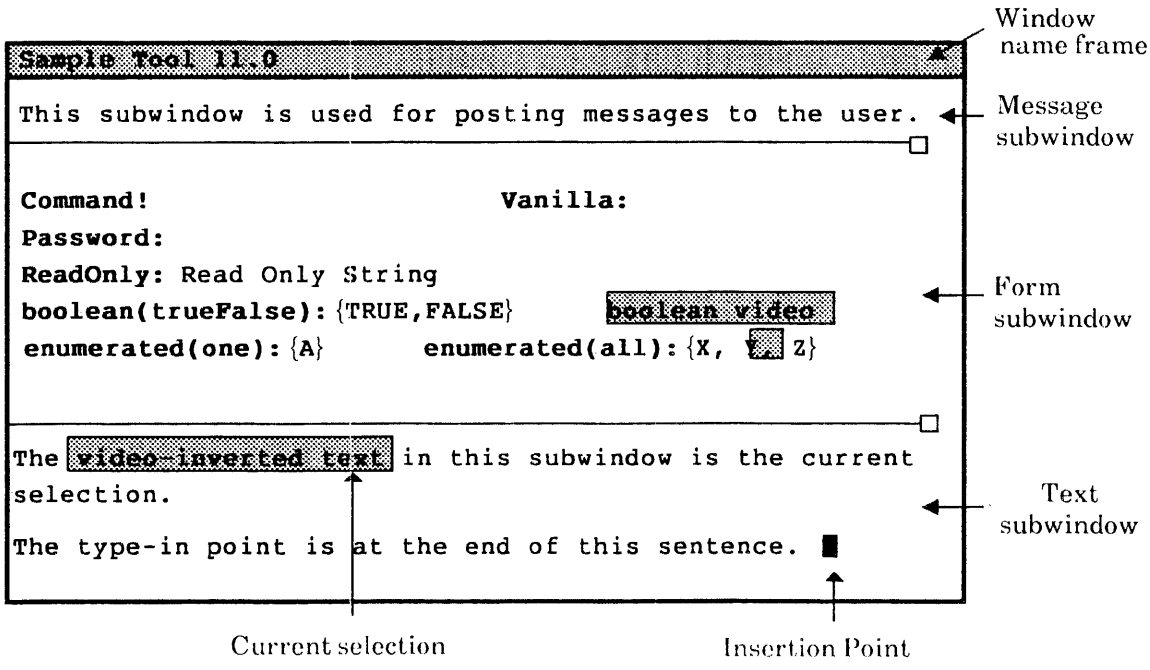


Figure 1.2 The Sample Tool

1.2.1.1 Form subwindows

In most systems, a command that requires multiple parameters dictates that they be collected in a specific order, usually via a command line. A program thus has complete control of the processor while it is getting input : you can't do anything else until you have finished interacting with that particular tool. This mode of operation is referred to as "get a command and execute it".

Such programs generally have a main procedure that initially receives control and then calls various procedures or subroutines. The program eventually calls a system routine, requests input, and waits for you to type something. It then must interpret the input, possibly through some command parser, and finally execute the desired command. The program controls the interaction: it has strong notions about the order in which information must be presented and about the actions that it is willing to recognize and act upon. The order in which information is provided is driven by the tool writer, not by the tool user.

The "get a command and execute it" paradigm conflicts strongly with the XDE philosophy that you should be free to interact with multiple tools simultaneously and never be restricted to interaction with just one tool. Tajo therefore provides *form subwindows* to prevent unncecessary restrictions on how a tool receives input, and to make parameter gathering easier, more flexible, and more uniform from one tool to the next.

The abstraction behind the form subwindow is a form, such as a personnel form or an income tax form, which has specific blanks to be filled in by the person using it. A form subwindow contains keywords, generally either the name of a command (indicated by a ! after the keyword), or a word suggesting necessary string input (followed by a : and space for you to insert your desired string), or a word suggesting necessary numeric input (followed by an =). The keywords thus remind the tool user of the possible options and the required input to each command.

Some form items have an associated menu that contains all possible values for the item. These are called *enumerated* items; they are designated by {} following the keyword. There are two different kinds of enumerated items: in one kind, the possible choices for the item are all listed within the brackets and the value currently in effect is highlighted; in the other kind, the possible choices are listed in a menu, and only the current value is displayed within the brackets. Both of these kinds are illustrated in Figure 1.1; you might want to take another look at that figure.

Figure 1.3 is an illustration of the Adobe Edit window, which has two form subwindows (and a message subwindow).

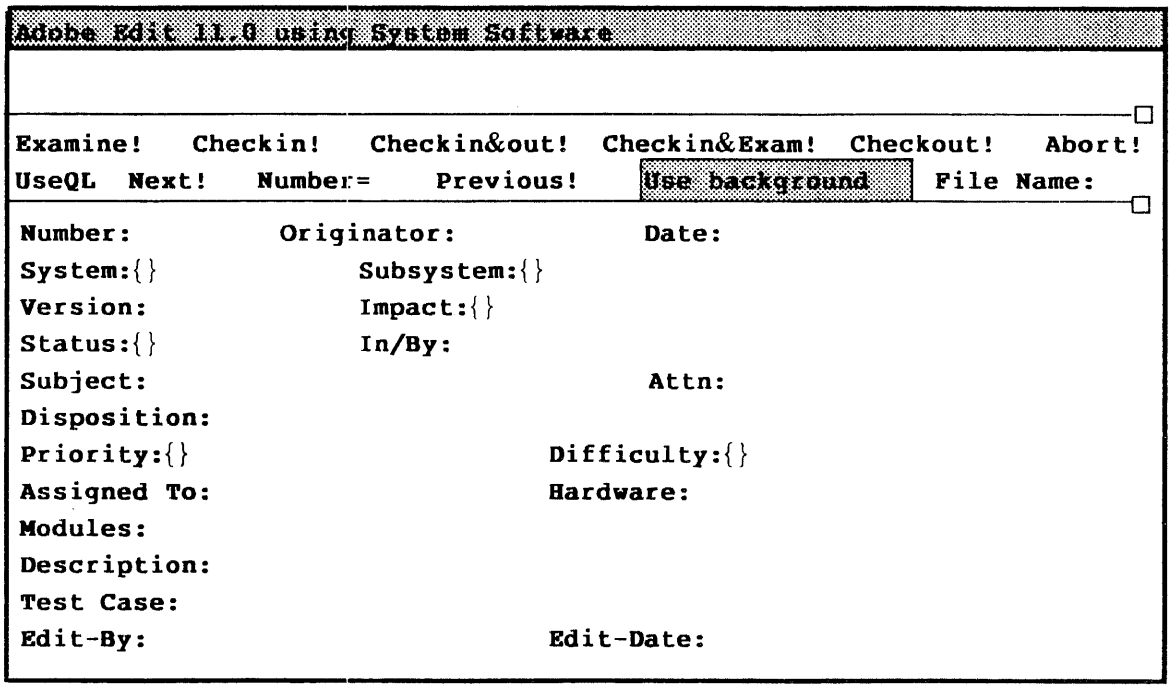


Figure 1.3 The Adobe Edit window

You can edit the parameters as much and as often as you like, using the keywords as reminders of what information is required, and then invoke commands. The code to execute a command is not called until you actually invoke the command. Form subwindows thus provide a means for the tool writer to inform the tool user of necessary input and possible options, without restricting the order in which the information is entered.

1.2.2 "Don't call us; we'll call you"

Tools are passive: they respond to user commands, but never seize control of the processor or act independently. Instead of a main procedure that calls subroutines, therefore, each tool contains an initialization procedure and individual command execution routines.

Loading a tool calls its initialization procedure, which then registers the available commands and user interface constructs with the system. When the tool is fully initialized, control returns to the system. Thus, if the tool is not actually processing a user action, there is no program counter location associated with the tool to which control must return. The tool simply provides a set of functions and arranges for the environment to notify it when you wish it to perform some action. This style is characterized by the phrase "Don't call us, we'll call you."

While a tool window is active on the display screen, Tajo provides facilities to notify it of user actions (mouse movement, keyboard transitions) that are directed toward it. The user actions are then converted into requests to execute commands, select menu items, and so forth. When the user requests execution of a tool command, the environment calls the client tool's execution routine to get the work done. (In general, a *client* is a program that uses the service of another program or system. A Tajo client is thus a program that uses the Tajo facilities.) The command execution routines are thus referred to as "call-back"

procedures, because the system will call back to those procedures when the user requests a specific action. (See also section 1.3.1.1.)

1.2.2.1 The notification process

The translation of user actions into procedure invocations is performed by several cooperating processes. (You can think of a *process* as something that uses the processor; check the Glossary for a more formal definition.) There is one very high-priority process that watches the hardware (keyboard and mouse) for user actions and queues them. A separate process then dequeues each user action, and determines which window the action is associated with. (Generally the window containing the input focus; sometimes the window containing the current selection. See the Glossary if you are not familiar with these terms.) Finally, the action or actions are looked up in a TIP (terminal interface package) table to determine which procedure in the associated tool is to be called.

The action lookup table, or TIP table, specifies translations between a sequence of user actions and a sequence of client actions. Each tool window has an associated chain of TIP tables. A user action is looked up in the first table associated with the designated window. If the event matches the left hand side of a statement in that TIP table, the right hand side (result list) of that statement is executed. If no match is found in that table, the next table in the chain is checked, and so on. If no match is found in any table, the event is discarded.

All of this is performed "automatically" by the Tajo facilities; you need not know anything about how a mouse or keyboard action is translated into a procedure invocation. (You will sometimes hear this function called "the notifier", since it is responsible for *notifying* tools of actions that are meant for them.)

1.3 Cooperation

The built-in tools conform to the XDE philosophy and guidelines, but there is no central facility responsible for ensuring that the individual tool builder follows the conventions as well. Thus, for example, there is nothing to prevent you from writing a tool that steals the notifier (so that you can't interact with any other tool) and then rearranges all the windows on the display. Similarly, programs must explicitly allocate and deallocate storage; there is no garbage collector to reclaim unused memory. All programs on a machine share the same pool of resources, and there is no scheduler watching for programs using more than their share of execution time, memory, or any other resource. Instead, the XDE depends on the assumption that programs are friendly, and that they are not trying to circumvent or sabotage the system; it relies on the voluntary cooperation of those who use its facilities.

1.3.1 The local file system

The XDE local file system is an example of how the environment depends on and exploits the principle of cooperation. In general, file systems must coordinate simultaneous access of files by concurrent client processes. (A client of the file system is a tool or program that uses its facilities.) Since most file systems consider processes to be antagonistic, they prevent one process from acting on a file if there is a chance that the actions will harm another process using the file. If several processes need to cooperate in the use of a file, they must communicate explicitly among themselves.

The XDE file system, on the other hand, views processes as cooperative, thus enabling a sophisticated sharing of files. The cooperation is provided by the file system instead of the programs themselves; the processes that share files need neither communicate explicitly nor know one another's identities. If one process wishes to use a file in a way that conflicts with the way a second process is using the file, the process using the file may be asked to relinquish it. For example, if a process (A) wants to write a file being read by another process (B), process B may be asked to stop reading the file. Process B can then decide whether or not to release the file.

Similarly, a process may ask to be notified when a file becomes available for a particular use. These facilities allow design strategies that would be impractical under other circumstances, secure in the knowledge that if an optional use of a file is interfering with other work, the offending program will be informed.

1.3.1.1 Call-back procedures

When a client program wants to use a file, it requests a *handle* for it from the file system. A file handle identifies the file in all subsequent calls to the file system. When a client finishes using the file, it must release its handle and relinquish its use of the file.

A request for a file handle includes an access parameter that indicates how the file is to be used. (For example, the access can be **readOnly**, **anchor**, **append**, **writeOnly**, **readWrite**, **delete**, or **rename**.) The file system then checks that the requested access does not conflict with other current uses of the file. When determining whether there is an access conflict, the file system tries to maximize the sharing of files. For example, consider the case of a client program that requests **read** access for a file, and another program already has **append** access for it. The file system will grant **read** access to the second process, but the read length will be fixed at the length that the file had when the request was made.

In some cases, however, the requested access conflicts with a current use of the file. For example, the file system must grant exclusive access to a process that wants to write a file, and all other accesses are considered to be conflicting. In such cases, the file system will check with the process using the file to see whether it is willing to relinquish its access. The file system thus asks its client programs to provide procedures (called **PleaseReleaseProc**'s) that it can call when such a conflict occurs. These procedures are thus also "call-back procedures", since the file system will call back to the client via these procedures when there is an access conflict.

When a process requests an access that causes a conflict, the file system will call the **PleaseReleaseProc** associated with each conflicting handle on the file, and ask the owner of each handle to release the handle. If all clients with conflicting handles release them, the request is honored and the new use is granted. Otherwise, the request is denied. Thus, a program that has registered a **PleaseReleaseProc** for a particular file can be confident that it will be notified if any other program is waiting for that file.

1.3.1.2 Access notification

A client can also ask the file system to be notifed when a file (or class of files) becomes available for some particular access. For example, when a client is denied access to a file, it might want to be notified when that file later becomes available. In this case, the client registers a **NotifyProc** with the file system. When the file system determines that the

conditions of a request have been satisfied, it calls the associated **NotifyProc**, passing the name of the file, a handle on the file, and the kind of access. Thus, **NotifyProc**s are also call-back procedures.

Because a client can acquire a file for a conflicting access before other interested clients have been notified that the file is available for some weaker access, there is no guarantee that a client will be called for every state change of a file. For instance, clients to be notified that a file is available for **readOnly** access will not be notified if another client acquires the file for **readWrite** access in the interim. When a client is notified, however, it is guaranteed that it can acquire the file for its desired access.

1.3.1.3 An example of cooperating files

Suppose that you load the compiler error log into a window to look at your compilation errors while you edit your source file. The file window program thus has a read access handle on the error log. After finishing the edits, you recompile the source, and the compiler needs to write a new error log. If you run the compiler without unloading the compiler log from your file window, there will be an access conflict: the compiler wants the file for write access, but the file window has a read access handle on it. The file system will then call the **PleaseReleaseProc** associated with the file window.

When the **PleaseReleaseProc** is called, the file window program checks the state of the window. If the file is being edited, it refuses to release the handle. Otherwise, it unloads the window, registers a **NotifyProc** for read access on the file, and relinquishes ownership of the file. The compiler can then write a new error log. When it finishes, it releases its write access handle on the file. The file system then notices that read access has become available on the file, so it calls the file window's **NotifyProc**. The file window program acquires the file for read access once more and reloads it into the window. Hence, a client will not be blocked if a file that it needs has been left loaded in a window, and file windows automatically update themselves to the most recent version of whichever files they contain.

2

Communications and network services

A Xerox Development Environment workstation equipped with the Tajo facilities provides wonderful support for most common programming and editing activities. However, XDE workstations are almost never used singly; instead, many individual machines are linked together with a communication network to enable communication among users and to provide shared access to expensive resources such as high-speed electronic printers.

Some of the machines attached to each network are dedicated to the management of a particular shared resource. A machine that contains such a resource is called a *server*; the software that makes it possible for you to use the resource is called a *service*. Some of the services provided by the Xerox Development Environment are printing, filing, mailing, internetwork routing (combines multiple networks into a single logical internetwork), and the Clearinghouse.

An interaction between a client and a service is always initiated by the client. A client always interacts with a service on behalf of a *user*. The user is generally a human, but may be some other entity (such as another service). When a client wants to access a service, it generally knows the name of that service. It must then use the *Clearinghouse* facility to find out the location (network address) of the appropriate server. The *Courier* remote procedure call facility is then used to actually invoke the desired operation. The *Authentication Service* is used to verify that the client is authorized to use the service.

2.1 Network Services

This chapter overviews some common services, and then describes the various pieces of communication software that cooperate to allow you to access those services.

2.1.1 File Service

A file server is a processor attached to a large capacity magnetic file store. A remote file server is used when local storage is inadequate, or when documents need to be shared. Remote file servers also provide substantial backup facilities, which are not usually available on the local disk. Access to the file server is controlled by the file service, which allows clients to do such things as retrieve a file, replace a file, change its name, or change some other identifier associated with the file.

2.1.2 Print Service

The Print Service provides a high-quality printing facility that can be accessed by other stations on the internet. An electronic printer "paints" images on paper using a laser much like a television set uses an electron beam to paint pictures on a screen.

When a print command is issued, the text file is converted into an Interpress master, which is then sent through the ethernet to a print server. Print Services usually accept and print only those documents in Interpress format, which is a file format used to encode documents to be transmitted to a printer. A document in Interpress format can be considered a kind of program. The characters transmitted to the Print Service are really instructions to construct a page of information. Interpress has instructions which apply to ordinary text as well as foreign alphabets and graphics.

Anything that can be placed on paper can be described in Interpress, though some printers may not be able to print what is described because it is too complex. For instance, some devices have limited internal storage and may not be able to handle a page with a large number of different fonts. Some may not have the computational power to rotate shapes or pages an arbitrary number of degrees. And few printers will have color capability or be able to reproduce images at arbitrarily high bit densitites.

2.1.3 Mail Service

Electronic mail is written communication between people that is transported electronically, not physically. An electronic Mail Service provides speed, reliability, and convenience: messages are usually received within a few minutes of being sent; there is a very low probability of a message being lost or damanged, and users are free to read and send mail at their convenience. It is particularly convenient to exchange electronic messages with people who are located in different time zones.

2.2 The Clearinghouse

Distributed objects, such as workstations and servers, are identified with *network addresses*. An XDE network address is a unique 48-bit number, assigned by Xerox, that identifies the location of an object in the network. Network addresses, however, are very unintuitive for humans. (It is much easier to think of a printer as "Nevermore" than as a 48-bit number.) Objects are therefore also given names; all objects are named according to the same hierarchical naming system. (Thus, an object has a *fully-qualified* name that consists of its local name suffixed by a domain and organization, as in *Nevermore:OSBU North:Xerox*. The name "Nevermore" must be unique within the domain "OSBU North", but there could be another object named *Nevermore:OSBU South: Xerox*.)

When a client wishes to access a service, it knows the *name* of that service. The name must then be mapped into the current *network address* for that object. The network address for a particular service may vary occasionally, as when the server is physically moved from one location to another. The binding between network address and name is therefore not permanent. Instead, the binding is done dynamically (each time the service is needed) by a service called the *Clearinghouse*. Naming thus provides a level of indirection between network address and users, since machines change location more frequently than they change name.

For each object registered in the Clearinghouse database, the service stores a unique name (chosen by the client), and related information, such as the location and type of the object. When a client wants to use a service, it presents the name of the service to the Clearinghouse, which returns the network address. The Clearinghouse thus enables users to specify resources with human-readable textual names, rather than machine-readable network addresses.

2.2.1 Clearinghouse structure

The Clearinghouse is *decentralized* and *replicated*. That is, instead of one global physical Clearinghouse server, there are many local Clearinghouse servers, each storing a portion of the global database. Conceptually, however, there is only one global database called "the Clearinghouse".

Decentralization and replication increase efficiency (it is faster to access a Clearinghouse server physically nearby), security (each organization can control access to its own Clearinghouse servers) and reliability (if one Clearinghouse server is down, perhaps another can respond to a request).

A client of the Clearinghouse may query the Clearinghouse about any named object, regardless of the location of the object, the location of the client, or the present distributed configuration of the Clearinghouse. The implementation makes no assumptions about the physical proximity of Clearinghouse clients to the objects whose names they present to the Clearinghouse.

2.2.2 Clearinghouse updates

Like many other aspects of the Xerox Development Environment, the Clearinghouse depends on the goodwill and voluntary cooperation of its clients. A user who has caused Clearinghouse information to be invalid must send the appropriate update: the responsibility for initiating updates to the database rests with the users of the system.

Thus, a Clearinghouse server database may occasionally have incorrect information, especially since the Clearinghouse is distributed. It cannot guarantee that the error in its database will be corrected. Since clients of the Clearinghouse are pieces of hardware and software, and not people, all aspects of the Clearinghouse interaction, including fault-tolerance, must be fully automated. Information gleaned from the Clearinghouse should therefore be treated as only a hint, and clients should check the information before using it. For example, before printing a document on a printer located by the Clearinghouse, the client should check to make sure that it is really a printer.

2.3 Authentication Service

When initiating interaction with a service, a client presents *credentials*, which must be verified by the *Authentication Service* to prove the identity of the client or user. At each subsequent request, the client presents a verifier (essentially a time stamp), which helps ensure security.

2.3.1 The authentication process

Each client of the Authentication Service recieves its own specially encrypted password, called a *key*. A sender (*initiator*), that wants to authenticate itself to a receiver (*recipient*) contacts the Authentication Service with his name, and the name of the service he wishes to access. The Authentication Service encrypts a set of credentials with the recipient's key. It then sends the encrypted credentials back to the initiator along with a conversation key encrypted with the initiator's secret key. The initiator then has a set of credentials that he cannot decrypt, and a conversation key that he can decrypt (because he knows his own key). Figure 2.1 diagrams this process.

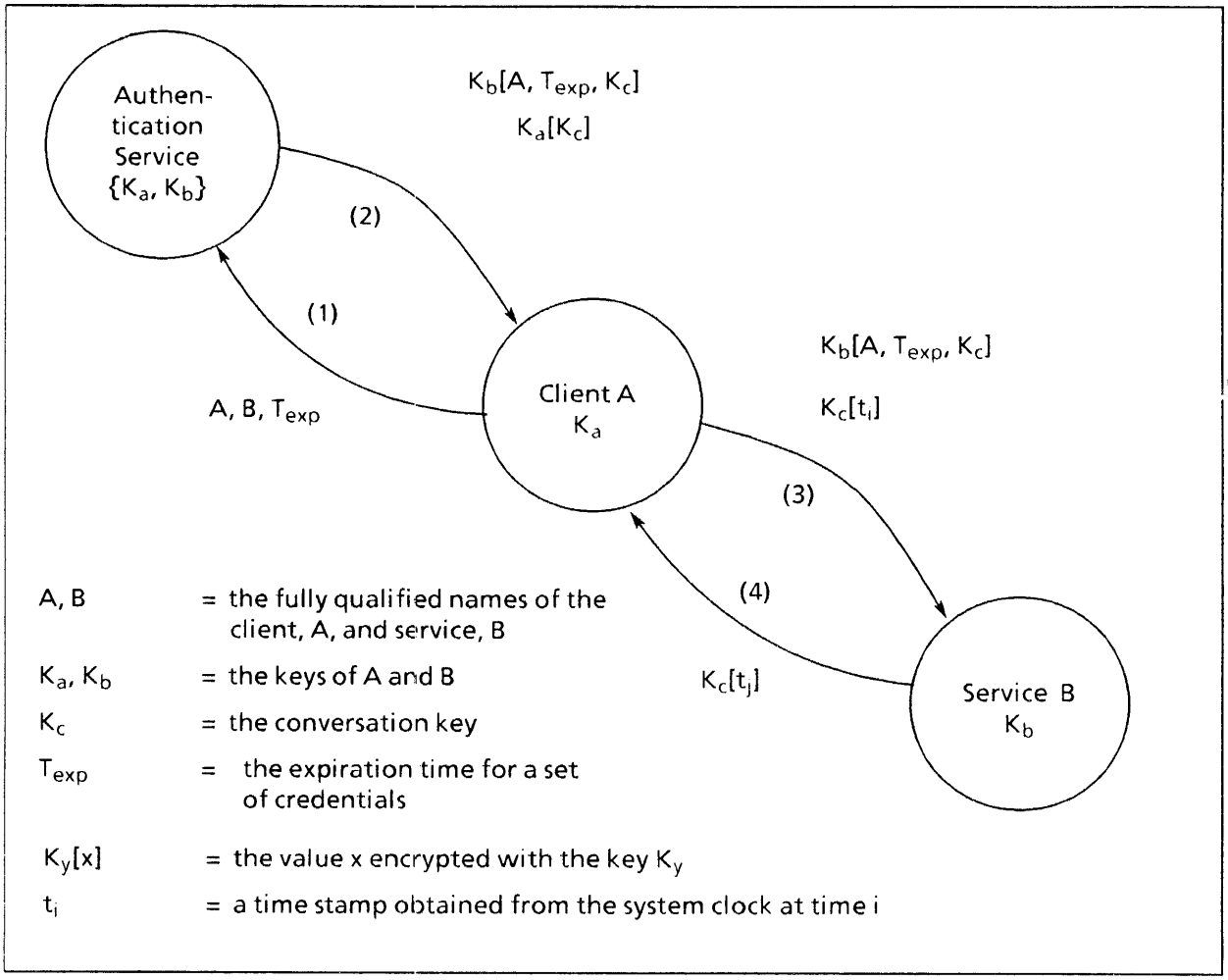


Figure 2.1 The Authentication Process

The initiator uses his key to decrypt the conversation key, and then re-encrypts it with a time stamp, called a *verifier*. The original credentials and the conversation key plus verifier are then sent to the recipient. The recipient decrypts the credentials with his own key: the decrypted credentials contain the name of the initiator, the conversation key, and a date and time, called the expiration date, after which the credentials are no longer valid.

Since only the recipient can decrypt the credentials, the recipient knows that the initiator is authentic.

The recipient then uses the conversation key to decrypt the verifier, and determines if the time stamp is reasonably close to the current time. This allows the recipient to detect a replay. (To be effective, a verifier may only be used once.) Only the initiator can encrypt the verifier (the time stamp), and only the recipient can decrypt it, since only these two parties know the conversation key. Each time the service needs to interact with the client initiator it must send the client the time stamp encrypted with the conversation key. For example, in order for the service to send a message to the client (ie, service is complete), it must send the time stamp with the message.

2.4 Courier

The Clearinghouse tells the client where a desired service is; the Authentication Service allows it to access that service; the *Courier* facility allows the desired function to be performed.

Because the procedure is the major control and data transfer mechanism in Mesa, Courier uses the procedure call as a metaphor for the exchange of a remote request and its positive reply. Courier thus extends the concept of procedure call from a mechanism that provides for transfer of control and data within a single computer to a mechanism that provides for transfer of control and data across a communication network.

Courier makes the semantics of a remote procedure call as close as possible to those of a local procedure call in Mesa. An operation is modeled as the name of a remote procedure; the parameters of the Courier request become the arguments of that procedure, and the parameters of the positive reply are the procedure's results. Because the Mesa language provides a *signal* facility for exception handling in local procedures, Courier also uses exception conditions (signals or errors) as a metaphor for the return of a negative reply. By using this model, Courier permits clients to make procedure calls without regard for the physical location of the service being accessed.

2.4.1 Remote programs

A family of remote procedures and the exception conditions those procedures can designate are said to constitute a *remote program*. Every remote program is assigned a program number, which identifies it at run time, and a version number, which distinguishes sucessive versions of the program.

A remote program usually represents a complete service, and its remote procedures represent the primitives of the service. One remote program, for example, might provide a file service and contain remote procedures to open or close a directory, and to create, delete or retrieve a file. Another remote program might provide a mail service and contain remote procedures to retrieve a piece of mail, or check the status of a mail server. A remote program to provide a print service might contain procedures to check printer properties, interrogate printer status, and print a document.

2.4.2 Courier operation

Figure 2.2 diagrams the model for making a remote procedure call; refer to it as you read the next two paragraphs.

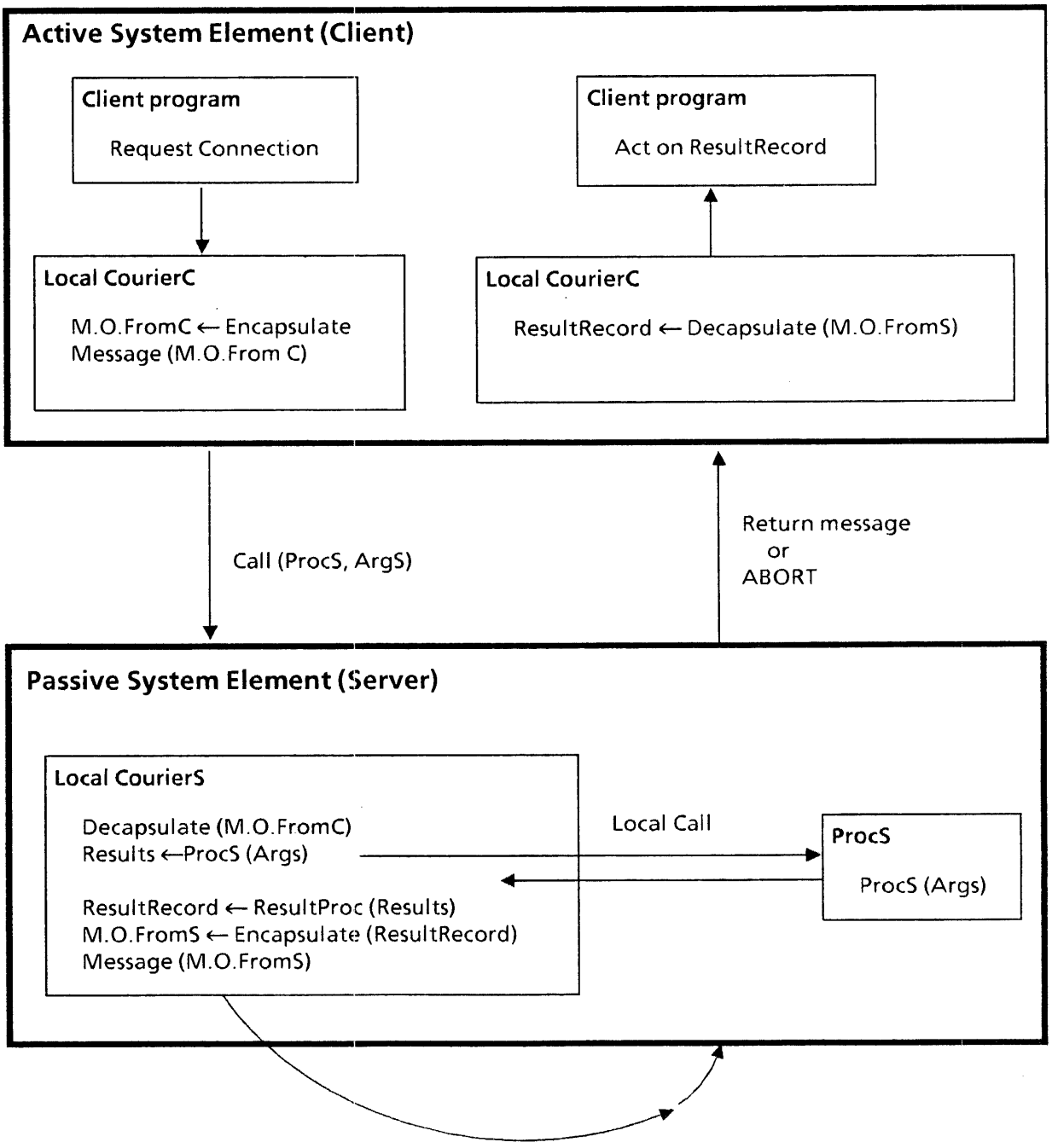


Figure 2.2 Courier Model

A local program on the client machine (C) initiates the process by making a "standard" procedure call, with the name and version of the remote procedure it wants to call, and the arguments to be passed to it. The courier stub on the client machine then encapsulates this information into a *Courier message object* (defined by the Courier protocol). The

message object (MO) is then transmitted across the network and delivered to a Courier on the destination server, which decapsulates (unpacks) it, and makes a local procedure call to perform the requested operation.

When the procedure on the server finishes, it calls a procedure known as the *results procedure* to inform Courier that it is done and that the results record may be created. This procedure sends a Courier *return message* back through the net to the client Courier, which returns results to the local user.

2.5 Ethernet architecture and protocols

There are many different kinds of local computer networks, such as Ethernet, Mitrenet, Primenet, LocalNet, and Cambridge Ring. The Xerox Development Environment uses the Ethernet.

The broad features of an ethernet are that it:

- has relatively high data rates [1..10Mbits /sec]
- is able to cover a geographic distance spanning about one kilometer (typically within a building or small set of buildings)
- can support several hundred independent devices, each with fair access to the system
- has good error characteristics, good reliability, and minimal dependence upon any centralized components or control
- enables effficient use of shared resources, particularly the commmunications network itself
- has stability under high load
- permits easy installation of a small system, with graceful growth as the system evolves
- allows ease of reconfiguration and maintenance
- is available at low cost
- has broadcast capability

The ethernet is a broadcast network: that is, there is a single communication channel that is shared by all hosts. Every receiver on the cable hears everything transmitted by every sender; receivers simply discard messages not addressed to them. There is no central resource to control communication among system elements: access to the channel by hosts wishing to transmit is coordinated by the hosts themselves, using a statistical arbitration scheme called carrier sense multiple access with collision detection (CSMA/CD).

2.5.1 CSMA/CD: Theory of operation

With the CSMA/CD approach, there is no central controller that manages access to the channel, and there is no pre-allocation of time slots or frequency bands. A station wishing to transmit "contends" for use of the common communications channel (sometimes called the ether) until it "acquires" the channel; once the channel is acquired, the station uses it to transmit a *packet*. The packet is the fundamental unit of information flow. An ethernet

packet contains control information (the header), source and destination network addresses, and digital data. Data length may range from 512 bits to 12096 bits.

To acquire the channel, a station checks whether the network is busy (that is, uses *carrier sense*) and defers transmission of its packet until the ether is quiet (no other transmissions are occurring.) When quiet is detected, the deferring station immediately begins to transmit. During transmission, the station listens for a collision (other transmitters attempting to use the channel simultaneously). Collisions should occur only within a short time interval following the start of transmission, since after this interval all other stations will sense the transmission and defer. This time interval, called the *collision window* or the *collision interval*, is based on the round-trip propagation time between the farthest two points in the system.

If no collisions occur during this time, a station has *acquired the ether* and continues transmission of the packet. If a station detects a collision, the transmission of the rest of the packet is immediately aborted. To ensure that all parties to the collision have properly detected it, any station that detects a collision invokes a collision consensus enforcement procedure that briefly jams the channel. Each transmitter involved in the collision then schedules its packet for retransmission at some later time. To minimize repeated collisions, each station involved in a collision tries to retransmit at a different time; each individual retransmission begins afer a delay period whose length is determined by random selection.

Under very high load, short periods of time on the channel may be lost due to collisions, but the collision resolution procedure operates quickly. Channel utilization under these conditions will remain high, particularly if packets are large with respect to the collision interval.

2.5.2 Protocols

For distributed system elements to communicate with one another, certain standard *protocols* must be observed. A protocol is essentially an exact technical description of events that must take place for a service to be successfully provided.

To reduce the design complexity of the network architecture, protocols are layered by conceptual abstraction. The purpose of the layering is to offer certain services to higher layers while shielding them from the details of the implementation. The number of layers, the name of each layer, and the function of each layer differ among the various network systems. The XDE uses a four level categorization, called the Xerox Network System (XNS) model. For those of you familiar with other models, Figure 2.3 diagrams the correspondence between the XNS model and the OSI model. The OSI (Open Systems Interconnect) model is a seven-layer model developed by the International Standards Organization.

7	Application	Serves a user directly	*Filing, Printing, etc.* *"Level 3"*
6	Presentation	Reconciles formats of files, representations of integers, etc.	*Courier*
5	Session	Facilitates interactions between cooperating parties	
4	Transport	Provides error recovery, data integrity, flow control	*Internet Transport Protocols* "Level 2"
3	Network	Routes packets through an internet	*Internet Datagram Protocol* "Level 1"
2	Data Link	Defines packets or frames	*The Ethernet Specification* *"Level 0"*
1	Physical	Defines plug shape, electrical signals, physical circuits	

Figure 2.3 Open Systems Interconnect Reference Model

The XNS system has no protocol corresponding to level 5, the session layer. In the Xerox network architecture, the linking of two parties is actually handled by the "transport" layer. Although the Clearinghouse Service may be used prior to establishing the link, this service properly belongs in the "application" layer.

2.5.3 XNS protocols

The rest of this chapter describes the XNS protocol layers. Remember that the protocol architecture is open-ended and permits evolution and growth. The protocols provided fulfill most common communication needs: implementors with special needs can use the layered protocols to build their own protocols.

Please refer to figure 2.4 for the following discussion.

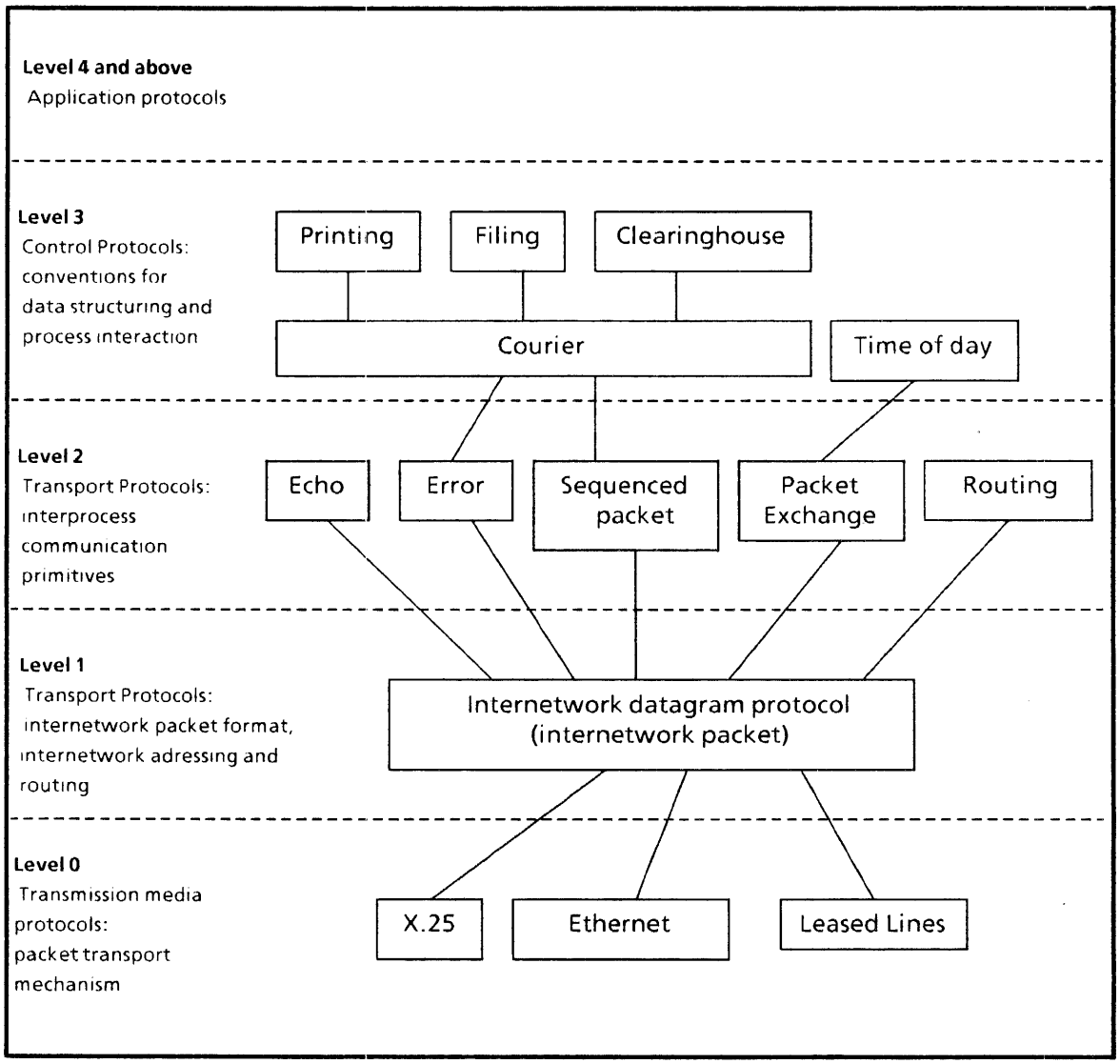


Figure 2.4 XNS Protocols

2.5.3.1 Level 0 protocols

The function of a level 0 protocol is to move data across a physical transmission medium. Level 0 protocols thus include specifications such as hardware interfaces, electrical and timing characteristics, bit encodings, and line control procedures. Level 0 protocols also include network-dependent packet formatting conventions; these conventions are naturally specific to the particular transmission medium being used (such as an ethernet cable and transceiver). Figure 2.5 is a picture of an Ethernet packet.

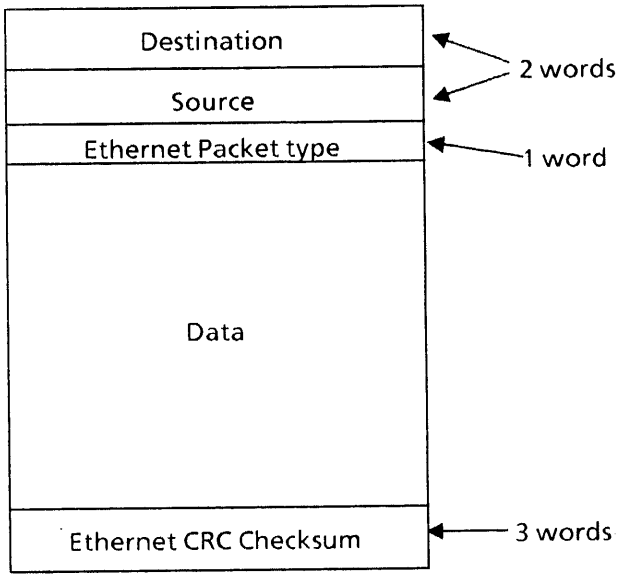


Figure 2.5 Ethernet packet

Level 0 protocols are either *broadcast* (such as the ethernet) or *point-to-point* (such as the synchronous point-to-point protocol used on phone networks.) In either case, a level 0 protocol will deliver packets only to machines directly physically connected to the transmission medium. In a broadcast network such as the ethernet, where every machine "sees" every packet that is sent, level 0 protocols also provide the way for a machine to decide whether a packet is addressed to it. When a machine receives a packet, the hardware can directly compare the address on the packet with its own machine identification. If the addresses don't match, the machine ignores the packet. The comparison process is very fast, so very little machine time is wasted.

The Xerox level 0 protocols provide only a "best-effort" delivery to directly connected machines: packets are delivered with high probability, but no guarantee of success. No notion of a *connection* exists at this level; packets are delivered individually and without guarantee. Reliability and transmission of packets to machines that are not directly connected to the physical network are provided as needed or by higher-level protocols.

2.5.3.2 Level 1 protocol

There is only one XNS level 1 protocol: the *internet datagram protocol* (IDP). The function of IDP is to address, route, and deliver standard packets, or *internet datagrams*. An *internet* is just a collection of physically distinct and potentially heterogeneous networks: the predominant network will generally be the ethernet, but an internet may also contain leased lines and other public or private networks. A *datagram* is a standardized internet packet which is media-, processor-, and application- independent, and has no relationship to any other packet in the system. (The word "datagram" thus also implies that each packet is handled separately, and that no resources are dedicated specifically to any one datagram.)

The IDP defines the format for the standard internet datagram. IDP thus serves as the common layer that unifies all the different media that are possible at level 0 and presents a uniform interface to all the layers above. The interface between levels 0 and 1 is very simple: IDP passes down a packet and a destination address, and level 0 encapsulates the packet (according to the protocol for that particular physical transportation medium) and does it its best to deliver it to the specified host. When a packet arrives at a host, level 0 decapsulates it and passes it up to IDP.

This means that IDP is responsible for the *routing* of a packet from source to destination. The packet itself contains only the ultimate destination address; the mapping of an actual route from source to destination is done dynamically. The piece of software that performs the logical function of switching (routing) packets between sockets on a single machine, or between machines on an ethernet, or between ethernets, is called a *router*. There is a router in every Xerox machine, although not all routers perform the full set of routing capabilities. For example, the router in a workstation only routes packets within the machine or within the network, but not across the internet.

A router that routes among networks is called an *internetwork router*, and provides a service called the *Internetwork Routing Service (IRS)*. A machine connected to two or more networks and providing the Internetwork Routing Service is called a *gateway*. The IRS routes a packet from its own ethernet (or other local area network) to another Internetwork Routing Service, which delivers it to the destination ethernet, and ultimately to the destination machine.

Every router maintains a *routing table*, or diagram of the network, that is used to direct packets toward another router on the way to their destination, along the path of shortest delay. The maps maintained by a workstation are a subset of the maps maintained by the internetwork routers. Each IRS map (routing table) of the internetwork is composed of information about remote networks: how far away each is, and the next IRS in the path to get there. When an IRS receives a packet to be forwarded to a remote network, it uses its map to send the packet to the next IRS on the shortest path to that network. The packet eventually reaches its destination host, either by being broadcast out onto the destination network (if it's an ethernet), or by point-to-point reception (if the host is on a medium other than an ethernet.)

The routing tables are maintained dynamically rather than statically: IRS's that are attached to the same ethernet exchange their maps on a regular basis. Thus, routing tables are not created on installation of an IRS; instead, a newly installed IRS gradually learns ofthe complete internetwork map from neighboring IRS's. Likewise, changes to the network gradually propagate from one IRS to another until all IRS's reflect the change.

The routing table itself is kept at level 1, but creating and updating is done by the IRS, in conjunction with a level 2 protocol called the Routing Information Protocol. This protocol basically sends *routing packets* around the network, using them to verify and update the information currently in the routing table, and to give out new information.

2.5.3.3 Level 2 protocols

The level 2 protocols give structure to a stream of related packets. Because different applications need different degrees of reliability and functionality, the protocol

architecture allows for many implementations of level 2 protocols. There are two basic kinds of level 2 protocols: *connection-based*, and *connectionless*.

Connectionless protocols are used to exchange packets that require little or no state, for example, requesting the time from a time server. Another good example of a connectionless protocol is the Echo protocol. The Echo protocol is a debugging protocol used to verify the existence and correct operation of a host, as well as the shortest path to it. The protocol simply specifies that all Echo protocol packets received shall be returned to their source. Thus, there is no need for a connection between machines, nor for any special reliability guarantees.

Connection-based protocols, on the other hand, enable an extended conversation between two machines in which more information is conveyed than can be sent in one packet. The Sequenced Packet Protocol (SPP) is an example of a connection-based level 2 protocol: it provides reliable, sequenced, and duplicate-suppressed transmission of successive internetwork packets.

The SPP makes a connection between two processes in different machines, and carries a sequence of packets in each direction. Each packet gets a sequence number when it is transmitted by the source. Sequence numbers are used by the recipient to order the packets, to detect and suppress duplicates and to acknowlege receipt of the packets. Each direction of flow is independently sequenced.

The Sequenced Packet Protocol provides reliable communication, in the sense that the data is reliably sent from the source's packet buffer to the destination's packet buffer. However, no guarantee can be made as to whether the data was succesfully retrieved by the destination client or whether the data was appropriately processed. This final degree of reliability must lie with higher level protocols, such as level 3 protocols.

2.5.3.4 Level 3 and above

Level 3 protocols are concerned more with the content of data than with communication per se. Level 3 protocols thus allow functions such as telnet and Courier (see section 2.4). For example, the File Transfer Protocol (FTP) consists of a set of conventions for talking abut files and a format for sending them through the net. Protocols at level 4 and above are applications protocols.

3

The supporting technology

The Xerox Development Environment is unusual in that the goals of the environment were formulated before development of the programming language, the operating system, or the hardware architecture was begun. Thus, all three pieces are specialized for the XDE, and provide extremely strong support for its goals and facilities. This chapter provides a brief overview of each piece, and discusses the ways in which it supports the design goals of the XDE.

3.1 The Mesa language

The Mesa language is a structured, strongly-typed language similar to PASCAL. Students of programming languages will also discern influences from Algol 68, BCPL, and several other system implementation languages.

Mesa's extensions of PASCAL are largely intended to simplify the building of large systems where many programmers work on individual components. In particular, Mesa provides for and enforces extensive modularization: large programs are built out of smaller pieces called *modules*. Modules are written, compiled, and tested separately, and then integrated together with complete assurance of matching data types (type safety). Mesa modules are thus a "programming in the large" mechanism for partitioning a system into manageable units, while still maintaining strict type-checking.

Mesa also provides support for multiple concurrent processes in the form of a *monitor* facility that enables synchronization between logically asynchronous processes. Mesa monitors will be discussed more fully in section 3.1.4.

3.1.1 Modules and interfaces

Interfaces are the mechanism by which modules share information about variables, procedures, constants, and types. There are two basic kinds of modules in Mesa: **DEFINITIONS** (or *interface*) modules, which define interfaces, and **PROGRAM** modules, which contain the executable code to implement the interfaces.

Interface modules typically declare some types, variables, and constants, and specify a collection of procedures that act on values of those types. An interface thus defines an abstraction; all operations on a class of objects are collected in a single interface. An

interface contains no executable code; it only contains enough information to allow the compiler to type check other programs that use the declared symbols. For example, a procedure in an interface is only a declaration; the procedure body is not part of the interface. Interface modules thus compile into symbol tables.

The data and executable code (such as procedure bodies) to implement an interface are contained in the second kind of module, called a *program module*. A program module that provides code for some or all of the symbols defined in an interface is said to *export*, or *implement* that interface. A module that uses non-constant declarations (e.g., exported types and procedures) from another module *imports* that interface. A program that imports an interface is a *client* of that interface. Several program modules can cooperate to implement an interface, with each module implementing some subset of the interface.

An interface can be thought of as a contract between client and implementor: the interface specifies items that are available for clients to use, but doesn't say how they will be provided; the implementing module(s) determine the details of the implementation. An interface is thus the link between client (user) and implementation (supplier): a client module need not communicate directly with an implementation module. Figure 3.1 illustrates this relationship.

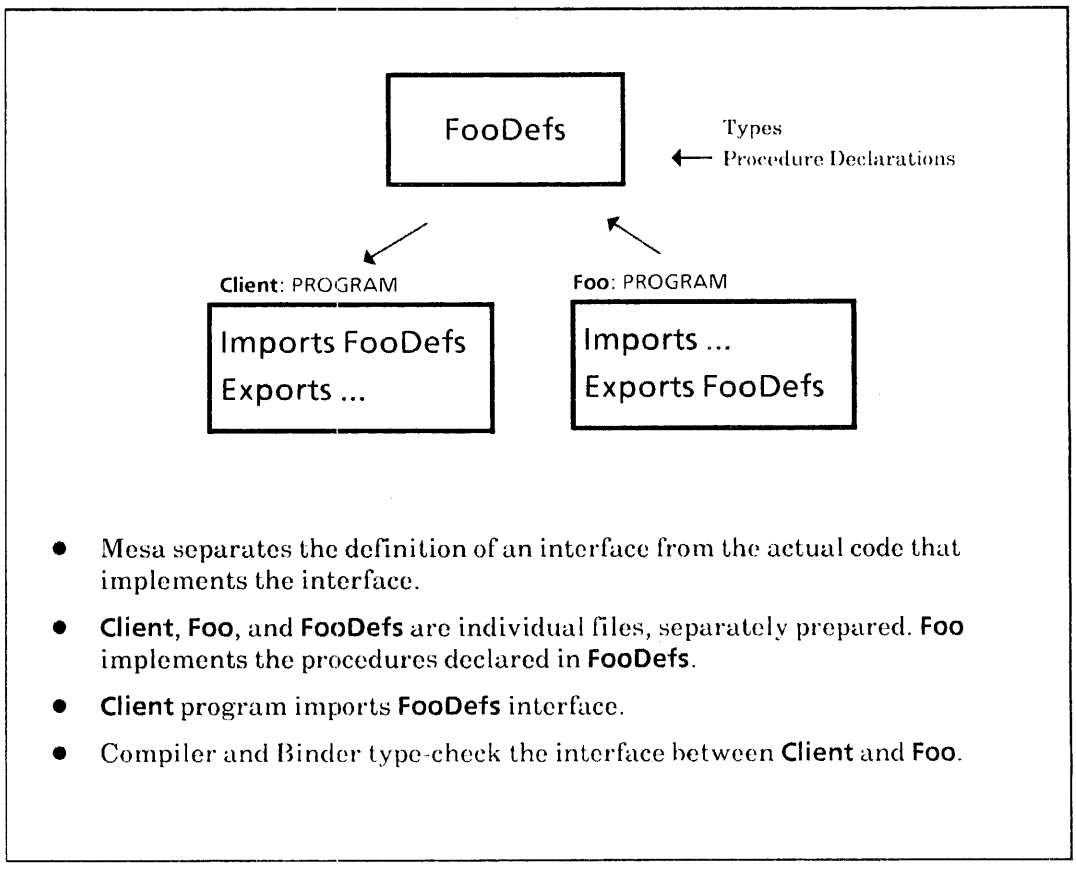


Figure 3.1 Mesa modularity

The Mesa approach to interfaces has several important advantages:

- Once an interface has been agreed upon, construction of the importer and exporter can proceed independently. In particular, interfaces and implementations are decoupled. This facilitates information hiding, and also permits minor changes to exporting modules without invalidating a previously established interface.
- Interfaces enforce consistency in the connections among modules. The operations upon a class of objects are collected into a single interface, not defined individually and in potentially incompatible ways.
- Nearly all of the work required for type-checking interfaces is done by the compiler.

3.1.2 Binding

Roughly speaking, *binding* is the process of matching the imports of one program to the exports of another. In simple systems, each interface is exported by exactly one module, and the binding can be done by the loader. But in more complex cases, there might be several different modules in the system that can implement the same components of an interface under somewhat different conditions, or with somewhat different performance. Describing exactly which modules are to supply which components to which other modules can then become rather subtle. A whole language, called C/Mesa (configuration Mesa), was devised to describe these subtle cases.

The binder is the program that reads a C/Mesa description and builds a runnable system by filling imports request from exports according to the recipe specified in the C/Mesa program. The input to the binder is a *configuration description* (config) file, which lists the basic modules and describes how they are to be combined and initialized. The job of the binder is to locate and assign actual code to all symbols used in the configuration; that is, to match import requests to export requests.

The binder combines modules, and possibly previously bound configurations, to produce a new object file. (A *configuration* is one or more modules that have been bound together; program modules and configurations are interchangeable in building a larger configuration.) Both the compiler and the binder produce object files; the compiler produces a simple object file that contains code and binding information for just one module; the binder produces complex object files from simple ones.

See figure 3.2 for an example of modules cooperating to implement an interface.

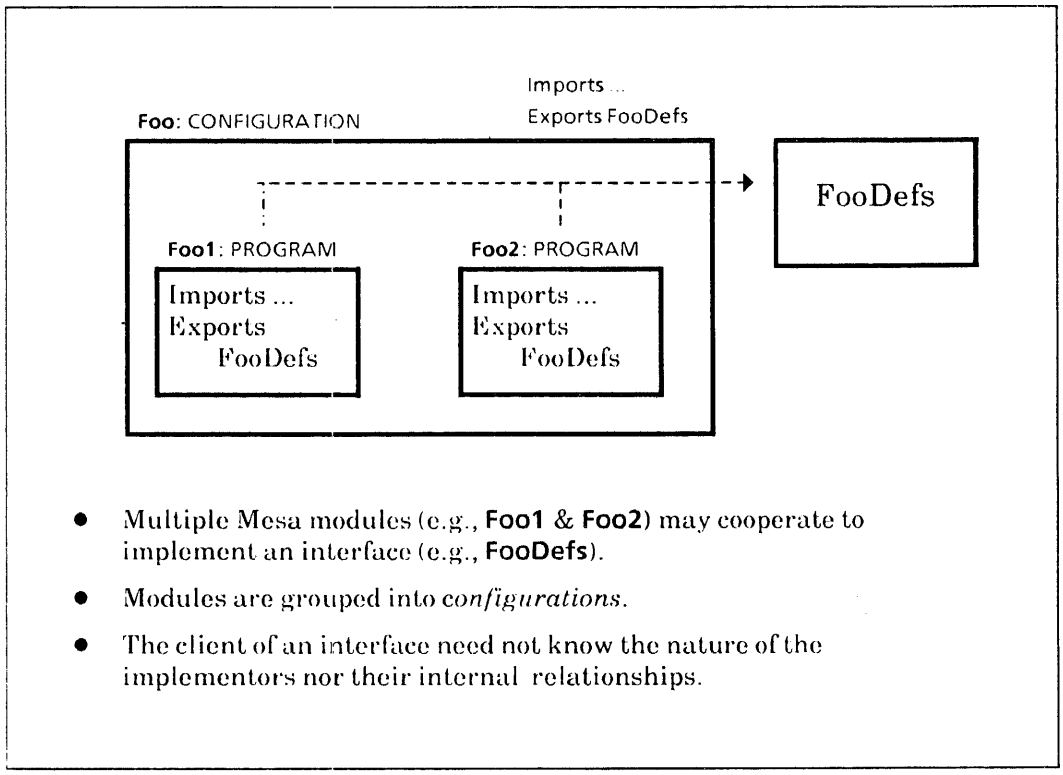
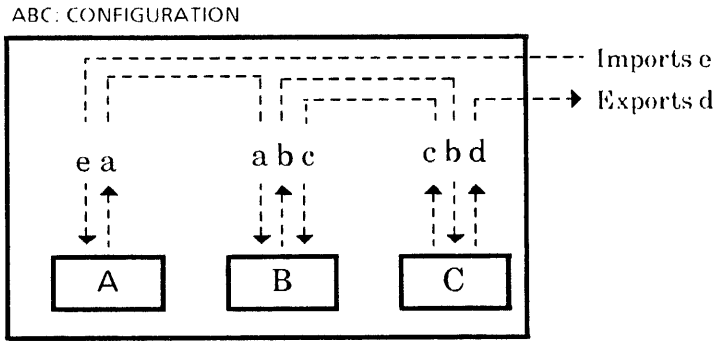


Figure 3.2 Mesa configurations
Multiple Implementors

3.1.2.1 Information visibility

If the code for an imported symbol is exported by another module in the configuration, then that symbol can be resolved within the configuration. Otherwise, the symbol must be listed as an import of the entire configuration. Similarly, a symbol may be exported by a particular module but not by the configuration as a whole. Only symbols that are listed as imports or exports of the configuration can be accessed by other modules; symbols that are resolved within the configuration are said to be private to the configuration. The programmer can thus control the visibility of variables and routines, as illustrated in Figure 3.3.

Mesa also has **PUBLIC, PRIVATE**, and **READ-ONLY** attributes that can be used to control access to specific identifiers. By default, identifiers are **PUBLIC** in definitions modules and **PRIVATE** otherwise. Any identifer with the attribute **PRIVATE** is visible only in the module in which it is declared and in any module implementing that module. Subject to the ordinary rules of scope, an identifer with the attribute **PUBLIC** is visible in any module that includes and opens the module in which it is declared. The **PUBLIC** attribute can be restricted by specifying the additional attribute **READ-ONLY**.



- A configuration is a collection of source modules and a *configuration description.*
- Mesa configurations *import* and *export* interfaces; they may in turn be bound to other programs and/or other configurations.
- a, b, c, d, and e are Mesa interfaces.
- This simple configuration named ABC exports d and imports e from the configuration of programs A, B, and C. Interfaces a, b, and c are private to (resolved within) the configuration.
- Compiler and binder type check the interface between A, B, and C.

Figure 3.3 Mesa configurations hiding interfaces

3.1.2.2 Inter-modular type-checking

When a **DEFINITIONS** (interface) module is compiled, the compiler generates a unique internal name for the interface (essentially a time stamp concatenated to the name of the file). The compiler places this unique name in the object file generated for any module compiled using that interface. An object file contains the executable code for the module, a symbol table, and a *binary configuration description.* As its name implies, the primary use of a binary configuration description is to describe a configuration. Hence, a bcd

- gives the internal names of the modules(s) used in the compilation
- lists the symbols available for other modules and configurations to use (exports)
- lists symbols that are used but declared elsewhere (imports)
- specifies where to find the object code and symbol table associated with the module or modules, since the code and symbols for a configuration may be scattered over many files.

Interfaces are "the same" for the purposes of binding only if they have the same internal name (version stamp) The binder checks that each interface is used in exactly the same version by every importer and exporter. Thus, the binder extends Mesa's compile-time checking to bind-time, and type-safety is guaranteed even across module boundaries.

This strategy has profound effects on the organization and management of large systems. It guarantees complete type-safety and consistency among all modules in a system communicating via a particular interface. On the other hand, accessing other modules introduces compilation order dependencies. Each module must be compiled after the modules it accesses (and recompiled if they change), since the compiler needs their symbol tables. However, modules that are not accessed by others (virtually all implementations) may be freely recompiled without invalidating previous compilation and checking of any other modules.

3.1.3 Loading and running a program

Once an object file has been created, it can be loaded and run. Loading allocates a global frame (activation record) for each module in the .bcd, allowing space for that module's variables as well as for information used by the system. The loader also resolves the relative binder-generated references into absolute references. The module body itself generally contains the code to initialize the global variables and establish any necessary invariants. It will be executed when the module is started, or upon application of one of the module's procedures, whichever comes first.

3.1.4 Processes and Monitors

The Mesa language provides efficient mechanisms for concurrent execution of multiple *processes* within a single system. In Mesa, a process is effectively a special procedure activation that executes concurrently with its caller, allowing asynchronous activities. Creation of a new process is done with the **FORK** operation, which spawns a new process parallel to the original. The result is a separate, independently executing thread of control, with its own local data (if any). A process thus has the same status as a procedure: it is a first class value that may call procedures, be assigned to a variable or an array element, passed as a parameter, and in general treated exactly like any other value.

This "process as procedure" approach also means that the method for passing parameters to a new process and retrieving the results is exactly the same as the corresponding method for procedures, and is subject to the same strict type checking. Since the Xerox Development Environment supports only one user and processes are assumed to be cooperating, no additional protection is associated with Mesa processes except that which is implicit in the type system of the language. Thus, all processes can execute in the same address space, which means that creating processes and switching between processes is inexpensive--not much more time-consuming than a procedure call. Thus Mesa processes are very "lightweight" and can be created and destroyed very frequently.

A forked process can later be re-synchronized with its caller, in which case the returning process communicates its results to the joining process, and is then deleted. In other cases, however, the role of a forked process is not to calculate a particular result, but rather to provide some continuing service. Such forked processes are "detached", and need never be synchronized with their creator. Such processes illustrate one major difference between

Mesa processes and the concept of *job* found in many other systems. A job has a well-defined beginning and end, but a process may exist for as long as the power is on.

3.1.4.1 Monitors

Multiple processes that work on different aspects of a problem need to communicate both data and control information with one another. Such cooperating processes need to interact in more complicated ways than simply forking and joining. The general philosophy of process synchronization in the XDE is that interaction among processes always reduces to carefully synchronized access to shared data.

Shared data is managed by a *monitor*. The simplest monitor is an instance of a module. In simple cases, a monitor's data comprises its global variables, protected by an implicit lock that is automatically allocated in its global frame. A **MONITOR** module differs only slightly from a standard Mesa program module.

To access a shared resource or data structure, a process calls a special kind of procedure in the monitor module, called an **ENTRY** procedure. Application of one of a monitor's **ENTRY** procedures automatically acquires the monitor's lock (waiting if necessary), and a return releases it. The monitor lock thus ensures that at most one process is executing a monitor procedure at a time; this process is said to be in the monitor. If a process is in the monitor, any other process that calls an entry procedure will be delayed. Thus, a process may be forced to wait on a queue until another process releases a monitor lock.

3.1.4.2 Condition variables

The *condition variable* facility provides additional flexibility in process synchronization. Suppose, for example, that a process only wishes to execute monitored code under certain conditions. If the process enters a monitor and finds that the condition is not satisfied, it can choose to *wait* until that condition is satisfied. The **WAIT** operation allows a process to release the monitor lock temporarily (and suspend execution) without returning. The blocked process then waits on a condition queue until some other process enters the monitor and establishes the necessary condition.

Many implementations of monitors require that a process waiting on a condition variable must run immediately when another process signals that variable, and that the signalling process runs when the waiting process has left the monitor. In Mesa, however, a process that establishes a condition for which some other process may be waiting *notifies* the corresponding condition variable. Notification is regarded as a hint to a waiting process, which then resumes execution at some convenient future time (reacquiring the monitor lock). There is no guarantee that some other process will not enter the monitor before the waiting process. Hence the waiting process must reevaluate the situation each time it resumes.

The condition variable facility also provices a *timeout* option. A condition variable has an associated time-out interval. A process that has been waiting for the specified interval will resume regardless of whether the condition has been notified. A process may also be *aborted* at any time, with the result that a waiting process may resume immediately.

3.1.4.3 Monitored objects

Programmers often wish to have a collection of shared data objects, each one representing an instance of some abstract object such as a file, a storage volume, or a database view, and to be able to add and delete from the collection dynamically. In a sequential program this is done with standard techniques for allocating and freeing storage. In a concurrent program, however, provision must also be made for serializing access to each object.

Mesa therefore provides a type constructor called a monitored record, which is exactly like an ordinary record except that it includes a monitor lock and is intended to be used as the protected data of a monitor. Such an object is declared as a **MONITORED RECORD**, and the lock is associated with the record itself, rather than with the module's global frame.

3.2 Pilot, the operating system

The Xerox Development Environment operating system, called Pilot, is not a typical, general purpose operating system. Instead, it is a nucleus of software that is viewed as an entity by applications programs. In particular, Pilot defines a "Basic Machine" that is an abstraction of the physical resources provided by the hardware. The purpose of this Basic Machine is to define a standard interface that is relatively independent of the size, speed, particular model, and configuration upon which it is operating. It thus provides a uniform environment for program design, and insulates clients as much as possible from variations in hardware configuration from site to site and from time to time.

Because Pilot is designed around the notion that its clients are a cooperative system, it is far more tolerant and permissive than most operating systems, and 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. Thus, Pilot is essentially co-equal with the applications programs that it supports.

The major features supported by Pilot are a hierarchical virtual memory mapped to a large file space, streams, and support for the concurrency features of the Mesa language. Pilot also provides substantial support for the network communications that are at the heart of the Xerox distributed environment. Pilot omits certain functions that have been integrated into some other operating systems, such as character-string naming and user-command interpretation; Pilot assumes that such facilities are provided as needed by the Xerox Development Environment software.

3.2.1 Files and volumes

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. The Pilot file system can support up to 2^{64} separate files of up to 2^{23} pages each, a page being 512 bytes. The space of files is "flat", in the sense that files have no recognized relationships among them (no directory hierarchy.) Pilot expects the XDE file system to super-impose further structure on files and volumes as necessary: the emphasis at the Pilot level is on simple, powerful primitives for accessing information.

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. A logical volume is the unit of storage for client files and the system data structures for manipulating them. A logical volume is either a physical volume or a subset of a physical volume or a collection of subsets of physical volumes. For instance, a large logical volume could span several physical volumes, or several logical volumes could be put on the same physical volume. A logical volume becomes logically available or unavailable as a unit and contains only complete files (i.e., files cannot span logical volumes).

Pilot supports access to files on local volumes. Each existing file is uniquely defined within that volume. Thus, a file is uniquely identified by its file ID and the ID of the containing volume.

3.2.2 Virtual memory and Pilot

The XDE processor defines a simple linear virtual memory of up to 2^{32} 16-bit words, organized into 256-word pages. All computations on the machine, including Pilot itself, run in the same address space. Pilot structures this homogenous address space into contiguous runs of pages called *spaces*. Above Pilot, the XDE software super-imposes still further structure upon the contents of spaces, casting them as client-defined data structures within the Mesa language.

Client programs allocate a region of virtual memory by creating a space of appropriate size: a new space is always created as a subspace of an existing space. To associate information with a region of virtual memory, a client program maps a space to a region of some file. 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, as described in the next paragraph. Pilot also provides operations to remove the mapping when it is no longer required.

When 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 either under client program control or on demand. A client program can inform Pilot that certain intervals of virtual memory will be needed in the immediate future and that swapping should be initiated as soon as possible, or that an interval is not currently needed and should be swapped out, or that an interval will never be needed again. 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 that page and any adjoining swapped-out pages of the containing swap unit.

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. Pilot further improves performance by attempting to allocate the pages of a file contiguously so that an interval can be swapped in a single I/O operation.

3.2.3 Streams and input/output devices

A *stream* is a temporary type of object that exists as an interim agent for accessing other objects. A stream can be used to access an object on a device, a file, another process, or a remote system element. For example, the simplest type of I/O to a keyboard may involve

reading bytes from a stream assigned to the keyboard. (The Pilot stream facility is thus similar in spirit to UNIX streams.) Streams provide reliable communication between any two machines on an internet. Most I/O devices are also made directly available to clients through low-level procedural interfaces in order to provide maximum flexibility to client programs.

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. See the glossary for definitions of transducer, filter, and pipeline.

3.2.4 Communications

As discussed in section 3.1.4, the monitor facility enables communication among tightly coupled processes executing in the same system element. Communication among client processes in different machines, however, is performed by Pilot's packet transportation facilities. The basic communication function provided by Pilot is the transport of *datagrams*. A datagram is a fundamental packet with internet-wide source and destination addresses that allow the datagram to be sent between any two nodes in the internet.

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

3.2.5 Mesa Language support

The implementation of processes and monitors is split among the Mesa compiler, the operating system, and the underlying machine. The compiler recognizes the various syntactic constructs and generates appropriate code. Pilot implements the less heavily used operations, such as process creation and destruction. The machine directly implements the more heavily used features, such as process scheduling.

3.2.6 World swapping

Pilot also provides a world-swap facility for debugging. Executing a world-swap to the debugger saves the contents of memory and the total machine state in a file, and then loads a snapshot of the debugger into memory and starts it running. Execution can be resumed by doing a second world swap back to the system being debugged. The state is saved with sufficient care so that the program being debugged will be unaffected by the visit to the debugger. The world-swap approach to debugging yields strong isolation between the debugger and the program being debugged. Not only the contents of main memory, but the version of Pilot, the accessible volumes, and even the microcode can be different in the two worlds.

3.3 The XDE processor

The XDE processor architecture is unusual in that the hardware architecture was designed for the software architecture, rather than the other way around. In the XDE, the architecture design is separated from any particular implementation. Thus, the term

architecture means the characteristics of the processor as seen by the programmer writing instructions to be executed by the machine; the term *processor* refers to a particular implementation of the architecture (or all such implementations). Thus, many different combinations of hardware and microcode might be used to implement an XDE processor.

Like other aspects of the Xerox Development Environment, the architecture is designed for cooperating, not competing, processes. There is no "supervisor mode", nor are there any "privileged" instructions. The primary goals of the architecture are to enable the efficient implementation of a modular, high level programming language, such as Mesa, and to provide a very compact representation of programs and data so that large, complex systems can run efficiently with relatively small amounts of primary memory. The emphasis is on efficiency of the object code and on a good match between the semantics of the language and the capabilities of the processor.

3.3.1 Compact program representation

The Mesa instruction set is designed for a compact representation of programs. The general idea is to introduce special instructions into the instruction set so that frequent operations can be represented in a minimum number of bytes. Instructions are variable length with the most frequently used operations and operands encoded in a single byte; less frequently used combinations are encoded in two bytes, and so on. For example, operations such as reading a word from a record given a pointer to the record in a local variable, storing values through pointers, and procedure calls are encoded in a single byte. The guiding principle of the Mesa instruction set is "if an operation, even a complex one involving indirection and indexing, occurs frequently in programs, then it should be a single instruction or family of instructions."

Similarly, frequently referenced variables are stored together. Most operands are addressed with small offsets from local or global frame pointers or from variable's pointers stored in the local or global frame. Using small offsets means that instructions can be smaller because fewer bits are needed to record the offset.

3.3.2 Stack machine

The Mesa processor is a stack machine; it has no general purpose registers. The evaluation stack is used as the destination for load instructions, the source for store instructions, and as both the source and destination for arithmetic instructions; it is also used for passing parameters to procedures.

The primary motivation for the stack architecture is not to simplify code generation, but to achieve compact program representation. Instructions can be smaller because they need not specify all operand locations: since the stack is assumed as the source and/or destination of one or more operands. Another motivation for the stack is to minimize the register saving and restoring required during procedure calls.

3.3.3 Control transfers

The Mesa architecture is designed to support modular programming, and therefore optimizes transfers of control between modules. The Mesa processor implements all control transfers with a single efficient primitive, called XFER, which is a generalization of the notion of a procedure or subroutine call. All of the standard procedure calling

conventions, and all transfers of control between contexts (procedure call and return, nested procedure calls, coroutine transfers, traps, and prcoess switches) are implemented using the XFER primitive.

3.3.4 Process mechanism

The Mesa processor provides for the simultaneous execution of up to one thousand asynchronous preemptable processes on a single processor. The process implementation is based on queues of small objects called Process State Blocks (PSB's), each representing a single process. When a process is not running, its PSB records the state associated with the process. If the process was preempted, its evaluation stack is also saved in an auxiliary data structure; the evaluation stack is known to be empty when a process stops running voluntarily (by waiting on a condition or blocking on a monitor).

Each PSB is a member of exactly one process queue. There are four kinds of queues: *condition variable, ready, monitor lock,* and *fault.* The ready queue contains all processses that are ready to run; that is, not blocked on a monitor, waiting on a condition variable, or faulted (e.g. suspended by a page fault). The process at the head of the ready queue is the one currently being executed.

The primary effect of the process instrucions is to move PSB's back and forth between the ready queue and a monitor or condition queue. A process moves from the ready to a monitor queue when it attempts to enter a locked monitor; it moves from the monitor queue to the ready queue when the monitor is unlocked (by some other process). Similarly, a process moves from the ready queue to a condition queue when it waits on a condition variable, and it moves back to the ready queue when the condition variable is notified, or when the process has timed out.

Each time a process is requeued, the scheduler is invoked; it saves the state of the current process in the process's PSB, loads the state of the highest priority ready process, and continues execution. To simplify the task of choosing the highest priority task from a queue, all queues are kept sorted by priority.

3.3.5 Virtual memory and the processor

Virtual addresses are mapped into real addresses by the processor. The mapping mechanism can be modeled as an array of real page numbers indexed by virtual page numbers.

Virtual memory is addressed by either long (32 bit) pointers containing a full virtual address or by short (16 bit) pointers containing an offset from an implicit 64K word aligned base address. There are several uses of short pointers defined by the architecture:

- the first 64K words of virtual memory are reserved for booting data and communication with I/O devices. Virtual addresses known to be in this range are passed to I/O devices as short pointers with an implicit base of zero.
- the second 64K of virtual memory contains data structures relating to processes. Pointers to data structures in this area are stored as short pointers with an implicit base of 64K.

Code may be placed anywhere in virtual memory, although in general it is not located within the reserved regions mentioned above. A code segment contains read only instructions and constants for the procedures that comprise a Mesa module; it is never modified during normal execution and is usually write-protected. A code segment is relocatable without modification; no information in a code segment depends on its location in virtual memory.

Any region of the virtual memory can contain additional dynamically allocated user data; it is managed by the programmer and referenced indirectly using long or short pointers.

An important interval of virtual memory recognized by the processor and the Mesa system is the main data space (MDS). This is a 64k block of virtual memory, any part of which may be addressed by a short pointer. An MDS contains the global data of program modules and the local data of procedure invocations. Of course, a program or procedure can refer to arbitrary data via a pointer in its local or global data.. Each process is associated with one and only one MDS. Although the processor supports multiple coexisting MDS's, Pilot does not. Thus, any Pilot-based system has only one MDS, which is shared by all of the system's processes.

The data associated with a Mesa program is allocated in a main data space in the form of local and global *frames*. A *global frame* contains the data common to all procedures in the module. The global frame is allocated when a module is loaded and freed when the module is destroyed. A *local frame* contains data declared within a procedure; it is allocated when the procedure is called and freed when it returns.

3.3.6 Contexts

In addition to a program's variables, there is a small amount of linkage and control information in each frame. A local frame contains a short pointer to the associated global frame and a short pointer to the local frame of its caller (the return link). A local frame also hold the program counter for a procedure whose execution has been suspended. Each global frame contains a long pointer to the code segment of the module.

To speed access to code and data, the processor contains registers that hold the local and global frame adddreses and the code base and program counter for the currently executing procedure; these are collectively called a context. When a procedure is suspended, the single sixteen bit value which is the MDS relative pointer to its local frame is sufficient to reestablish this complete context by fetching global frame and program counter from the loccal frame and code base from the global frame.

Glossary

Abstract machine: An *abstract machine* is a set of functions, provided by hardware or software, that forms the underpinnings of a system sitting above. Pilot, for example, is an abstract machine that runs on a variety of machines.

Abort: To *abort* is to terminate a process abnormally, such as by using the **ABORT** key.

Accelerator: An *accelerator* is an easier or faster way of doing a common operation. Clicking **ADJUST** in the center third of the name stripe, for example, is an accelerator for sizing a window (rather than bringing up the window menu and selecting "Size".)

Active window: *An active window* is a window that is ready for interaction with the user and is displayed full size. (Compare **Tiny window**, **Inactive window**.)

Address Fault: An *address fault* occurs when an attempt is made to reference an illegal address.

Adjective: An *adjective* is an identifier constant from an enumerated type, used to select one of the alternatives in a variant record. (See **Tag**.)

ADJUST: *ADJUST* is the right mouse button, generally used to extend selections and for accelerators.

ALT B: *ALT B* is a boot button used to do alternate booting, such as booting from another device.

Argument: An *argument* to a procedure or command is a piece of data upon which the operation is performed. For example, the argument to a **MOVE** command is the video-inverted text to be moved.

Asynchronous call: An *asynchronous call* is a procedure call that initiates an operation, but returns control to its caller without waiting for the operation to complete.

Atom: An *atom* is a Mesa primitive providing a unique identifier in a global naming space. An atom has an associated property list.

Authenticate: To *authenticate* is to establish that a user or client is who he, she, or it claims to be, such as by checking the user or client's credentials. (See **Credentials**.)

Background process: A *background process* is a process that receives machine resources only if higher priority processes are idle or blocked.

Backing store: *Backing store* is a sequence of pages from a file to which a part of virtual memory is mapped. Any part of virtual memory with useful contents is mapped to a backing store. (See **Map**.)

BCD: A binary configuration description (*BCD*) is a compiled and possibly bound Mesa module, sometimes called an object file. (See **Configuration description**.)

Bind: To *bind* is to combine object modules into one executable unit (called a configuration) by resolving intermodule references.

Bitmap: A *bitmap* is 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 bitmap to a displayed image.

Boot: To *boot* is to load and start a system on a machine whose main memory has essentially undefined contents. The Dandelion can be booted by pressing the **B RESET** boot button. ("Boot" is short for "bootstrap", which is in turn short for "bootstrap load".)

Boot button: A *boot button* is a maintenance panel button used to boot the processor. The Dandelion has two boot buttons, labelled **B RESET** and **ALT B**.

Boot file: A *boot file* is a file that contains a bootable system, such as CoPilot, that receives control when the volume is booted.

Broadcast network: A *braodcast* network is one in which a packet can be sent to every host on the network, rather than to just one specific host. (See **multi-cast**.)

CALL DEBUG: *CALL DEBUG* is the action of pressing **SHIFT-ABORT** together, which transfers control to the debugger.

Call Stack: The *call stack* is 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.

Caret: The *caret* is a blinking pointer that indicates the type-in point.

Catch Phrase: A *catch phrase* is a Mesa construct that establishes code to catch one or more signals.

Channel: A *channel* is a low-level procedural interface for accessing and driving I/O devices.

Chord: To *chord* keys or buttons is to push them down at the same time, as when chording the mouse buttons.

Clearinghouse: A *clearinghouse* is a server for locating named objects in a distributed environment.

Click: To *click* a mouse button is to press down on it and let it up.

Client: A *client* is a program (as opposed to a person) that uses the services of another program or system. (See **User**.)

CoCoPilot: *CoCoPilot* is the name usually given to the debuggerDebugger volume used to debug programs running in the CoPilot volume.

Command Central: *Command Central* is a tool for compiling and binding programs on a development volume and running them on a client volume.

Command file: A *command file* is a file containing commands, especially Executive commands.

Compile: To *compile* is to translate a source file into an object file (BCD).

Condition variable: A *condition variable* is a Mesa construct by which processes wait for or provide notification of an event. A condition variable is associated with a monitor.

Configuration description: A *configuration description* (*config* for short) is a C/Mesa source file that tells the Binder how to combine modules into a configuration. A *configuration* file is the bound code of one or (usually) more modules.

Context: See *Debugger context*.

Continue: To *continue* a signal is to resume program execution at the statement following the one to which the catch phrase belongs. Thus, control is resumed in the procedure where the signal was caught, not the procedure that raised the signal.

CoPilot: *CoPilot* is the name of the debugger volume used to debug programs in Tajo and other normal volumes. The boot file that

contains the debugger, used on both the CoPilot and CoCoPilot volumes, is also called CoPilot.

Courier: *Courier* is the Network Systems remote procedure call facility. A remote procedure call causes a procedure to be executed in another machine over a network.

Create date: The *create date* is the date and time that the information contained in a particular version of a particular file was created. Since create dates are accurate to the nearest second, the pair <file name, file version's create date> serves as a unique identifier for the contents of a file.

Credentials: *Credentials* are the identification, such as name and password, presented by a client to a service for authentication.

Critical section: A *critical section* is a portion of a program in which only one process may be executing at a time. In Mesa, access to critical sections is arbitrated by monitors.

Current selection: See **Selection**.

Cursor: The *cursor* is an icon that tracks the mouse position: moving the mouse moves the cursor. The system may change the cursor shape to provide feedback about what it is doing.

Dandelion: *Dandelion* is the internal name for the processor that supports both the Xerox Development Environment and the Office System products.

Dangling Pointer: A *dangling pointer* is a pointer to an invalid memory location, usually the result of deallocating storage while a pointer to it remains.

Deactivate: To *deactivate* is to make a tool inactive, removing all windows associated with the tool from the display and discarding the state of the tool.

Debugger context: A *context* in the debugger is a referencing environment that determines the meaning of symbols. The current context identifies one of the executing processes (within a particular module within a particular configuration) that the debugger will use in interpreting other commands. For example, the current context determines which variables in which procedure invocations to use in evaluating an expression.

debugger volume: A *debugger volume* is a logical volume that contains a debugger and is used to debug normal volumes. (See **normal volume, debuggerDebugger volume**.)

debuggerDebugger volume: A *debugger-Debugger volume* is a logical volume that contains a debugger and is used to debug debugger volumes.

Dereference: To *dereference* a pointer is to follow the pointer through one level of indirection toward the value it is referencing.

Device: A *device* is a peripheral unit (almost always hardware) that is separately accessible through its own channel.

Device driver: A *device driver* is a program that translates channel requests into physical device actions.

Directory: A *directory* is a named subdivision of a logical volume. A directory can in turn be divided into subdirectories. The top-level directory on a volume has the same name as the volume.

Discrimination: A *discrimination* statement provides access to the fields in the variant part of a variant record, based on the value of the tag.

Disk page: A *disk page* is a contiguous 256-word region of disk storage.

Dynamic allocation: *Dynamic allocation* acquires storage during program execution.

Encapsulate: To *encapsulate* a packet is to transform it in whatever fashion is necessary to allow the packet to pass as data. Generally, encapsulation consists of mechanism such as adding headers and trailers.

Error: An *error* is a Mesa language construct similar to a signal, except that a signal can return to where it was raised (like a procedure), whereas an error cannot.

Ethernet: The *Ethernet* is a communications system for carrying digital data among locally distributed computer systems. The Ethernet is implemented as a 10 megabit/second multi-access packet-switched network.

Exception: An *exception* is an unusual event that programs must be prepared to handle, such as I/O error. In Mesa, exceptions are associated with signals. (See **Signal**.)

Executive: The *Executive* is a tool with a simple teletype interface for loading and running Mesa programs. Some commands are built in to the Executive.

Export: To *export* is to implement all or part of an interface for use by other modules. (See **Import, Interface**.)

Face: A *face* is a Mesa interface that embodies part of the XDE abstract machine.

File: A *file* is 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: The *file extension* is the (possibly null) portion of a file name that follows a period. By convention, some extensions indicate the format of the data in the file (although not all tools use default extensions consistently). Some common extensions are:

archiveBcd	Mesa object program module
bcd	Mesa object program module
boot	boot file
cm	command file
config	a C/Mesa source file (configuration description file)
doc	documentation file
errlog	error message file
ip	Interpress format
log	history of program actions
mesa	Mesa source module
symbols	Mesa symbol table in binary format (for debugging)
tip	TIP tables

File handle: A *file handle* is a data structure that identifies a file being accessed.

File service: The *file service* is a set of network facilities that provide file storage and retrieval. A machine implementing this service is called a file server.

File Tool: The *file tool* is a tool that allows the user to store, retrieve, delete, and list files on remote file services.

File type: A *file type* is a file attribute provided by Pilot for the use of higher level software.

File window: A *file window* is a window whose main subwindow is a text subwindow for displaying and editing the contents of a file. A contiguous group of pages within a file into which a map unit is mapped is also called a file window.

Filter: A *filter* is a software entity that implements a stream for transforming, buffering, and manipulating data.

Font: A *font* is a set of characters of similar size and style. Fonts come in different families (such as Classic or Gothic), different sizes (such as 10 point or 14 point), and different styles (such as plain, bold, or italic). This sentence is in Classic 10 plain font.

Formatter: The *Formatter* is a tool that transforms Mesa source files into a standard format.

Form subwindow: A *form subwindow* is a system-provided subwindow type that supports invoking commands and displaying or changing the values of data.

Frame: A *frame* is a data structure allocated for the variables and internal data structures of an executing module or procedure. Module frames are called *global frames*, and procedure frames are called *local frames*. Since Mesa supports recursion, there may be several frames for a given procedure.

Frame pack: A *frame pack* is a swap unit produced by the Packager that contains the global frames for a collection of modules.

Frozen: A system is *frozen* if no program can respond to input from either the mouse or keyboard, including a panic interrput. (Compare **Wedged**.)

Gateway: A *gateway* is a processor serving as a forwarding link between Ethernets. (See **Router**.)

Germ: The *germ* is the Pilot program that loads a boot file into memory and starts it executing. The germ also creates outload files and implements communication with remote debuggers. The germ is so named because it is the first program executed when a boot button is pushed.

Head: A *head* is an implementation of a face for some processor or device. A collection of heads provides a processor-independent environment in which Pilot and its clients execute.

Heap: A *heap* is a system-designated area of virtual memory used for dynamic allocation of storage. Heaps, which provide more automatic management of storage than zones, support the Mesa language operators **NEW** and **FREE**, which allocate and deallocate storage dynamically.

Herald Window: The *herald window* is a tool (usually a wide, short window at the top of the screen) that displays information about the state of the environment, has a menu to boot logical volumes, and allows tools to display messages.

Hint: A *hint* is information that is usually accurate and is easy for a program to use. A program can detect when a hint is inaccurate and find the truth in some other (usually less efficient) way.

Icon: An *icon* is a small picture on the display representing some entity.

Implementation module: An *implementation* or **PROGRAM** *module* is a program that codes (*implements*) and makes available to clients (*exports*) items in an interface. One implementation module can export all or part of one or several interfaces, and an interface can be jointly implemented by several implementation modules.

Import: To *import* is to make accessible to one module the procedures and variables exported by other modules. (See **Exports**.)

Inactive window: An *inactive window* is one that is not represented on the display and that retains no state. (Compare **Active window**, **Tiny window**.)

Input Focus: The *input focus* is the window to which keyboard commands and typed characters are sent. The input focus contains the type-in point.

Interface: An *interface* is 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: An *interface* or **DEFINITIONS** *module* defines types, variables, constants, procedures, and signals, thus specifying the services to be provided by its implementation modules.

Interlisp: *Interlisp* is an interactive version of LISP with a large library of facilities.

Internet: An *internet* is a collection of networks mutually accessible via internet routing services.

Interpress: *Interpress* is a print file format standard.

Lister: The *Lister* produces listings of information in object files, such as dates of the interface modules used and cross references of procedure calls.

Log file: A *log file* is a file containing a history of program actions. For example, `compiler.log` contains summary statistics for each source file compiled by the most recent invocation of the compiler.

Logical volume: A *logical volume* is a partition of storage for client files, including system data structures for manipulating those files. A physical volume is divided into one or more logical volumes. Each logical volume is largely protected from actions in other logical volumes.

Loophole: *Loophole* is a Mesa operator that coerces a value of one type into another type, thus circumventing Mesa's strong typing. For example, loopholing is often used in the debugger to translate a **CARDINAL** (the internal

representation of a signal) into the name of the signal.

Machine: A *machine* is a hardware configuration consisting of a processor, main memory, and peripheral devices. Workstations and servers are machines.

Main data space: The *main data space (MDS)* is a subspace of virtual memory that provides the local execution environment for Mesa programs and holds the implicit Mesa data structures. The MDS can contain up to 64K words. Thus, only short (16-bit) pointers are needed to address any part of the MDS.

Maintenance panel codes: *Maintenance panel codes* (MP codes) are three or four-digit status and error codes that indicate the current processor state.

Maintenance release: A *maintenance release* is a re-release of a system to correct flaws that cannot be worked around. Maintenance releases do not introduce new features.

Map: To *map* is 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.

Map unit: A *map unit* is a contiguous group of virtual memory pages that is the principle unit for allocating, mapping, and swapping virtual memory.

Menu: A *menu* is a list of available commands or data chosen by mouse selection. More than one menu may be associated with a tool window or subwindow or with the unused portion of the display.

Mesa: The *Mesa* language is a Pascal-like, strongly typed, system programming language that forms the basis of the Xerox Development Environment.

Message subwindow: A *message subwindow* is a system-provided subwindow type for posting messages (including errors).

MLM: The *Mesa Language Manual* is a reference manual for the Mesa programming language.

Mode: A *mode* is a special state of a system in which user actions have special meaning.

Modeless: A *modeless* user interface is one that is free of modes. In such an interface, pressing a particular key always has essentially the same effect.

Module: A *module* is a Mesa program. A *source module* is a text file that can be compiled into an *object module*.

Monitor: A *monitor* module is a Mesa module that controls access to shared data, thus synchronizing interactions among processes.

Monitor invariant: A *monitor invariant* is a logical assertion about the state of monitored data whenever the monitor is unlocked (i.e., exited). Every monitor has a monitor invariant.

Monitor lock: A *monitor lock* is essentially a hidden data item associated with each monitored record or program that indicates when a process has entered and not yet exited a critical section.

Mouse: The *mouse* is a pointing device that allows the user to direct the attention of the machine to a particular point on the display. A mouse usually has two buttons, **POINT** and **ADJUST**. (See **POINT, ADJUST**.)

Mouse-ahead: Analogous to type-ahead, *mouse-ahead* is mouse clicks made before a program has asked for them.

Movable boundary: A *movable boundary* is a horizontal line with a small box on its right end that divides a window into subwindows or splits a text subwindow. A movable boundary is used to change the relative heights of adjacent subwindows.

MPM: The *Mesa Programmer's Manual* describes the interfaces that provide the framework and run-time system for writing Mesa programs in the Xerox Development Environment.

Multi-cast network; A *multi-cast* network is local network that has the capacity to transmit a packet to more than one host. *Broadcasting* is

thus a special case of multi-casting that allows a packet to be transmitted to all hosts.

Name lookup: *Name lookup* is the process of mapping a character string to a network address.

Name stripe: The *name stripe* is a rectangular region at the top of a window. It is usually black, with the window's name and other information in white.

Network: A *network* is a communication medium, such as an Ethernet, known to routers by a unique network number.

Network address: A *network address* 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 machine, independent of which network it is on. A socket number identifies a particular socket on that host. (See **Socket**.)

Network stream: A *network stream* is a stream representing a connection over a network between two processes, often on different machines.

Node: A *storage node*, or *node* for short, is a block of allocated storage, often with a record structure.

Normal volume: A *normal volume* is a logical volume used to run client programs. (See **debugger volume**, **debuggerDebugger volume**.)

Notifier: The *Notifier* process in Tajo handles user actions, informing each tool of each user action directed to it. Because tools perform their work in the Notifier process, further user input is not acted on until a tool operation is finished.

NS: Network Systems (*NS*) are the Xerox standard protocols for using the Ethernet.

Object file: An *object file* is a BCD.

Othello: *Othello* is a utility for managing Pilot volumes, including initializing physical and logical volumes, installing and invoking boot files, and scavenging logical volumes.

Outload file: An *outload file* is 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**.)

Package: To *package* is to group components of modules together into swap units to try to improve use of real memory.

Packet: An NS *packet* is the unit of information in the internet. A packet consists of a header and data, and has a maximum length of 576 bytes. The information in the header is specified by the Internet Datagram Protocol.

Page: A *page* is a block of 256 words of information in either virtual memory or a file. The page is the basic addressable unit of a file.

Panic interrupt: A *panic interrupt* in the Xerox Development Environment is caused by hitting **STOP** while holding down both **SHIFT** keys.

Path name: The *path name* is the complete name of a file, including the file server or workstation and directory or subdirectory on which it is stored. A path name is usually denoted by a machine name in square brackets followed by a directory name in angle brackets, optionally followed by one or more subdirectory names separated with right angle brackets, followed by the file name itself, such as [Iris]<Mesa>Doc>Compiler.doc.

Physical volume: A *physical volume* is the basic unit available for random access file page storage. A physical volume corresponds to a storage device, typically a disk.

Pilot: *Pilot* is the operating system for the Xerox Development Environment. Pilot provides a single-user, single-language environment including virtual memory, a large flat file system, network communication facilities, and Mesa run-time support (including concurrency facilities).

Pilot kernel: The *Pilot kernel* comprises the basic facilities of Pilot.

Pipeline: A *pipeline* is a sequence of concatenated filters that perform a series of

transformations on the contents and properties of a stream.

POINT: *POINT* is the left mouse button, generally used to identify data and to invoke commands.

Pointer: A *pointer* is a data item containing the location of a value. The Mesa language has pointer types.

Point-to-point: *Point-to-point* connectivity menas that a channel has exactly two hosts attached to it; a host can thus send a packet to any other single host.

PPM: The *Pilot Programmer's Manual* describes the visible structure and interfaces of Pilot.

Print service: A *print service* provides printing facilities, usually for files formatted in Interpress format.

Process: A *process* is effectively a procedure activation that runs concurrently with its caller, allowing asychronous activities.

Processor: A *processor* is a computing engine (including its memory) in a workstation or server.

Raise: To *raise* a signal is to instruct the Signaller to look on the call stack for the most recently invoked procedure with a catch phrase for that signal. If none is found, an uncaught signal occurs.

Real estate: *Real estate* is any part or all of the display screen.

Real memory: *Real memory* is the physical memory that holds software and data during processing (as opposed to *secondary* or *virtual memory*).

Reject: A catch phrase *rejects* a signal when it is not prepared to resolve it. A catch phrase rejects a signal either by explicitly placing a **REJECT** statement in the code or by not specifying how to resolve the signal.

Release: A *release* is an official, consistent version of software produced and maintained by its developers.

Resume: To *resume* a signal is to return program control (and possibly values) to the statement immediately following the one that raised the signal. An **ERROR** cannot be resumed.

Retry: To *retry* a signal is to tell the Signaller to re-execute the statement containing the catch phrase.

Router: A *router* is a software package that sends packets between sockets. The path chosen by a router includes intermediate stops if the destination socket is on another network. A router that sends packets between networks is called an *internet router*.

RS-232-C: *RS-232-C* is a standard established by the Electronic Industries Association for serial binary data interchange between a machine and data communication equipment. An RS-232-C controller connects a machine to a modem, allowing data to be sent across telephone lines.

Scavenge: To *scavenge* is to check for damaged file structures and to attempt to repair them.

Scroll: To *scroll* is to reposition the data visible in a subwindow as though it were part of a long, continuous sheet of paper. Scrolling up, for example, moves the data near the bottom of the window toward the top.

Scrollbar: A *scrollbar* is a tall, narrow rectangle near the left border of a subwindow, used in scrolling and thumbing.

Search path: The *search path* is a sequence of directories (with subdirectories) used as prefixes to look up file names that are not fully specified; i.e., that do not start with a directory name.

Selection: The *selection* is a text string or icon that the user has caused to be highlighted. Many actions operate on the current selection, which need not be in the window associated with the action.

Server: A *server* is a machine dedicated to performing one or more services.

Service: A *service* is a related set of facilities provided for general use, such as a print service or file service.

Signal: A *signal* is 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 *Signaller* is 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.

Size: To *size* a window is to switch its state either from active to tiny or vice versa. (See **Window state**.)

Smalltalk: *Smalltalk* is an object-oriented programming language (and its integrated programming system) developed by Xerox.

Snarf: To *snarf* is to copy files between logical volumes, especially from the CoPilot volume.

Socket: A *socket* is a source or destination of packets on a given machine. A socket is uniquely identified by a 16-bit socket number. Several streams of packets may share a single socket. A socket is accessed through a channel interface and is thus a logical input/output device. The Clearinghouse and the time server, for example, each has its own socket.

Source module: There are three kinds of source modules: PROGRAM, MONITOR, and DEFINITIONS. (See **Module**.)

Space: *Space* is the Pilot interface for managing virtual memory. Space often refers more generally to virtual memory.

Storage Leak: A *storage leak* occurs when a program neglects to free all the storage nodes it has allocated, thus reducing the total amount of space available for the system.

Stream: A *stream* is an abstraction for device- and format-independent sequential access to a collection of data. Some streams also provide random access to the data. A stream is a sequence of bytes, possibly marked by attention flags and possibly partitioned into identifiable subsequences.

Stream component manager: A *stream component manager* is the software entity that implements a stream component–a transducer, filter, or pipeline.

Stream Handle: A *stream handle* is a pointer to a stream object, which identifies the particular stream being accessed and contains the data and procedures for operations on the stream.

String: A *string* is conceptually a sequence of characters, such as "that" A string is represented in Mesa as a pointer to a record containing a sequence of characters, the current length, and the current maximum length.

Stub: A *stub* is a program that implements a Mesa interface in terms of Courier calls to a remote server or workstation.

Subdirectory: A file directory can be divided into a hierarchical collection of *subdirectories*. Subdirectory names are listed from the root of the tree down to the leaves, separated by ">". (See **Path name**.)

Subwindow: A window is often composed of one or more rectangular *subwindows*. The Xerox Development Environment provides several standard subwindow types, including form subwindows and text subwindows.

Swap: To *swap* is 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: A *swap unit* is a portion of a space to be swapped. Proper choice of the size of swap units can improve use of real memory and reduce disk overhead.

Swat: To *swat* is to strike CALL-DEBUG to invoke the debugger.

Switch: A *switch* is a modifier to a command or subcommand, often preceded by a "/".

Symbiote: A *symbiote* is a subwindow that can be added dynamically to a subwindow in an existing tool without changing the tool or Tajo. A symbiote provides extra facilities via stick-around menu items or functions.

Synchronous call: A *synchronous call* is a procedure call that returns control only after the operation completes.

Tag: The *tag* is a field of a variant record whose value selects one of the alternatives of the variant part by matching one of the adjectives.

Tajo: *Tajo* is the user interface part of the Xerox Development Environment. The main client volume and its boot file are also often called Tajo.

Teledebug: To *teledebug* is to debug remotely, that is, to debug one machine from another over the internet.

Text subwindow: A *text subwindow* is a system-provided subwindow type with text display and editing capabilities.

Thumb: To *thumb* is to position the data in a file (usually text) to an arbitrary position for viewing on a display. The "thumb-index" in some dictionaries performs somewhat the same function: it gets you to roughly the right place quickly.

Timeout: *Timeout* is the failure to complete an operation within a specified amount of time.

Tiny window: A window is *tiny* if it is represented on the display by an icon. A tiny window is not ready for interaction with the user, but maintains the state of the tool. (Compare **Active window**, **Inactive window**.)

TIP: *Terminal Input Processor* (*TIP*) is a system for interpreting keyboard and mouse actions and turning them into sequences of commands based on TIP tables.

Tool: A *tool* is a Xerox Development Environment applications program. A tool can run in parallel with other tools, including other instances of the same tool. Tools react to prompting and seldom carry out operations when not in use. A tool usually, but not always, has an associated window.

Transducer: A *transducer* is a software entity that implements a stream, such as **MStream**, connected to a specific device or medium through a Pilot channel.

Trash bin: The *trash bin* is the conceptual container of the most recently deleted selection, which can be retrieved to a different spot or a different window.

Type-ahead: *Type-ahead* is the ability to type characters to a program before that program has asked for them.

Type-in point: The *type-in point* is the text location where typed characters are to be inserted. The type-in point is indicated by a flashing caret or box.

Uncaught signal: An *uncaught signal* occurs when no module in the call stack handles a *signal* that has arisen. If a signal is uncaught, the Signaller transfers control to the debugger.

Unwind: *Unwind* is a special signal raised by the Signaller to allow procedures about to be deleted from the call stack to do clean up (such as deallocate storage and close files). When there is an unconditional branch out of the catch phrase, the Signaller raises the unwind signal at the point where the original signal originated.

User: A *user* is a person (rather than a program) who uses the services of some program or system. (See **Client**.)

User.cm: *User.cm* is a file on the system volume used to set defaults for many of the tools in the Xerox Development Environment. This file allows users to customize their environment.

User Interface: The *user interface* is the man/machine interface. It is the manner in which information is presented to you on the display screen, and the way that you communicate using keyboard and mouse.

User profile: A *user profile* is commonly accessed global information that identifies a

user in the internet. A user profile includes name, password, and Clearinghouse domain.

Valid memory location: A location is *valid* if it is currently allocated. A location that has been freed is *invalid* and should not be referenced.

Version stamp: The *version stamp* is the date and time, accurate to the nearest second, at which a file was created. Different versions of a file are distinguished by their version stamps. Version stamps allow tools such as the binder and the debugger to ensure that proper versions of files are used.

Video-invert: To *video-invert* a region is to cause black areas of the region to become white and white areas to become black.

Virtual memory: *Virtual memory* is the large word-oriented address space of up to 2^{32} words that forms the execution environment.

Volume: See **physical volume**, **logical volume**.

Wedged: A program is *wedged* when it cannot respond to input from either the keyboard or the mouse. If all programs are wedged, the system is *frozen*. (See **Frozen**.)

Window: A *window* is a rectangular region of the display in which text and graphics can be displayed. Most tools communicate via windows.

Window state: The *state* of a window is either active, tiny, or inactive. (See **Active window**, **Tiny window**, **Inactive window**.)

Word: A *word* is the basic 16-bit unit of information manipulated by Mesa processors.

Workstation: A *workstation* is a machine connected to the network and used as a personal computer. Most Dandelions are used as workstations. (See **Server**.)

World-swap: A *world-swap* is the process of writing out the complete state of a logical volume onto a disk file and reading in a different state. CoPilot normally works by world-swaps between the debugger and the program being debugged. (See **Outload file**.)

Xerox Development Environment: The *Xerox Development Environment* is 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.

XUG: The *Xerox Development Environment User's Guide* introduces the XDE and describes how to use the tools that make up the environment.

Zoom: To *zoom* a window is to switch the size of an active window either from normal to full screen or vice-versa. Zooming a normal-sized window also puts it on top of all other windows.

Zone: A *zone* is a client-designated area of virtual memory used to allocate and free arbitrary-sized storage nodes. (See **Heap**.)

OFFICE SYSTEMS DIVISION

Reader's Feedback

Xerox's Technical Publications Departments want to provide documents that meet the needs of all our product users. Your comments help us correct and improve our publications. Please take a few minutes to respond. If you have comments on the product this document describes, contact your Xerox representative.

1. Did you find any errors in this publication? What were they? On which pages?

_____

_____

2. Were there any areas that were hard to understand because of descriptions or wording? What were they? Where?

_____

_____

3. Did this publication give you all the information you needed? If not, what was missing?

_____

_____

4. Was this manual at the right level for your needs? If not, what other types of publications do you need?

_____

_____

5. What *one thing* could we do to improve this manual for you?

_____

NAME _____ DATE _____

TITLE _____ COMPANY _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

XDE3.0-1001