

XDE User Guide

XEROX

**610E00140
December 1986**

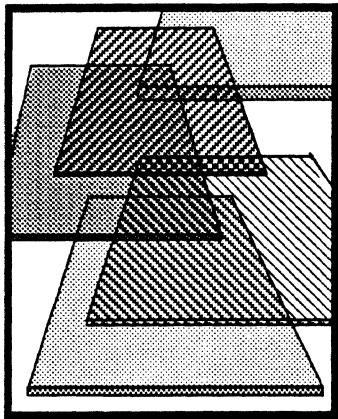
**Xerox Corporation
Information Systems Division
XDE Technical Services
475 Oakmead Parkway
Sunnyvale, California 94086**

**Copyright © 1986, Xerox Corporation. All rights reserved.
XEROX®, 8010, and XDE are trademarks of XEROX CORPORATION.**

Printed in U.S. A.

XEROX

Xerox Development Environment



XDE User's Guide

**XDE3.0-2001
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.

**Copyright © 1984 by Xerox Corporation.
All Rights Reserved.**



Preface

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

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

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

Preface



Table of contents

I General tools

I.1	System overview	I-1
I.1.1	User interface	I-1
I.1.2	Development scenario	I-1
I.1.3	Hardware	I-2
I.1.4	Software components	I-2
I.2	Definition of terms	I-3
I.3	User interface	I-5
I.3.1	Windows and subwindows	I-6
I.3.2	Text manipulation	I-10
I.3.3	Menus	I-12
I.3.4	Keyboard commands	I-17
I.4	The user command file	I-21
I.4.1	Format of the user command file	I-21
I.5	Documentation roadmap	I-24
I.5.1	XDE Concepts and Principles	I-24
I.5.2	The XDE User's Guide	I-24
I.5.3	Mesa Language Manual	I-24
I.5.4	Pilot Programmer's Manual	I-24
I.5.5	Mesa Programmer's Manual	I-25
I.5.6	Appendices	I-25
I.6	Typographical conventions	I-25
I.7	Other features, other tools	I-25

1 DMT

1.1	Files	1-1
1.2	User interface	1-1

Table of contents

2	Dictionary tool	
2.1	Files	2-1
2.2	User interface	2-1
2.3	Dictionary tool	2-1
2.3.1	Commands	2-2
2.3.2	File format	2-2
2.4	User.cm	2-2
3	Editor Symbiote	
3.1	Files	3-1
3.2	User Interface	3-1
3.2.1	Editor menu	3-1
3.3	Search and pattern matching	3-5
3.3.1	Search	3-5
3.3.2	Replace	3-6
3.3.3	Character classes and closure	3-6
3.3.4	Examples	3-6
3.3.5	Editor as programmer's tool	3-7
3.4	User.cm file entries	3-9
4	Executive	
4.1	Files	4-1
4.2	User interface	4-1
4.2.1	Editing functions	4-1
4.2.2	Command line expansion	4-2
4.2.3	Command line interpretation	4-3
4.2.4	Built-in commands	4-3
4.2.5	Exec Ops menu	4-10
4.3	User.cm processing	4-10
5	HeraldWindow	
5.1	Files	5-1
5.2	User interface	5-1
5.2.1	Boot From: menu	5-1
5.3	User.cm processing	5-2

6	Profile tool	
6.1	User interface	6-1
7	Tool Driver	
7.1	Files	7-1
7.2	User interface	7-1
7.2.1	Message subwindow	7-2
7.2.2	Form subwindow	7-2
7.2.3	File subwindow	7-3
7.3	Script files	7-3
7.3.1	Script file format	7-3
7.3.2	Sample script	7-6
7.4	BNF for script files	7-7
7.5	The subwindows file	7-9
7.6	Running the Tool Driver	7-9
II	File-related tools	
II.1	File system conventions	II-1
II.2	File names	II-1
II.3	File-related tools	II-2
8	Brownie	
8.1	Files	8-1
8.2	User interface	8-1
8.3	Script file	8-1
8.3.1	Parameters	8-2
8.3.2	Commands	8-2
8.4	Example	8-3
9	FTP	
9.1	Files	9-1
9.2	User interface	9-1
9.2.1	Command line syntax	9-1
9.2.2	Command line switches	9-1
9.2.3	Commands and examples	9-3
9.2.4	Command line errors	9-6
9.3	Tutorial	9-7

Table of contents

10	File Tool	
10.1	Files	10-1
10.2	User interface	10-1
10.2.1	Form subwindow	10-2
10.2.2	Command subwindow	10-3
10.2.3	List Options window	10-4
10.3	User.cm	10-4
10.4	Operational notes	10-5
11	Floppy commands	
11.1	Files	11-1
11.2	User interface	11-1
11.2.1	Common argument definitions	11-1
11.2.2	Commands	11-1
11.3	Partial files	11-3
11.4	Examples	11-3
11.5	Error messages	11-4
12	Search Path Tool	
12.1	User interface	12-1
12.1.1	Form subwindow	12-1
12.1.2	Directories menu	12-2
12.1.3	Search Path menu	12-2
13	Compare	
13.1	Files	13-1
13.2	User interface	13-1
13.2.1	The Compare Tool window	13-1
13.2.2	Compare via the Executive window	13-3
14	Find	
14.1	Files	14-1
14.2	User interface	14-1
14.2.1	Switches	14-1
14.2.2	Switches on file names	14-2
14.2.3	Special characters	14-2
14.3	Examples	14-3

15	File Window	
15.1	Files	15-1
15.2	User interface	15-1
15.2.1	Debugger Ops menu	15-1
15.2.2	File Window menu	15-2
15.3	User.cm	15-3
16	Print	
16.1	Files	16-1
16.2	User interface	16-1
16.2.1	Switches	16-2
16.2.2	Defaults	16-3
16.3	Formatting	16-4
16.4	User.cm entries	16-4
III	System-building tools	
III.1	Files	III-1
III.2	Creating a source file	III-1
III.3	Creating an object file	III-2
III.3.1	Compiling a program	III-2
III.3.2	Binding a configuration	III-3
III.3.3	Summary	III-4
III.4	Running a program in the Tajo environment	III-5
III.4.1	Snarfing and running	III-5
III.4.2	Using Command Central	III-5
III.4.3	Summary	III-6
III.5	Making boot files	III-6
III.5.1	Packaging a system	III-6
III.5.2	Package operator	III-6
III.5.3	Using MakeBoot	III-7
III.5.4	Summary	III-7
III.6	Using the Debugger	III-7
III.6.1	Invoking CoPilot	III-7
III.6.2	Talking to the Debugger	III-8
III.6.3	Debugging a client program	III-8
III.6.4	Pilot symbols files	III-14
III.6.5	Interpreting signals	III-14
III.6.6	Address and write-protect faults	III-15
III.6.7	Tracing an address fault	III-16

Table of contents

III.7	Program-building tools	III-18
III.8	Program analysis tools	III-18
17	Binder	
17.1	Files	17-1
17.2	User interface	17-2
17.2.1	Command line	17-2
17.2.2	Switches	17-3
17.2.3	Associating files with modules and configurations	17-4
17.3	Examples	17-4
17.4	Error messages	17-5
17.5	Current limitations	17-7
18	Command Central	
18.1	Files	18-1
18.2	User interface	18-1
18.2.1	Message subwindow	18-2
18.2.2	Command subwindow	18-2
18.2.3	Log subwindow	18-3
18.3	Communication between client and development volumes	18-3
18.4	User.cm	18-4
19	Compiler	
19.1	Files	19-1
19.2	User interface	19-1
19.2.1	Command line	19-2
19.2.2	Switches	19-3
19.3	Examples	19-6
19.4	Error messages	19-6
19.5	Compiler failures	19-8
19.6	Current limitations	19-8
20	Formatter	
20.1	Files	20-1
20.2	User interface	20-1
20.2.1	Command line	20-2
20.2.2	Switches	20-2
20.3	Formatting rules	20-3
20.3.1	Spacing	20-3

20.3.2	Structure	20-4
20.4	User.cm	20-5
20.5	Examples	20-5
20.6	Formatter failures	20-6
21	MakeBoot	
21.1	Files	21-1
21.2	User interface	21-1
21.2.1	Commands	21-2
21.2.2	Switches	21-3
21.2.3	Parameter files	21-3
21.2.4	Examples	21-5
22	MakeDLionBootFloppy Tool	
22.1	Files	22-1
22.2	User interface	22-1
22.2.1	Form subwindow	22-1
22.2.2	Command subwindow	22-2
23	Packager	
23.1	Files	23-2
23.2	User interface	23-2
23.3	Information about modules	23-4
23.4	Packaging description language	23-5
23.4.1	Code segments	23-5
23.4.2	Discarded code packs	23-8
23.4.3	Frame packs	23-9
23.4.4	Merging	23-9
23.4.5	Rules governing packaging descriptions	23-10
23.4.6	Placement of multiword read-only constants	23-11
23.4.7	Example	23-11
23.5	Operation	23-12
24	Debugger	
24.1	Files	24-1
24.2	Installing and invoking CoPilot	24-1
24.2.1	Teledebugging	24-2
24.3	User Interface	24-3
24.3.1	Talking to the Debugger	24-3

Table of contents

24.3.2	Debugger commands	24-8
24.3.3	The Debugger interpreter	24-19
24.4	Signal and error messages	24-23
24.4.1	Entering the Debugger	24-23
24.4.2	Symbol lookup	24-25
24.4.3	Unrecognized structures	24-26
24.4.4	Command execution errors	24-26
24.4.5	Breakpoints	24-27
24.4.6	Displaying the stack	24-29
24.4.7	Interpreter	24-29
24.5	User.cm	24-32
24.6	CoPilot interpreter grammar	24-33
24.7	CoPilot summary	24-34
25	DebugHeap	
25.1	Files	25-1
25.2	User.cm	25-2
25.2.1	Form subwindow	25-2
25.2.2	DebugHeap menu	25-3
25.3	Example	25-4
26	IncludeChecker	
26.1	Files	26-1
26.2	User interface	26-1
26.2.1	Tool interface	26-2
26.2.2	Command line	26-4
26.2.3	Operating switches	26-4
26.3	Examples	26-5
26.4	User.cm	26-7
27	Lister	
27.1	Files	27-1
27.2	User interface	27-1
27.2.1	Commands useful to general Mesa users	27-2
27.2.2	Commands useful to wizards	27-3

28	Performance tools	
28.1	Control Transfer counter tool	28-2
28.1.1	Files	28-2
28.1.2	User interface	28-2
28.1.3	Operation	28-4
28.1.4	Limitations	28-5
28.1.5	Getting started.	28-6
28.1.6	Sample session.	28-6
28.2	Performance Measurement Tool	28-8
28.2.1	Files	28-9
28.2.2	Concepts	28-9
28.2.3	Definition of terms.	28-9
28.2.4	User interface	28-10
28.2.5	Operation	28-13
28.2.6	Limitations	28-14
28.2.7	Getting started.	28-15
28.2.8	Sample session.	28-15
28.3	Spy	28-17
28.3.1	Files	28-17
28.3.2	User interface	28-17
28.3.3	Operation	28-19
28.3.4	Getting started.	28-19
28.3.5	Error messages.	28-20
28.3.6	Limitations	28-21
28.4	Ben	28-21
28.4.1	Files	28-21
29	Statistics	
29.1	Files	29-1
29.2	User interface	29-1
29.2.1	Switches	29-1
29.3	Types of statistics.	29-2
29.4	Example	29-2
IV	Mesa Services	
30	Mail tools	
30.1	Mail Tool	30-1
30.1.1	Files	30-1
30.1.2	User interface	30-1

Table of contents

30.1.3	The Mail Tool via the Executive window	30-7
30.1.4	Send Tool	30-7
30.2	MailFileScavenger	30-13
30.2.1	Files	30-13
30.2.2	User interface	30-13
30.3	Maintain	30-13
30.3.1	Files	30-14
30.3.2	User interface	30-14
31	MFileServer	
31.1	Files	31-1
31.2	User interface	31-1
31.2.1	Form subwindow	31-2
31.2.2	Executive commands	31-2
31.3	User.cm entries	31-2
31.4	Operational notes	31-3
32	Network executive tools	
32.1	Chat	32-1
32.1.1	Files	32-1
32.1.2	User interface	32-1
32.1.3	Special keys	32-3
32.1.4	Chat User.cm	32-4
32.2	NSTerminal	32-4
32.2.1	Files	32-4
32.2.2	Setting up	32-4
32.2.3	User interface	32-5
32.2.4	Opening a connection	32-9
32.2.5	NSTerminal User.cm	32-10
32.2.6	User.cm example	32-10
32.3	Remote Executive	32-11
32.3.1	Files	32-11
32.3.2	User interface	32-11
32.3.3	Commands	32-11
32.3.4	Remote Executive User.cm	32-12
32.4	TTY Tajo	32-13
32.4.1	Files and installation	32-13
32.4.2	User interface	32-13
32.4.3	Commands	32-14

32.4.4	User.cm	32-14
32.4.5	Program interface	32-14
V	TCP/IP Related Tools and Applications	
33	ARPA Getting Started	
33.1	Installing the ARPA network protocols in XDE	33-1
34	ArpaCacheAddress	
34.1	Files	34-1
34.2	User Interface	34-1
35	ArpaChat	
35.1	Files	35-1
35.2	User Interface	35-1
35.2.1	Message subwindow	35-1
35.2.2	Form subwindow	35-1
35.2.3	TTY subwindow	35-3
35.2.4	Special keys	35-3
35.2.5	ArpaChat User.cm entries	35-3
36	ArpaRemoteExec	
36.1	Files	36-1
36.2	User Interface	36-1
36.2.1	Commands	36-2
36.2.2	Remote Executive User.cm.	36-3
37	ArpaFileTool	
37.1	Files	37-1
37.2	User Interface	37-1
37.2.1	Form subwindow	37-1
37.2.2	Command subwindow	37-2
37.2.3	Options window	37-3
37.2.4	Options command subwindow	37-3
37.2.5	Options form subwindow	37-3
37.3	User.cm entries	37-5
37.4	References	37-5

Table of contents

38	ArpaFileServer	
38.1	Files	38-1
38.2	User Interface	38-1
38.2.1	Tool window interface	38-1
38.2.2	Executive interface	38-2
38.2.3	Server activity log	38-2
38.3	User.cm Entries	38-2
39	ArpaMailTool	
39.1	Files	39-1
39.2	User Interface	39-1
39.2.1	Text subwindow – Table of contents	39-2
39.2.2	Form subwindow	39-3
39.2.3	Options window	39-4
39.3	ArpaSendTool	39-5
39.3.1	Form subwindow	39-6
39.3.2	Text subwindow	39-7
39.3.2.1	Subject: field	39-7
39.3.2.2	To: field	39-7
39.3.2.3	Reply-To: field	39-7
39.3.2.4	cc: field	39-7
39.3.2.5	bcc: field	39-8
39.3.2.6	Message body	39-8
39.3.2.7	User.cm entries	39-8
39.4	MailFileScavenger	39-10
39.4.1	Files	39-10
39.4.2	User interface	39-10

Appendices

A	Othello	
A.1	Files	A-1
A.2	Running Othello	A-1
A.3	User interface	A-1
A.3.1	Accessible disk drives	A-2
A.3.2	Checking a pack	A-3
A.3.3	Physical volumes	A-3

A.3.4	Logical volumes	A-4
A.3.5	Initial microcode, Pilot microcode, diagnostic microcode, germ, and boot files	A-6
A.3.6	Time	A-10
A.3.7	Routing tables and echo user	A-11
A.3.8	Accessing the debugger during early initialization of Pilot	A-11
A.3.9	Exiting Othello	A-12
A.3.10	Special commands	A-12

B Getting started/Operations guide

B.1	Booting	B-1
B.1.1	The maintenance panel	B-2
B.1.2	Standard booting	B-2
B.2	Setting up volumes: initializing your system	B-4
B.2.1	Example of initializing volumes	B-5
B.2.2	Booting volumes from other volumes	B-6
B.2.3	Boot switches	B-6
B.2.4	Xerox Development Environment boot switches	B-8
B.3	Installing boot files	B-9
B.3.1	Initializing debuggers	B-9
B.3.2	Setting debugger pointers	B-10
B.4	Installing the development environment	B-10
B.4.1	Tools	B-10
B.4.2	The user command file	B-11
B.5	Recovering from disasters	B-11
B.5.1	Dandelion boot microcode maintenance panel error codes	B-12
B.5.2	Pilot maintenance panel codes for errors	B-12
B.5.3	Pilot error messages	B-14
B.6	Ending a session	B-16

C TableCompiler

C.1	Mesa object file format	C-1
C.2	Using the output	C-1
C.3	ModuleMaker	C-2
C.4	StringCompactor	C-3
C.4.1	Example	C-3
C.5	File format	C-4
C.6	Options	C-4
C.7	Command line syntax and switches	C-5
C.8	Examples	C-5
C.9	Switches on the input file name	C-6
C.10	Switches on auxiliary file names	C-6

Table of contents

D Parser Generator System

D.1	Using the Parser Generator	D-1
D.2	Format the input file	D-2
D.3	Output of the Parser Generator	D-3
D.3.1	The input record	D-4
D.3.2	The log file	D-4
D.4	The module file	D-9
D.5	The binary file	D-9
D.5.1	Binary file format	D-9
D.5.2	The LR and first files.	D-11
D.6	The Preprocessor	D-12
D.7	Operation	D-13
D.7.1	PGS operation	D-13
D.7.2	TableCompiler operation.	D-16
D.8	Example	D-17

E Sword Debugger

E.1	Events	E-1
E.2	Styles of debugging	E-1
E.2.1	Local debugging	E-1
E.2.2	Outload debugging	E-2
E.2.3	Remote debugging	E-2
E.3	User interface	E-3
E.3.1	Sword Tool	E-3
E.3.2	Interpreter Tool	E-5
E.4	Debugger commands	E-14
E.4.1	Breakpoints	E-14
E.4.2	Display runtime state	E-17
E.4.3	Current context	E-20
E.4.4	Program control	E-21
E.4.5	Low-level facilities.	E-22
E.5	The Debugger interpreter.	E-22
E.5.1	Statement syntax	E-23
E.5.2	Loopholes	E-23
E.5.3	Subscripting	E-23
E.5.4	Explicit qualification vs qualification in the current context	E-24
E.5.5	Type expressions	E-24
E.5.6	Radix conversion	E-24
E.5.7	Arithmetic expressions.	E-24

E.5.8	Procedure calls	E-24
E.5.9	Sample expressions	E-25
E.6	Signal and error messages	E-26
E.6.1	Entering the Debugger	E-26
E.6.2	Symbol lookup	E-26
E.6.3	Unrecognized structures	E-27
E.6.4	Command execution errors	E-27
E.6.5	Breakpoints	E-28
E.6.6	Displaying the stack	E-29
E.6.7	Interpreter	E-29
E.7	User.cm	E-32
E.8	Mesa Interpreter grammar	E-32
E.9	Commands summary	E-33

Index

Illustrations

Figure I.1: User interface	I-5
Figure I.2: Scrollbar	I-7
Figure I.3: Windows	I-8
Figure I.4: Form subwindow	I-9
Figure I.5: Menus	I-12
Figure I.6: Text window	I-16
Figure I.7: Keyboard	I-18
Figure 3.1: Editor Symbiote subwindow	3-1
Figure 3-2: Editor property sheet	3-3
Figure 7.1: Tool Driver executive window	7-2
Figure 10.1: File Tool window	10-1
Figure 12.1 : Search Path Tool window	12-1
Figure 13.1: Compare Tool window	13-2
Figure 18.1: Command Central tool window	18-1
Figure 22.1: MakeDLionBootFloppy tool	22-1
Figure 24.1: CoPilot	24-3
Figure 25.1: DebugHeap tool window	25-2
Figure 26.1: IncludeChecker tool window	26-2
Figure 28.1: Control Transfer Counter tool	28-2
Figure 28.2: PerfPackage window with node commands.	28-11
Figure 28.3: PerfPackage window with histogram commands	28-12
Figure 28.4: Spy tool window	28-18
Figure 30.1: The MailTool	30-3
Figure 30.2: Maintain tool window (normal level)	30-15
Figure 30.3: Maintain tool window (owner level)	30-15

Table of contents

Figure 31.1: MFileServer	31-1
Figure 32.1: Chat	32-2
Figure 32.2: NSTerminal	32-5
Figure 39.1: ArpaMailTool	39-2
Figure 39.2: ArpaMailTool Options Window and SMTP Debugger	39-5
Figure 39.3: ArpaSendTool	39-6
Figure E.1: Sword	E-3
Figure E.2: LocalWorld with Interpreter Options	E-6



Errata

Tajo

- On a 6085 keyboard, the **CASE** key has the same function as **KEYBOARD-L** on a 8010 keyboard. It will make the selection lower case, and if shifted it will make the selection upper case.
- **PROP'S-CR** "unindents" one level. For example, you can type it instead of a CR when you want to close a scope on the next line.
- **FileWindows.Save[]** from the debugger saves Empty windows to a file named "**ScratchWindows.saved**" on your client volume. You need **FileWindows.bcd** on your debugger volume to use this command from the debugger.
- **ScratchSources.Save[]** saves all scratch sources to a file named "**ScratchSources.saved**." Unlike **FileWindows.Save[]**, this one saves your mail send windows as well as your Empty Windows, but doesn't save FileWindows that you were editing. You need **ScratchSources.bcd** on your debugger volume to use this command from the debugger.
- A **SetPositionBalanceBeam** affects the way text is displayed in your windows. When you do a **FIND** or **Position** in a window, the position in question is displayed at the top of the window in "top" mode (the way Tajo has always worked), in the middle in "middle" mode, or at either the top or bottom, which ever is more convenient, in "topBottom" mode. TopBottom mode minimizes the repainting needed when you jump between various positions in the window. Top mode only saves repainting when jumping backwards. Middle mode doesn't save much at all, but it always positions things of interest in the middle of your window. Top mode is the default. A sample **User.cm** entry is:

```
[System]
SetPositionBalanceBeam: top | middle| topBottom
```

- A **CaretShape** switch selects between two different styles of caret. The default is "triangle," which gives you the standard Tajo TextSW and TTYSW carets. With **caretShape = IBeam**, however, you get an I-Beam caret in TextSWs and a gray

rectangle in TTYSWs. You can set this switch from the System section of your **User.cm**:

```
[System]
CaretShape: triangle | iBeam
```

- **MenuSymbiotes** can have their own font. You can specify what font you want them to have in the **FileWindow** section of your **User.cm**. The file name should have the **.strike** extension on it. The file should be on your root directory, <>, so the system can find it even before your search path is set up. You can also specify how many lines you would like your **MenuSymbiotes** to be. The **MenuSymbioteLines** field in your **User.cm** can be a real number, such as 2.37. It may take a few tries to get the **MenuSymbiotes** looking just the way you want them to. A sample **User.cm** entry is:

```
[FileWindow]
MenuSymbioteFont: MenuSymbolsFont.strike
MenuSymbioteLines: 1
```

- When you hit **DoIt** in a **FileWindow**, several default extensions are tried. These defaults (**.mesa .config .cm**) can be changed by specifying a list of extensions in the **FileWindow** sections of your **User.cm**. Any string starting with a '.' is allowed. For Example:

```
[FileWindow]
Extensions: .mesa .config .cm .doc .df .log
```

- **J.Last** positions the last line of the file in the middle of your window (even if you don't have **SetPositionBalanceBeam = middle**).
- If the **Notifier** is busy and is not taking any page faults, **Shift-STOP** won't take you to the debugger. In this situation, use **Shift-Shift-STOP** to get to the debugger. If you must do this, you can't execute **Interpret-Calls** from the debugger.
- When chording the mouse buttons to bring up a menu, release the **POINT** (=left=red) button as soon as you have brought up the menu. The menu stays up as long as you hold down the **ADJUST** button. Address faults may occur if you release the **ADJUST** button before releasing the **POINT** button when using menus.
- Avoid running Tajo or CoPilot with extremely full volumes. Tools can fail otherwise.
- Three boot switches have been added to set the parameters of the system zone:

```
'[ tiny initial: 4 pages, increment: 4 pages, large node: 128
  % standard initial: 40 pages, increment: 20 pages, large node: 260
  ]' large initial: 100 pages, increment: 50 pages, large node: 260
```

The largest size specified by the switches is the one used. If no switch is specified, the default switch '**%**' is used. Four-page uniform swap units are used throughout.

Reminder: the '**I**' boot switch causes all TIP tables to be re-initialized, which means all existing **.TIP** files are ignored, and new ones are written as needed.

- **NEXT** and **NEXT-DELETE** search from the insertion point, not the selection.
- **FileSystem**: If some tool in CoPilot gives the message that it could not close a volume, try to figure out why the **volumeAboutToClose** was cancelled. Fix the error, then close the volume manually by using the Exec's **CloseVolume** command. If you still can't close the volume, you must reboot your machine before proceeding to the client. If you open your client volume, or any volume readable from your client volume, for write, you must not proceed to your client.

Debugger

- The symbols for **HeapImpl** are in **UnpackedHeapImpl.bcd**. Once retrieved, Copilot realizes where the symbols are unless it has already given the "No symbols for **HeapImpl**" message. In this case, after retrieving **UnpackedHeapImpl.bcd**, you have to tell Copilot to either

Attach Symbols global frame: x Filename: UnpackedHeapImpl.bcd

or

↑Validate caches [Confirm]
--undocumented, done by **CONTROL-N <cr>**

- If you first interpret an expression containing a multi-word constant identifier and later try to interpret that (or another expression) containing the same multi-word constant identifier, CoPilot displays the error message "**!Literal problem. Invalidate Caches (CONTROL-N) and try again.**" This is a work around.
- Don't delete **.symbols** files or **.bcd** files that CoPilot has used for symbols. If you must delete such a file, you can improve your chances of CoPilot working by invalidating CoPilot's caches (**CONTROL-N** command).
- The syntax specifying a host to CoPilot's Remote Debug command as a network address is: "**net.host.**" (note the two periods; numbers are octal only). The command also recognizes any of the other formats documented in the AddressTranslation chapter of the *Mesa Programmer's Manual*.
- You can set conditional breaks of these forms:

1. **<number>** -- copilot breaks every **<number>**th occurrence

2. **<expression> <relation> <expression>**

where

<relation> is one of **< <= = > = > #** and

<expression> is one of the following:

- constant
- local variable
- parameter or return result at ENTRY/EXIT

- d. global variable
 - e. pointerToRecord.field -- note that
pointerToRecord.array[constantExpression] is legal
 - f. pointerToArray[constantExpression]
 - g. pointerToSequence[constantExpression]
 - h. descriptor[constantExpression] all or these values can be [1..16] or 32 bits in length
- To go backward in the stack, use "b" or "jump -n."
 - Hex numbers may be entered with an 'H' or 'h' suffix. Relations are implemented (=, #, >, <, >=, <=). Real numbers and their operations are implemented.
 - To get concrete values of opaque types, use **Attach Opaque**. For example,

Attach Opaque: Window.Object Filename: WindowImplB

causes all **Window.Objects** to be printed as **WindowImplB\$Object**. The attachment remains in effect until a new session.

- The debugger supports multiple remote debuggers.
- A sample debugger **User.cm** entry is:

```
[Debugger]
Boot: VolumeName
cRadix: octal | decimal | hex
cSigned: TRUE | FALSE
iRadix: octal | decimal | hex
iSigned: TRUE | FALSE
pRadix: octal | decimal
lpRadix: octal | decimal
relRadix: octal | decimal
unspec: CARDINAL | INTEGER
elements: ArrayElementsToShow
chars: StringCharactersToShow
```

Executive

- The commands **Floppy.^ erase** and **Floppy.^ scavenge** can be called from the executive.

Tools

- **Compare** does not work for files on NS file servers, and ignores more than 5,120 lines of a file.
- MakeDLionBootFloppyTool now has an option to let you reserve the last cylinder for diagnostics. The user interface has changed slightly. The command **formSW** has changed to contain three booleans and a command:

Format ReserveLastCylinderForDiagnostics InstallFiles Start!

After selecting the desired options, select **Start!**

- Brownie won't transfer Non-XDE file from NS file servers. In particular, long file names, non-standard file types (such as ViewPoint file types), and multi-segmented files (such as ViewPoint documents) are not supported.
- The **List/f** and **List/b** commands of FTP have a syntax different from that described in the XDE User Guide. Only one of the /f or /b switches can be used and it must be the last switch. After one of these switches is seen, the rest of the command line is assumed to be a list of files. The new syntax is:

>FTP List/f date-with-no-spaces <files>

>FTP List/f "date with spaces" <files>

The date can be in any valid format for dates. The /f switch lists the files that have a create date after the date given. The /b switch lists those files with a create date before the date given.

Example:

> FTP RamRod Dir/c AMesa List/dalf 10-Oct-84 '*'

- Command Central and the **Run.~** and **Load.~** commands of the Executive now recognize the v switch, which causes version mismatches to be ignored.
- There are new built-in Exec commands. **Protect.~** changes the protection status of files and directories. **Registry.~** sets the default registry. **Clearinghouse.~** sets the default domain and organization. You must execute the Login command after the Clearinghouse command in order to update the Clearinghouse of the logged in user. Type "**Help.~ <command>**" in the Executive window for more information.
- When trying to re-execute an Executive command by selecting and stuffing a previous command, you may accidentally select the prompt character '>'. If so, the command that the Executive tries to run will start with the character > and will not match any of the registered commands. However, it will match the corresponding file when the Executive tries its autoload heuristic, causing the Executive to load another instance of your bcd.
- You can specify the font to use on the command line. For example:

>Formatter /-tikg Souvenir/f Def.mesa Impl.mesa

The 'f' switch says that this is a font. It should come after the global switches and before any files to be formatted. Note that no size is given, just the name of the font. The formatter picks 10 point for portrait and 8 point for landscape. There are also new **User.cm** entries for the formatter:

```
[Formatter]
LandscapeFont: Souvenir
PortraitFont: Classic
```

- The /k (Output Packager Command) switch writes Packager commands in the output file that make Packager source and object files consistent (default TRUE).
- Makeboot contains some new options and some old options have been removed. The GFT entry is obsolete in the bootmesa file. The gftLength command line argument is also obsolete.

In the bootmesa file the options:

```
LOADSTATEMODULES: number
LOADSTATEBCDS: number
```

are now available. These items set the number of empty module and bcd slots you want in the initial LoadState. Such entries are used when, for instance, modules are loaded or NEW'ed. This number does not include the modules and bcds in the boot file. In the command line, the options:

```
lsModules: number
lsBcds: number
```

are now available. These items override the numbers in the parameter file given by LOADSTATEMODULES and LOADSTATEBCDS, respectively. They have the same meaning as LOADSTATEMODULES and LOADSTATEBCDS, respectively. Since Pilot automatically expands the loadstate as necessary, these numbers are optional and need not be accurate. The /u switch may be added to the **MakeBoot** command for UtilityPilot-based bootfiles. This switch has no effect on program execution but makes the bootfile smaller by eliminating unnecessary data.

With multiple bootmesa files, the last file takes precedence for all parameters except lsModules, lsBcds, and processes, in which cases the first bootmesa file takes precedence. Parameters on the command line override those in a bootmesa file.

Also, MakeBoot takes "\nnn" boot switches in its command line if they are enclosed in double quotes, such as:

```
MakeBoot OthelloTriDLion[parm: UtilityPilot, parm:
UtilityCommunication, switches: "\372"]/dhu
```

Librarian

- The **Set Backup Path** command also prompts for the number of backup versions to keep. It keeps this number of copies of the data base index and record files for recovery purposes. Only one version is necessary; additional copies are simply precautionary.

Example:

```
LS!Set Backup Path
Which database?
```

```
1 Mesa-Libjects
2 Pilot-Libjects
Enter choice number: 2
Path to back up files for this data base: (Rasp:osbu
north)LibrarianBackup/
Number of backup versions to keep (1..100): 2
```

- The **Checkin Libject** command allows administrators to check in libjects checked out by someone other than themselves. It prompts for the data base name and libject to be checked in. Example:

```
LS!Checkin Libject
Which database?
1 Mesa-Libjects
2 Pilot-Libjects
Enter choice number: 2
Libject name: FileTransfer.df
Checking in FileTransfer.df ... Done.
```

- Strong authentication is now supported.

Internal Tools

- **Filename/F** (read commands from a file) for Chat doesn't work. Chat stuffs initial commands into its window when a connection is opened if the autologin feature is enabled. The commands stuffed can be specified in the **User.cm** as follows:

```
[Chat]
machine: quotedStringWithCarriageReturns
booter: "execpupchat /u vaxc
"
oxnard: "sets <main>me <main>
"
```

- The **Protect.**~ command isn't in TTYTajo.

Errata

A *text item* is a display string that you may modify using the editing functions (see the section in this chapter on Text manipulation). A text item is distinguished from other form items by the ":" (note the space after the colon) appended to a text form item keyword. Several accelerators are available for text form items. Clicking **Point** over the keyword selects all of the text in the form item and moves the type-in point to the end of the text. For example, clicking **Point** over **Password:** in the Profile Tool causes the type-in point to be positioned after the colon, ready for you to type in your password. Generally, clicking the **Adjust** button over the keyword deletes the text and sets the type-in point.

Fine point: When a password is entered, an asterisk is displayed for each character typed.

A *numeric item* is like a text form item, except that only strings representing numbers may be modified. A numeric item is distinguished from other form items by the "=" (note the space after the equal sign) appended to the keyword.

A *tag items* is a text string used to annotate a form. A tag item labels something that appears either elsewhere on the screen or entirely off the screen.

Menu prompts are always available for enumerated form fields and are optional in some textual form fields. When you chord the mouse buttons with the cursor over the keyword for an enumerated field, a menu of allowed values for the form item is displayed. Choosing one of the values from the menu sets the form item to that value. Similarly, when you chord with the cursor over the keyword for a textual field, a menu of character strings is displayed. Choosing one of the items (strings) from the menu will cause the menu string to be appended at the current position of the type-in point.

Specific form items are described in later chapters with the tools to which they belong.

I.3.1.4.2 Text subwindows

Most text display, other than in form subwindows, occurs within *text subwindows*. Text subwindows may be associated with a file that contains the text. A *TextOps* menu is supplied with a text subwindow. The Text Ops menu contains commands specific to text manipulation (see next section).

I.3.2 Text manipulation

Text may be entered, edited, moved, and deleted in certain subwindows, which are appropriately called *text subwindows*. Selections may also be moved between subwindows.

I.3.2.1 Selecting text

The concept of a current selection is global. There is only one current selection at any time (not one per window); it is generally used as the argument to commands.

Fine point: Although a current selection is always video-inverted, not all video-inverted entities are considered current selections (such as when a menu command is invoked).



General tools

This chapter is an overview of the Xerox Development Environment (XDE) and its use. It describes the types of features in the environment and how they interact. The final sections of this chapter discuss other XDE documentation, the organization of this manual, and its typographical conventions.

This chapter also introduces a number of helpful tools found on the XDE system. These tools are discussed in chapters 1 through 7.

I.1 System overview

The Xerox Development Environment provides development tools for programmers writing tools and applications, including tools to aid in editing, compiling, binding, running, and debugging Mesa programs.

I.1.1 User interface

A tool communicates with the user via *windows*, which are rectangular regions of the display screen in which text, icons, and graphics are displayed. User input to a window is collected using *menus* or *form subwindows*. A menu is a list of options or commands associated with a window. Tajo, the XDE runtime environment, allows programmers to define specific menus meaningful to a particular tool. Another way to collect user input is through a form subwindow, which is a horizontally ruled section of a window used for displaying commands and argument names.

In addition to window-oriented facilities, XDE provides a simple executive facility for invoking the same tools using a less sophisticated teletype-style interface. Tools of this type are invoked through the Executive window by typing the tool name and the appropriate parameter syntax.

I.1.2 Development scenario

A complete development scenario includes design, implementation, testing, and release of systems. During implementation, the programmer produces code using pre-existing modules consistent with the design. After writing or retrieving the necessary modules, they are separately compiled and then bound together. Once bound, the entire system, referred to as a *configuration*, can be debugged. Each time an error is corrected, the

process of compiling and binding is repeated until the system is free of bugs. After debugging, modules are stored on file servers, the entire system is tested, and then it is released to the user community.

For more general information about the XDE system, see *XDE: Concepts and Principles*.

I.1.3 Hardware

The XDE programming environment is designed for a personal computer. It runs on a powerful microcoded processor (the Dandelion) with a large virtual address space. The user interface uses a high-resolution bitmap display, with a keyboard and a pointing device called a mouse. Secondary storage is provided by a rigid disk and an optional eight-inch floppy disk. The Ethernet, a local area network, provides a high-bandwidth connection to other personal computers and to network services, such as print and file servers. (*XDE: Concepts and Principles* provides general information about networking concepts used in Xerox products.)

I.1.4 Software components

To illustrate the interaction between the various systems, it is helpful to envision a hierarchy with Pilot, the operating system kernel, at the lowest level. The next system up the hierarchy is Tajo, a specialized collection of interfaces designed to facilitate the implementation of software development tools. At the top of the hierarchy is CoPilot, the debugger. Although Tajo and the Xerox Development Environment may seem similar since they both support programming activities, the distinguishing factor is that the development environment includes programs specific to the Mesa language, whereas Tajo is language independent.

Othello is a Mesa program that manages Pilot physical and logical disk volumes. Since it does not provide any programming facilities, it is not considered part of the hierarchy. Appendix A describes Othello.

I.1.4.1 Pilot

Pilot provides Mesa runtime support, including processes, monitors, and synchronization facilities. Pilot supports a collection of cooperating user-defined processes, some of which are the tools. Since allocation of major system resources is generally on a cooperative rather than a competitive basis, Pilot does not contain elaborate resource allocation functions. Instead, resources and resource management are typically planned statically when systems are configured. In instances requiring dynamic resource control, such as the sharing of physical memory, Pilot provides facilities that allow the applications to state their current requirements. Consistent with the notion of clients as cooperating processes, Pilot provides only limited protection against malicious programs, thereby shifting the responsibility of ensuring smooth operation to Pilot clients. The Pilot operating system is implemented entirely in the Mesa language. (Pilot is discussed briefly in Appendix B and described in detail in the *Pilot Programmer's Manual*.)

I.1.4.2 Tajo

Tajo is a unified set of facilities supporting the implementation and execution of software development tools. "Using" Tajo can be viewed in two ways; a *user* is a person who interacts with Tajo via the mouse and keyboard; a *client* is a program that uses the Tajo software interfaces. *Tools* are the Client programs that call upon Tajo.

I.1.4.3 CoPilot

CoPilot supports source-level debugging. It allows users to interpret Mesa statements, set breakpoints, trace program execution, and display the runtime state. Pilot provides the code necessary for a program to communicate with CoPilot; it resides with the user program. CoPilot, however, resides in a different memory image (on a separate logical volume) that is loaded when called for. This protects the client and the debugger from each other, in addition to providing the separate address space required to implement all of CoPilot's capabilities.

There are several ways of invoking the Debugger, some under programmer control and others not. Those under programmer control include setting breakpoints and interrupting a program during execution. These techniques are used when a programmer anticipates some problems and wishes to halt execution temporarily to examine (and possibly change) the program state before proceeding. CoPilot may also be invoked automatically when a program generates runtime errors, such as address faults or uncaught signals. If the Debugger is invoked because of a runtime error, you can often change the state of the program by using the appropriate debugger commands and continue executing from the new program state. However, some errors, such as memory overwrites, cause irreparable damage. When this happens, you must end the debugging session and re-boot the client.

I.1.4.4 Othello

Othello is a utility for managing Pilot physical and logical volumes. It is used to initialize physical and logical volumes, to install boot files on logical volumes, to invoke a boot file on a particular logical volume, and to start scavenging logical volumes. In the normal development cycle, Othello is booted from a rigid disk. However, if the disk has never been booted or has been erased, Othello can be booted from the Ethernet or from a bootable floppy disk. For more information about Othello, see Appendix A.

I.2 Definition of terms

<i>Accelerator</i>	An <i>accelerator</i> 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 ").
<i>Argument</i>	An <i>argument</i> 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.

<i>Chord</i>	To <i>chord</i> keys or buttons is to push them down at the same time, as when chording the mouse buttons.
<i>Click</i>	To <i>click</i> a mouse button is to press down on it and let it up.
<i>Current selection</i>	The <i>current selection</i> is text, icons, or graphics you have chosen by using the mouse (current tools do not implement selection of icons or graphics). It is visually highlighted on the screen and is generally used as the argument to a command.
<i>Cursor</i>	The <i>cursor</i> 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.
<i>Icon</i>	An <i>icon</i> is a small picture on the display representing some entity.
<i>Input Focus</i>	The <i>input focus</i> is the window to which keyboard commands and typed characters are sent. The input focus contains the type-in point.
<i>Interface</i>	An <i>interface</i> 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.
<i>Menus</i>	A <i>menu</i> 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
<i>Mouse</i>	The <i>mouse</i> is a pointing device that allows you 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 .)
<i>Movable boundary</i>	A <i>movable boundary</i> 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.
<i>Name frame</i>	The <i>window name frame</i> is a rectangular region at the top of a window. It is usually black, with the window's name and other identifying information displayed in white.
<i>Subsystem</i>	A <i>subsystem</i> is a program that runs in the Xerox Development Environment Executive window. Some subsystems and tools accomplish the same task.
<i>Subwindow</i>	A window is often composed of one or more rectangular <i>subwindows</i> . The Xerox Development Environment provides several standard subwindow types, each providing different functions. (See <i>Window</i>).
<i>Tool</i>	A <i>tool</i> 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.

<i>Type-in point</i>	The <i>type-in point</i> is the text location where typed characters are to be inserted. The type-in point is indicated by a flashing caret or box.
<i>Video-invert</i>	To <i>video-invert</i> a region is to cause black areas of the region to become white and white areas to become black.
<i>Window</i>	A <i>window</i> is a rectangular region of the display in which text and graphics can be displayed. Most tools communicate via windows.

I.3 User interface

The user interface for tools provides the unifying framework for the development environment. Tools portray their capabilities through windows and menus. Windows and menus rely on XDE features such as text handling and keyboard or mouse commands.

This section describes text manipulation, keyboard commands, symbiates, windows, subwindows, and menus. It discusses some important menus and their commands. (The definition of a particular window or menu is always found in the chapter on the related tool.)

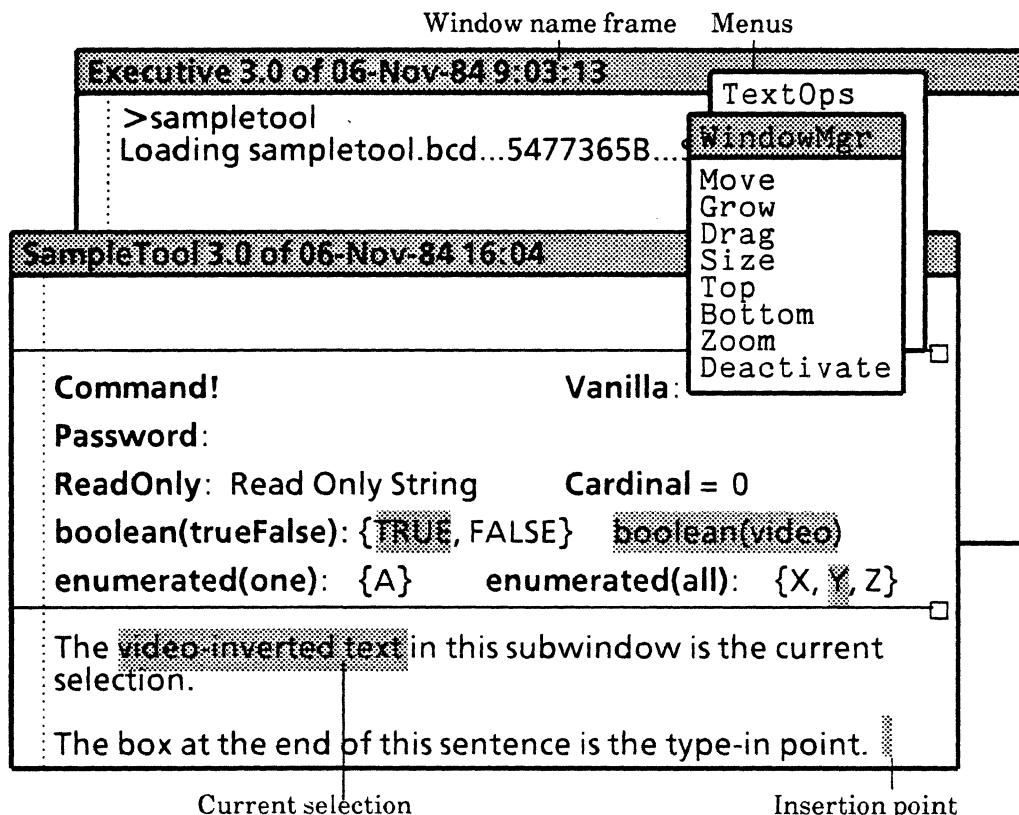


Figure I.1: User interface

I.3.1 Windows and subwindows

A window is a rectangular region of the display screen that offers a view of a potentially infinite plane. Most tools have one or more windows.

Each window is composed of one or more subwindows. Subwindows are regions of the window, each with individual characteristics. Subwindows are usually arranged vertically, with horizontal black lines dividing them. A window allows you to communicate with the tool to which it belongs and allows a tool to create a representation of a world owned and managed by that tool. The tool displays text and graphics, some of which may be lying out of sight.

One tool can create multiple windows, but each window is owned by a single tool. There may be multiple windows on the screen, and they may overlap and partially or fully obscure other windows. There may be stacks of windows lying on top of each other, each with its status and context intact, as if they were pieces of paper lying on a desk.

A tool window has three states: active, tiny, and inactive. An active tool window appears ready for communication. Like a hammer or wrench, an active tool can be picked up, used, and put down again; it remains exactly as it was left. When an active tool window is made tiny, it is represented on the display by a small box (an iconic representation) containing only its name. Making a tool tiny is like putting a tool in a tool belt: it will probably be used soon, but the tool user wants to get it out of the way for a while. When a tiny tool is returned to normal size, the contents of its window reappears. When a tool is made inactive, any information it keeps while active or tiny is discarded. When the tool window is subsequently activated, it appears as if it had just been created. Making a tool inactive is similar to cleaning off a wrench and placing it into the tool box. It will probably not be used for a while, and the tool user wants to make room for other tools.

An exception to this general behavior of windows is the root window. You can think of it as a window the size of your display screen that lies at the bottom of any stack of windows. The root window can never be at the "top" of the stack of menus on your screen, or all the rest would be covered! Certain menus are attached to the root window as to any other window: the Exec Ops menu, the Inactive menu, and the Symbiote menu. (See the section on menus below for more specific information about these menus.)

I.3.1.1 Communicating via subwindows

A tool accepts input via the keyboard and mouse buttons. Each subwindow may have different interface characteristics, and the meaning of the keyboard keys and mouse buttons may change when they are accepted by a different subwindow.

In general, all keystrokes are sent to the subwindow that has the input focus. The following keystrokes are exceptions: they are sent to the subwindow that contains the cursor: **MENU**, **FIND**, **J.FIRST**, **ABORT**, and the mouse buttons (**Point** and **Adjust**). If no window has the input focus, the screen blinks when keys are pressed. If the tool is busy when keystrokes are sent to it, the system queues the keystrokes and delivers them to the tool as soon as it is ready to accept input.

A subwindow keeps the input focus unless it is deactivated or the input focus is explicitly moved to a different window. For instance, it keeps the input focus if it has been made tiny or if it is completely obscured by other windows. You can set the input focus by depressing

one of the mouse buttons in the subwindow you would like to take the input focus. If the subwindow is unwilling to accept the input focus, the screen will blink.

If you set the input focus by pressing the **Point** button, the type-in point is set to the location under the mouse button (except in TTY windows, which insist that the type-in point always be at the end of the text). If you set the input focus by pressing the **Adjust** button, the type-in point is the last location that was the type-in point in the subwindow. Thus the **Adjust** button can be used to recover the type-in point in a subwindow after it has lost the input focus. While **MOVE** or **COPY** is depressed, using the mouse buttons will not change the input focus. If a subwindow does not want type-in itself, it may redirect it to another subwindow.

I.3.1.2 Scrolling

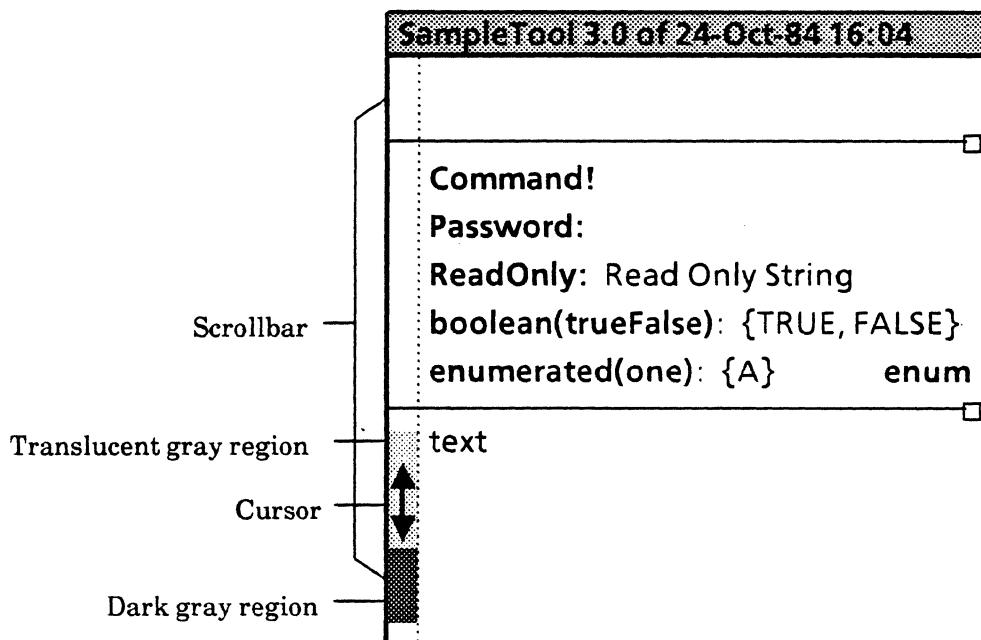


Figure I.2: Scrollbar

A subwindow may contain more information than can be displayed on the screen at one time. The development environment provides *scrollbars* (Figure I.2) to facilitate access to information lying out of view. Vertical scrollbars are long thin rectangles near the left border of subwindows. Some subwindows have horizontal scrollbars near the bottom border of a subwindow.

When the cursor is not in the scrollbar region, the scrollbar is a narrow transparent strip bordered by a gray stripe. When the cursor is in the scrollbar region, the scrollbar looks like a translucent gray region with a dark gray region within it (much like a thermometer). The transparent gray region represents the entire length of the contents of the subwindow. The dark gray region represents the text currently displayed; its size and position correspond to the position of the displayed text in the file.

When the cursor is in the scrollbar region, it changes to a double-headed arrow and the meaning of the mouse buttons change: they now direct the scrolling operation. The cursor changes again when one of the buttons is depressed: **Point** scrolls up and **Adjust** scrolls

down. Pressing both keys together (a "chord") is used for *thumbing*. Thumbing is analogous to opening a book by placing your thumb at the approximate position of the section you want to start reading and pulling the book open at that point. Releasing the chord while the cursor is positioned in the scrollbar invokes the scrolling operation; releasing the chord while the cursor is outside the scrollbar aborts scrolling.

I.3.1.3 Adjusting boundaries

You can change the movable boundaries of a subwindow by pressing **Point** while the cursor is positioned over the small box at the right end of the black boundary line, moving the cursor to the desired position, and releasing **Point**. Subwindows adjusted this way cannot be smaller than the height of the font being used.

Figure I.3 illustrates a stack of three windows belonging to two tools and the Executive. The Profile Tool is in tiny form in the upper right of the display.

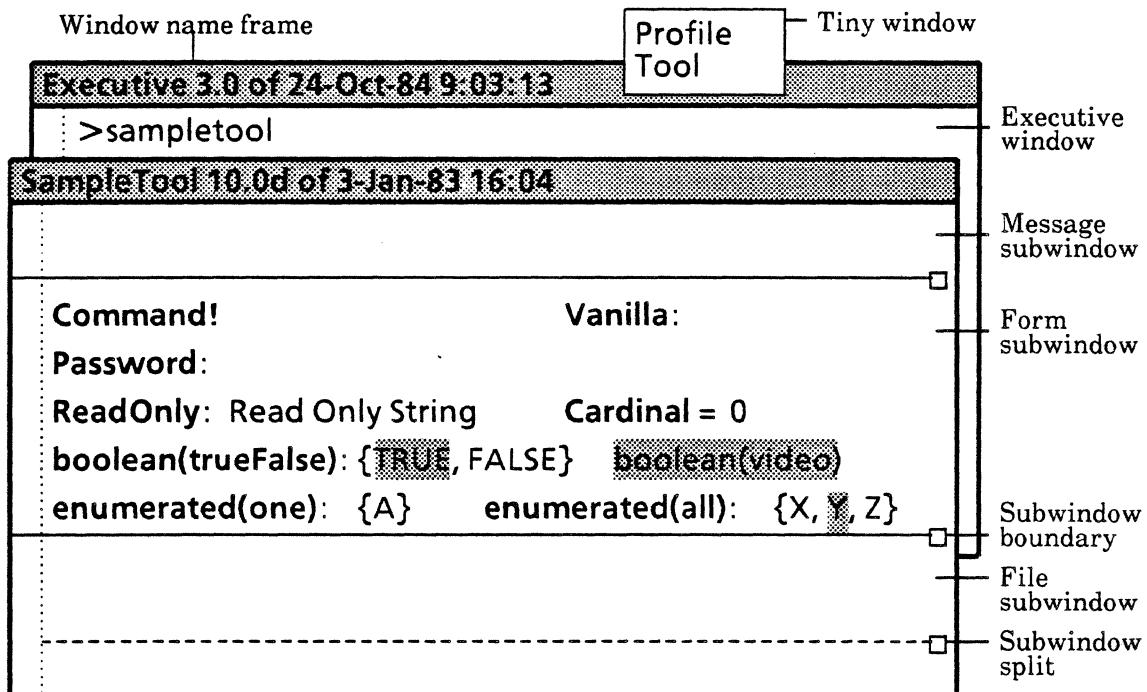


Figure I.3: Windows

I.3.1.4 Subwindow types

The two most important subwindow types for most purposes in XDE are form subwindows and text subwindows. They are described in the next sections.

I.3.1.4.1 Form subwindows

Form subwindows, which belong to specific tools, have two primary uses: First, they are used to display and alter the current values of the internal state of tool-specific data. Current values can be altered at any time in any order. Second, most form subwindows are

equipped with tool-specific command form items that act as accelerators for menu commands. A form subwindow is illustrated in Figure I.4.

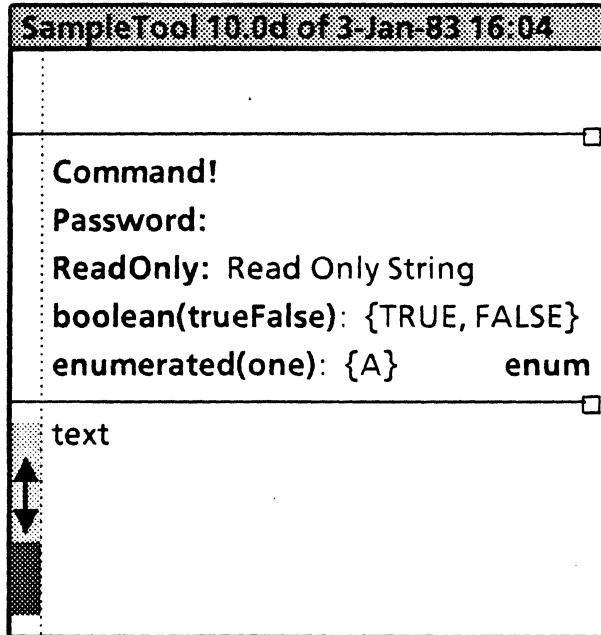


Figure I.4: Form subwindow

Tools normally display the arguments, and a single command invokes them. When an operation requires several arguments, they must be specified before invoking the operation. (Specific form subwindows are described in later chapters with the tools that own them.)

A form can have a variety of types of fields:

A *command item* performs the same function as a menu command. Command items are distinguished from other items by the ! appended to them. You can activate a command item by positioning the cursor over the keyword and depressing **Point**. Releasing **Point** over the keyword after the keyword is video-inverted invokes the operation. Releasing **Point** when the cursor is no longer positioned over the keyword cancels selecting that command.

An *enumerated item* is one of a lists of text items. These items may be displayed in two ways: *keyword: {a, b, c, ...}* or *keyword: {a}*. In either cases, choosing may be done via *menu prompts* (see below). In the first form, a choice in the list may also be chosen by positioning the cursor over it and clicking **Point**. The highlighted item is the current value. In the latter form, only the currently active enumerated-list element is displayed.

A *boolean item* is a form item that takes on the two values **TRUE** or **FALSE**. The feedback is a display of the keyword with the Boolean state video-inverted. The video-inverted Boolean means **TRUE**.



DMT

DMT is a tool whose purpose is to keep the phosphor on the display screen from wearing out. It should be run whenever you leave your workstation unattended.

1.1 Files

Retrieve **DMT.bcd** from the Release directory.

1.2 User interface

DMT is activated when you type DMT to the Executive. DMT then puts a solid black window on top of all of the existing windows. Embedded in this black window is a small white moving rectangle that shows the current date and time. Making DMT active does not affect any other processing already in progress; it merely covers up the display screen.

If DMT is running and you wish to resume work, you can deactivate it by pressing **ABORT** or by using the **Deactivate** or **Size** commands in the Window Manager menu.

DMT fails to achieve its purpose if your display is white-on-black; when run, it will display a solid white window covering the screen. Change it to black by pressing the **COMMAND-I** keys.

You select text by clicking **Point** within the selection. If you click **Point** in the same place several times within a brief period (within roughly a second), successive units of text are selected: clicking once selects a character, twice selects a word, three times a line, four times the whole body of text, and five times back to a single character. You can extend a selection to the left or right either by holding down **Adjust** while moving the mouse or by pointing to where the end point is to appear and pressing and releasing **Adjust**. The selection is extended in the same units used to make the original selection: a character selection is extended by characters, a word selection by words, and so on. A selection is extended by characters if you start over the first or last character of the selection and move the mouse while pressing **Adjust**. You can contract selections as well as expand them by using **Adjust**. If you **Adjust** to a place within the current selection, the selection shrinks by the units of the selection. However, if you begin the adjust action over either the first or last character of the selection, character mode is used instead. There will always be at least one unit left in any selection after contracting.

I.3.2.2 Entering text

Any characters typed into the window are inserted before the current type-in point. You can set the type-in point by moving the cursor to the desired place and clicking **Point**. The type-in point will be set as close as possible to the cursor's position. For example, when you select a single character, the type-in point precedes the character if you select the left half of the character and follows the character if you selected its right half. (Setting the **Balance Beam** in the **user.cm** file, described below, changes the positioning of the type-in point relative to the selection.)

The type-in point can also be set by holding down the **CONTROL** key and clicking the **Point** button over the desired location. This is useful with the **STUFF** command (see the section on Keyboard functions).

I.3.2.3 Deleting text

Text may be deleted by selecting it and pressing the **DELETE** key. Many tools place such deleted text into a global "trash bin." The **BS** (backspace) and **BW** (backward) keys delete text to the left of the current type-in point. Text deleted this way is not entered into the trash bin. The **BW** key deletes any white space or punctuation between the type-in point and the closest preceding word (alphanumeric string) and then deletes the word itself.

I.3.2.4 Current selection and trash bin

The *trash bin* is a conceptual container of the most recently deleted selection. In a subwindow that supports editing, the current selection may be deleted and deposited in the trash bin, where it is held for potential retrieval and placement. This allows text to be either moved from one position to another within a window or sent to subwindows other than the point of origin.

Any of the following steps copies text from one place in a window to another:

- Select the text, move the type-in point with **CONTROL-Point**, and press the **STUFF** key.

- Select the text, press **DELETE**, **PASTE** to move a copy into the trash bin, put the selection back where it was, move the selection to the desired location, and press **PASTE**.
- Set the type-in point to the desired target location, hold down **COPY**, select the text to be copied, and release **COPY** when finished selecting text.

I.3.3 Menus

A *menu* is a set of options or commands associated with a window or subwindow. Most windows have multiple menus. When the menus associated with a subwindow are displayed, the menus associated with its tool window are also displayed.

A menu contains either commands or data items. A menu command often takes the current selection as its argument. Sometimes, as with Window Manager commands, the semantics of the command implies its argument.

I.3.3.1 Invoking menus

In Figure I.5, the Window Manager menu is shown on top of the TextOps and File Window menus. This grouping of menus would probably be associated with a file window or text subwindow. Each type of window has specific types of menus associated with it. These menus are used to give commands to the process that owns the window.

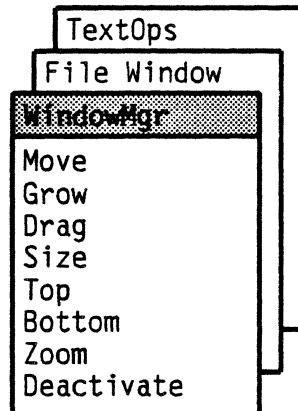


Figure I.5: Menus

Menus are invoked either by chording the mouse buttons or by pressing the **MENU** key (in the explanations below, the term "chording" will also stand for using the **MENU** key). Available menus appear in the vicinity of the cursor whenever (and as long as) you are chording. The position of the cursor determines which menus are available. If the cursor is in a subwindow, the menus associated with that subwindow and the menus associated with the tool to which the subwindow belongs are available. Some menus are available when the cursor is in any portion of the screen not covered by any window.

I.3.3.1.1 Choosing a menu

There are usually at least two menus for a window: the Window Manager menu (explained below), whose commands modify the window rectangle, and a menu that lists

the commands available for that tool. More menus are possible; subsequent menus underlie the others.

You can choose menus from the stack by positioning the cursor over the visible portion of the desired menu (the menu name frame) and chording again. When you chord again, the chosen menu appears on top of the others. Alternatively, as an accelerator, you may click **Point** over the title of the desired menu while continuing to hold down **Adjust**. The chosen menu immediately appears on top of the stack.

I.3.3.1.2 Invoking a command

Once a menu is displayed, choosing a menu item requires you to position the cursor over the list until it rests over the desired item, while you continue to chord. The selected menu item is video-inverted; when you release the chord, the command is invoked. If you release the chord when the cursor is not over a menu, the displayed menu disappears.

A quick method (called an accelerator), is to click **Point** over the desired menu item while continuing to hold down the **Adjust** key. The command is invoked; after it is executed the menu usually reappears.

Fine point: A menu does not reappear (1) if it is destroyed by the command invocation (such as by activating the only file in the Inactive menu), (2) if the source from which the command was invoked is no longer visible (as when invoking **Bottom** sends a window to the bottom of a stack, where it is completely obscured from view), or (3) if the window is tiny.

I.3.3.1.3 Confirming or aborting a command

Some menu commands require you to confirm or abort a command. In these cases the cursor changes to a tiny picture of a mouse with **Point** highlighted; this informs you that clicking **Point** will confirm the command. Clicking **Adjust** aborts a command.

I.3.3.2 Specific menus

There are several generally important menus: the Window Manager menu, the Inactive menu, the TextOps menu, and the Symbiotes menu.

I.3.3.2.1 Window Manager menu and accelerators

All tool windows allow you to manipulate window size, location, and state by using commands found in the Window Manager menu. For example, a window may be made to cover the entire available display space, change position, become smaller, turn into its iconic form, or disappear from the screen. The commands available in the Window Manager menu are:

Move	allows the window to be moved around the display area but does not change its size. When you invoke this command, the cursor changes into the shape of a corner bracket. As you move the cursor from one corner of the display area to another, it changes shape to indicate which corner of the window the operation will affect. When you position the cursor over the desired location and click Point , the window moves to the
-------------	--

area that begins in that corner.

Grow	allows you to pull a corner of the window in any direction, growing or shrinking the window along its width or height. This command acquires position information in the same way as Move .
Drag	allows you to elongate a window by pulling an edge of the window in any direction; it also requires position information.
Size	turns the window from a normal size into its tiny form, usually a small iconic rectangle showing an abbreviation of the window's name. If the window is already tiny, invoking Size changes it back to its normal size.
Top	displays the window on top of all the other windows in its stack.
Bottom	places the window at the bottom of all the windows in its stack.
Zoom	causes the window to grow, taking up all available display space and appearing on top of all other windows. Clicking Zoom again puts the window back to its previous size.
Deactivate	causes the tool window, and all other windows associated with a tool, to be removed from the display and become inactive. An abbreviation of the window's name is entered in the Inactive menu; the tool is re-activated by choosing the window name on the Inactive menu.

Window Manager operations may also be invoked more quickly by positioning the cursor in the left, middle, or right regions of the window name frame (or in the top half of a tiny window) and clicking one of the mouse buttons. The region of the window name frame in which the cursor is positioned video-inverts to provide feedback. The name-frame operations are:

<u>Mouse Button</u>	<u>Left Region</u>	<u>Middle Region</u>	<u>Right Region</u>
Point	Top/Bottom	Zoom	Top/Bottom
Adjust	Move	Size	Move

The operations available are as described above, with the exception of **Top/Bottom**. **Top/Bottom** specifies that if the window is not on top, move it to the top. If it is already on top, move it to the bottom. Pressing **Adjust** in the left or right portion of the name stripe brings up the **Move** cursor. Clicking **Point** while **Adjust** is still down cycles the cursor through the three shapes (**Move**, **Grow**, and **Drag**.)

These name-frame operations are also available on the upper half of a tiny window. In some tools, menu commands are available in the lower half of the window even when it is tiny.

I.3.3.2.2 Inactive menu

The Inactive menu contains a list of the tools that have been installed but are currently inactive. The Inactive menu is available in any part of the screen not covered by a window.

I.3.3.2.3 Text Ops menu

A text subwindow generally has a Text Ops menu that provides commands for manipulating text placement:

- Find** finds the next occurrence of the current selection in the subwindow. If the current selection is in the subwindow, the search begins at the end of the selection; otherwise, it begins at the first character visible in the subwindow. If the search is successful, the next occurrence of the text becomes the new selection. The search continues into text not visible on the screen; if the selection is found past the text displayed, the text is scrolled to the top of the split region. If no further instances of the text are found, the display blinks.
- If the **SHIFT** key is down, **FIND** works backward from the current selection, if any, or from the last character visible in the window.
- Split** divides a region of the subwindow into two subregions separated by a dashed line, with a small box at the right end of the line. This line can be moved by depressing **Point** over the small box, moving the cursor, and releasing the button. The subregions can be scrolled independently from each other. To remove the line, move it off the top or bottom of a region.
- Position** positions the text in the subwindow so that the character specified by the current selection, which must be a positive number, is at the top. For example, if you select 275 and invoke **Position**, the 275th character in the text is scrolled to the top of the subwindow.
- J. First** positions text in a window so that the first line of text is at the top of the window.
- J. Insert** positions the text in the subwindow so that the type-in point is at the top.
- J. Select** positions the text in the subwindow so that the line containing the leftmost character of the current selection is at the top.
- J. Last** positions text in a window so that the last line of text is at the top of the window.
- Wrap** reverses the current state of line wraparound in all the subwindows. When wrapping is on, a line that has not been terminated by a carriage return by the time it reaches the right edge of a subwindow is continued onto the next line. When wrapping is off, the same line disappears off the right edge of the subwindow.

I.3.3.2.4 Symbiotes and the Symbiote menu

A symbiote provides extra functionality for a tool window without requiring changes to the code of the tool or to Tajo itself. Using the Symbiote menu on the root window, you can attach a *symbiote* to any text window (Figure I.6). Symbiotes appear as subwindows that you can add to an existing tool dynamically, without disturbing its current processes or facilities. Symbiotes can be attached to any text or form window or subwindow.

In particular, the XDE provides a symbiote that adds editing capabilities to any text or form subwindow. (See the Editor Symbiote chapter for details.)

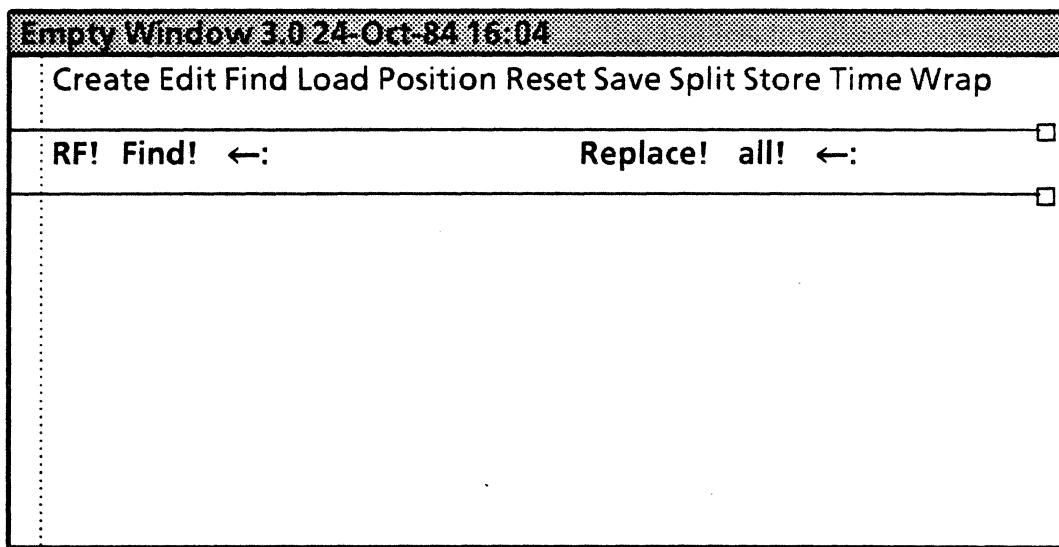


Figure I.6: Text window

The following commands are in the Symbiote menu, which is available in any part of the screen not covered by a window.

Attach Menu adds a one-line menu symbiote above a host subwindow after you have selected that target host subwindow with the cursor and pressed the **Point** mouse button to confirm the choice.

Detach Menu removes the menu symbiote above a host subwindow after you have selected that symbiote with the cursor and pressed the **Point** mouse button to confirm the choice.

Attach Edit Adds a one-line editor form above a host subwindow after you have selected that target host subwindow with the cursor and pressed the **Point** mouse button to confirm the choice.

Detach Edit removes the editor form from above that host subwindow after you have selected that symbiote with the cursor and pressed the **Point** mouse button to confirm the choice.

User.cm causes the system to reprocess the [FileWindow] section of the **User.cm** file to determine the default symbiote values.

I.3.4 Keyboard commands

The keyboard is made up of alphanumeric keys, special symbol keys, and special function keys. The function keys are referred to in this document by the names of their XDE functions, not their keycap names. The keycap name is also given below if it differs from the keyboard function name. The layout of the keyboard and the mapping from their keyboard names to their interface functions is shown in Figure I.7 (next page).

MENU	SCROLL-BAR	J.LAST J.FIRST	J.INSERT J.SELECT	reserved	client1	client2	DEFAULTS
------	------------	-------------------	----------------------	----------	---------	---------	----------

COMPLETE	! 1	@ 2	# 3	\$ 4	% 5	- 6	& 7	*	(8 9) 0	-	+	=	BW BS
TAB	Q reserved	W move	E expand	R replace	T define	Y	U undo	I invert	O	P	{ [}]	RET
LOCK	A stuff	S delete	D find	G	H	J j.select	K next-del	L	:	:	"	+	-	
SHIFT	Z	X doit	C copy	V paste	B	N next	M	<	>	?	/		SHIFT	
SPACE														

COMMAND +

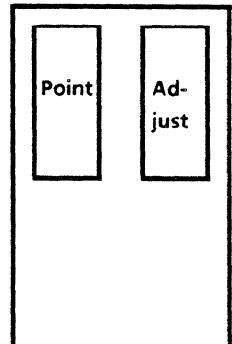
C =>	COPY
D =>	DELETE
E =>	EXPAND
F =>	FIND
I =>	SCREEN INVERT
J =>	J.SELECT
K =>	NEXT-DEL
N =>	NEXT
Q =>	RESERVED
R =>	REPLACE
S =>	STUFF
T =>	DEFINE
U =>	UNDO
V =>	PASTE
W =>	MOVE
X =>	DOIT
1 =>	J.FIRST
5 =>	J.INSERT
9 =>	J.LAST
ABORT =>	CLEAR USER ACTION BUFFER (ASYNCHRONOUS)
COMPLETE =>	AGAIN

Left function group

AGAIN	REPLACE DELETE
FIND	COPY
PASTE	MOVE
STUFF	CONTROL

NEXT	HELP	UNDO
NEXT-DEL	DOIT	
DEF'N	CALL DEBUG	
EXP'D	COMMAND	ABORT
	T	

Right function group



Mouse buttons

Keyboard configuration using Level IV hardware

Double inscription on function keys indicates use of Shift (i.e., SHIFT + BS = > BW)

client 1,2 reserved for client definition

I.3.4.1 Keyboard functions

The keyboard functions are:

ABORT	sets an abort "flag" in the window containing the cursor. A running tool checks periodically to see whether an abort flag has been set. If it has, the tool aborts itself. If you press ABORT a second time before the flag in a window is reset (i.e., turned off), a global abort flag is set and all tools abort. The window's abort flag is reset when anything is typed into the window except SHIFT or ABORT . The global abort flag is reset whenever the abort flag is reset in any window.
AGAIN	replaces the selection with the last text that was typed or stuffed.
CALL DEBUG	(SHIFT-ABORT) calls the debugger. If both shift keys are held down when invoking it, a panic call is made to the debugger. Panic calls should only be made in dire emergency, since calling procedures out of the debugger interpreter may not work.
COMMAND	is a shift key used with other keys to invoke various functions.
COMPLETE	treats the token to the left of the type-in point as the beginning of a file name and attempts to complete the name. This function is currently implemented only by the Executive.
CONTROL	is a shift key used with other keys. Used with Point , it moves the type-in point without changing the current selection.
COPY	clears the current selection and maintains the type-in point while the key is held down, thus allowing a new selection to be made. When the key is released, that new selection is stuffed into the window at the type-in point.
DEFINITION	(SHIFT-EXPAND) puts the current selection into the expansion field of the Dictionary Tool. (See the Dictionary Tool chapter.)
DELETE	deletes the selected text, replacing the contents of the trash bin with the deleted text.
DOIT	is a client-specific function. In a file window, it causes the window to be loaded from the token in the window, using the token as a file name. (If there is no such file, it tries to append each of the extensions .mesa , .config , and .cm until it finds a match.)
EXPAND	replaces the alphanumeric token to the left of the type-in point by its expansion, as defined by the current dictionary (see the Dictionary Tool chapter).
HELP	invokes the subwindow Help function, if there is one.



Dictionary Tool

The Dictionary Tool allows you to expand abbreviations according to a user-defined dictionary, called the Edit Dictionary, and to add abbreviation-expansion pairs to the dictionary.

2.1 Files

The Dictionary Tool is built in; no additional files are needed. The default name for the Edit Dictionary on your system is **default.dict**.

2.2 User interface

The Dictionary Tool implements the **EXPAND** and **DEFINITION** function keys in text and form subwindows. (See the section on keyboard functions in the User Environment chapter for descriptions of the **EXPAND** and **DEFINITION** keys.)

The **EXPAND** function treats the word to the left of the insertion point as an abbreviation and looks it up in the dictionary, ignoring case. If an entry is found, the abbreviation is replaced by the definition. If the definition contains fields, the field is selected. The abbreviation may be a unique prefix of the abbreviation-expansion pair.

The **DEFINITION** function invokes the Dictionary Tool. If the Dictionary Tool is already active, it deactivates it.

2.3 Dictionary Tool

The Edit Dictionary is maintained by the Dictionary Tool. It contains one or more files, each of which is a list of abbreviation-expansion pairs. The Dictionary Tool is invoked by the **DEFINITION** key or by standard window manager methods.

The Dictionary Tool interacts through a message subwindow, a form subwindow, and a log subwindow. The message subwindow is used to post error messages. The form subwindow is used to invoke commands and provide parameters. The log subwindow is used to record the results of commands.

The Dictionary Tool maintains its dictionary in memory in a format that allows fast lookup of expansion strings, given the abbreviation. There is no limit to the number of entries in this dictionary. The dictionary may be initialized by loading .dict files that contain abbreviation-expansion pairs in human-readable and -editable form.

2.3.1 Commands

The form subwindow has the following layout:

Record! **LookUp!** **List!** **Load!** **Store!** **Dictionary:**
Abbreviation:
Expansion:

- Record!** enters a pair in the dictionary with abbreviation **Abbreviation:** and expansion **Expansion:**. If **Expansion:** is empty, the current abbreviation-expansion pair is deleted.
- LookUp!** fills in **Expansion:** with the expansion of the abbreviation **Abbreviation:**.
- List!** writes all the pairs in the dictionary to the log subwindow.
- Load!** reads the pairs in the .dict file specified by **Dictionary:** and loads them into the dictionary.
- Store!** stores the pairs in the dictionary onto the .dict file specified by **Dictionary:**.

If the dictionary is modified by recording new entries or by loading a new .dict file, the modifications are not stored in the .dict file unless the **Store!** command is invoked or the **StoreOnDeactivate User.cm** entry is included (see below).

2.3.2 File format

An entry in the .dict file has the following format:

abbrev:<TAB> "expansion string"<CR>.

The double quotes around the expansion string are optional if it does not contain any embedded returns. The expansion string should not contain any double quotes.

2.4 User.cm

Two entries are implemented:

[DictionaryTool]

Dictionary: My.dict

Initializes the dictionary from the specified .dict file.
Default.dict is used if there is no **User.cm** entry.

StoreOnDeactivate: TRUE

Automatically stores the dictionary when the tool is deactivated to the specified .dict file if the dictionary has changed.

FIND	finds the current selection in the window containing the cursor. SHIFT-FIND looks backward, either from the current selection, if the current selection is in that window, or from the bottom of the window, otherwise.
J.FIRST	positions the text in a subwindow so that its first line at the top of the subwindow.
J.INSERT	(SHIFT-SELECT) positions the text in the subwindow so that the type-in point is at the top.
J.LAST	(SHIFT-J.FIRST) positions the text in a subwindow so that its last line is at the top of the subwindow.
J.SELECT	positions the text in the subwindow so that the line containing the leftmost character of the current selection is at the top.
MENU	brings up the menus in the subwindow containing the cursor; it is the same as chording the mouse buttons.
MOVE	is like COPY , except that the selection is deleted after it has been stuffed into the window containing the input focus.
NEXT	advances the cursor either to the next field in a form subwindow or to the next bracketed field in a text subwindow, setting the type-in point to that field.
NEXT-DEL	like NEXT , only it deletes the contents of the field before setting the type-in point.
PASTE	takes the contents of the trash bin and inserts it at the type-in point. It is like STUFF , only it operates on the contents of the trash bin.
REPLACE	(SHIFT-DELETE) is like DELETE , but it changes the type-in point to the point from which the text was deleted.
STUFF	takes the current selection and copies it to the type-in point of the subwindow that is currently taking type-in. If no window contains the input focus, this action fails and the display blinks.
UNDO	swaps the selection with the trash bin.

I.3.4.2 Global functions

Various keys invoke functions that affect the development environment globally or affect the tool that is in the process of performing a user-initiated action. These functions are available regardless of where the cursor is positioned:

COMMAND-I	inverts the display to white-on-black or black-on-white, whichever is the opposite of what it currently is.
------------------	---

COMMAND-ABORT causes the development environment to forget all buffered user actions that have not yet been processed, such as type-ahead.

These commands work only in text subwindows:

COMMAND-L	sets the case of the characters in the current selection to upper case if the SHIFT key is down, or to lower case if it is up.
COMMAND-<	brackets the selection on the left by < and on the right by >.
COMMAND-[brackets the selection on the left by [and on the right with].
COMMAND-{	brackets the selection on the left by { and on the right with }.
COMMAND -(brackets the selection on the left by (and on the right with).
COMMAND -"	surrounds the selection with quotes.
COMMAND --	surrounds the selection by the "--" comment delimiter.

I.4 The user command file

The **user** command file, **User.cm**, is a file on the current volume used to set defaults for a user. Many subsystems and tools pick up the information from the **User.cm** file to initialize various options, such as font information, window placement and size, and where to send files to be printed. Some **User.cm** values are used at user login; others when a tool is activated.

To create a **User.cm** file for yourself, retrieve **SampleUser.cm** from **Docs >** onto your Tajo and CoPilot volumes, edit it to contain such information as your name and domain by replacing the fields all currently delimited by angle brackets, and rename it to be **User.cm**.

I.4.1 Format of the user command file

A **User.cm** section consists of a section title in brackets, followed by a carriage return, and the entries for that section. Each entry is on a separate line. Entries consist of **Name:** followed by the value. Any line that begins with **--** is ignored.(Here, as in several other types of files, text preceded by **--** is treated as comments and not processed.)

It is possible to have volume-specific entries for the values in a section when, for example, you need different defaults in your CoPilot and Tajo volumes to determine which tools get loaded at initialization time. This is specified by putting **[Volume:SectionName]** as a title. The section entries in the volume-specific sections override those of the generic sections when the volumes are booted.

Note: There are no spaces before or after the colon in a section title name, but all entries must have a value after the colon.

In the example below, [FileWindow] is the generic section title. The menu line in the FileWindow section in the CoPilot volume has *Break* in the menu line, but it is not needed in the Tajo volume.

```
[FileWindow]
SymbioteSetUp: Always Menu Edit

[CoPilot:FileWindow]
Menu: Break Edit Load Reset

[Tajo:FileWindow]
Menu: Edit Load Reset
```

The development environment processes the [System], [Librarian], and [FileWindow] sections of the **User.cm** at start-up time; all other sections are processed when the corresponding tool is run. You should ensure that your **User.cm** file, as well as any files needed in the processing of these sections, are in your top-level directory, since the initial search path may not be set while these sections are processed. This is most likely to be a problem when processing the **InitialCommand:** entry.

Below are examples of [System] entries. You can edit many of these values with the Profile tool while the system is running (see the Profile Tool chapter).

User: CSmythe

This is your user name.

Domain: Bayhill

This is the default domain section of your clearinghouse name, used in authenticating who you are, for accessing network services like printing.

Organization: Xerox

This is the default organization section of your clearinghouse name, similar to the default domain section.

InitialCommand: Run.~ Editor.bcd

This is an executive command line to be executed as part of the boot sequence. You cannot have any carriage returns in the command line. The log file for this command is **Initial.log**. Feedback will appear in the Herald window as a result of executing commands in this line.

Font: LaurelFont.strike

A font is built in; provide this entry only if you want to override the default.

MenuFont: Helvetica7.strike

This is the font used for menus; a default font is built in.

Debug: FALSE

This sets the debugging variable for the system. The default value is **FALSE**. Certain bugs call the debugger if this is **TRUE**. Otherwise, the system ignores the error and attempts to work around it.

Screen: White

This determines the background color of the display. The default is **White**; **Black** is the alternative.

SwapControlAndCommand: FALSE

This swaps the functions of the control and the command keys, which is especially useful on a microswitch keyboard because the command key is awkward to use.

SearchPath: <Tajo>Temp <Tajo>

This is the intitial value of the file system search path.

BalanceBeam: Always

This sets the value of the variable that controls positioning of the type-in point relative to a selection. It has three possible values:

Always: the type-in point is as close as possible to the cursor position.

Never: the type-in point is at the end of the selection.

NotForCharacter: the type-in point is after a single character selection, but it will be as close as possible to the cursor posititon for multiple character selections.

FileWindow: [x: 512, y: 30, w: 512, h: 439] [x: 900, y: 778] Calendar/t

An arbitrary number of FileWindow entries is permitted in the System section. Each specifies a file window to be created. The first set of bracketed values indicates the position of the window when it is active. **x** and **y** are the horizontal and vertical bitscreen coordinates of the upper-left corner of the window. **w** and **h** are the width and height of the window in bitscreen coordinates. Any or all of these fields may be omitted, in which case they have the following default values: [**x**: 0, **y**: 0, **w**: 512, **h**: 400]. The second set of bracketed values indicates the position of window when it is tiny. **x** and **y** are the horizontal and vertical bitscreen coordinates of the upper-left corner of the window. Any or all of these fields may be omitted, in which case they have the following default values: [**x**: 0, **y**: 0]. The next item in the line, which is optional, is the name of the file to be loaded into the window. If there is a switch on the file name, it specifies the initial state of the window (**a** for active, **t** for tiny, and **i** for inactive). *You must always specify the active box and tiny box position, even if they are defaulted, by specifying []*.

I.5 Documentation roadmap

This section describes how the XDE documentation is structured and where to look to find information about a particular subject. The documentation for this system, written for system developers who are familiar with the Mesa programming language, consists of five separate manuals: *XDE: Concepts and Principles*, the *XDE User's Guide*, the *Mesa Language Manual*, the *Pilot Programmer's Manual*, and the *Mesa Programmer's Manual*. This manual, the *XDE User's Guide*, describes the tools that make up the programming environment. Its introductory chapters contain general information on getting started and how to use the environment. The *Mesa Language Manual* is a reference manual for the programming language. The *Pilot Programmer's Manual* and the *Mesa Programmer's Manual* are reference manuals that describe Pilot and Mesa client interfaces. The *Pilot Programmer's Manual* describes operating system facilities, while the *Mesa Programmer's Manual* documents the software interfaces that implement user-interface functions.

I.5.1 XDE: Concepts and Principles

The *XDE Concepts and Principles* guide introduces the Xerox Development Environment. It describes the organization of the system broadly, focusing on the metaphors and theories the developers had in mind when they built the system. It discusses each of the parts of the system and explains their interaction.

I.5.2 The XDE User's Guide

If the development environment is new to you, read the *XDE Concepts and Facilities* manual. Along with this introductory chapter of the *XDE User's Guide*, it tells you how to get started, gives information about programming in the development environment, and describes the user interface.

Most of the remaining chapters of the *XDE User's Guide* (this document) describe the tools, which are utility programs that run in the development environment. The tools are grouped according to their function. Each one is described in a separate chapter containing information about the user interface for the tool, examples of how to use it, an explanation of error messages, and background information necessary to understand how the tool operates. This *XDE User's Guide* is best used to develop the "hands-on" knowledge you need for accomplishing programming tasks. It is also a reference manual for using tools.

I.5.3 Mesa Language Manual

The *Mesa Language Manual* is a reference manual defining the Mesa programming language. It explains how to use the Mesa language, with examples, and describes the grammar that defines Mesa.

I.5.4 Pilot Programmer's Manual

The *Pilot Programmer's Manual* is intended for designers and implementors of client programs of Pilot. It describes the external structure and interfaces of Pilot, the operating system, and the other packages released with it, providing sufficient information for programmers to understand the facilities available and to write procedure calls in the

Mesa language to invoke them. Similar to the *Mesa Programmer's Manual*, the *Pilot Programmer's Manual* documents procedures, parameters, results, data types, and signals for each Pilot software interface.

I.5.5 Mesa Programmer's Manual

The *Mesa Programmer's Manual* describes the collection of interfaces that provide a framework and runtime system for writing Mesa programs in the development environment. For each interface, the *Mesa Programmer's Manual* lists all procedure names, parameters, results, arguments, data types, and signals. The interfaces documented in the *Mesa Programmer's Manual* implement and support the window-oriented user interface available for use in tool writing.

I.5.6 Appendices

Appendix A of this document describes Othello. Appendix B describes procedures for getting started in the Xerox Development Environment.

In the *Mesa Programmer's Manual*, Appendix A discusses the Example Tool, a tool that helps you learn about tools. Appendix B contains information about interfaces.

I.6 Typographical conventions

The typographical conventions in this document are as follows:

Keycap and mouse button names are **MODERN 8 BOLD CAPS**.

Commands are **Titan 10 bold**; file names, menu items, and switches are **Titan 10**.

Interaction with the system is represented in **Titan 10**. When an example is given, what you are required to type is underlined (with the exception of the special symbol for the carriage return key). A ***** indicates that you should press the carriage return key.

I.7 Other features, other tools

Some of the other useful features of the Xerox Development Environment are within the General tools described in the rest of the chapters in this section. These tools affect processes system-wide, so they can help you to work more efficiently in many situations.

I

General Tools

Editor Symbiote

The XDE 3.0 Editor provides a way to edit files stored on disk as well as to create new files. This screen-oriented editor, which includes an extensive and powerful pattern-matching facility, can be associated with any text or file window (or subwindow).

3.1 Files

The Editor Symbiote is included in the boot files.

3.2 User interface

The editor interfaces with users as a symbiote that attaches to any text subwindow or form subwindow. The Editor Symbiote can be invoked via the Editor menu associated with the Root subwindow. The editor is loaded with the boot files when CoPilot is booted.

The Editor Symbiote's user interface is described below.

3.2.1 Editor menu

To use the Editor Symbiote, chord on the mouse to get the Symbiote menu from the root window. **Attach edit** will attach an Editor Symbiote subwindow to a host text or form subwindow, and **Detach edit** will remove it. (Note that the Editor Symbiote commands will work on form subwindows.)

3.2.1.1 Editor Symbiote subwindow

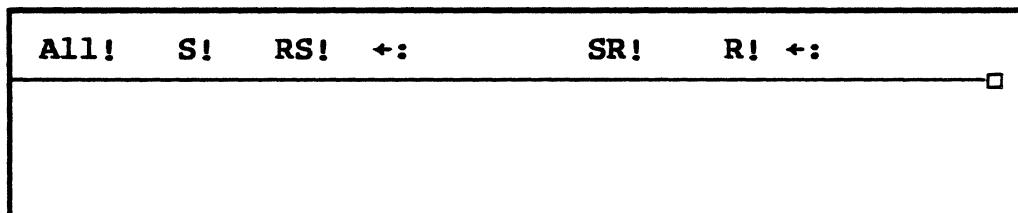


Figure 3.1: Editor Symbiote subwindow

The Editor Symbiote is a form subwindow with the following items. (The behavior of the Editor Symbiote menu items is affected by the Editor property sheet, as explained in the next section.)

- ←: The *search field*--the text that will be searched for (the ←: following **RS!**). This field may contain expressions specifying variable patterns to be matched.
- S!** Searches for text matching the search field. The search starts immediately following the current selection *if it is visible in any split of the window*; otherwise, the search starts from the first character visible in the top split of the window.
- ←: The *replace field*--the text that will replace the selection (the ←: following **R!**). This field may also contain variables denoting elements of the search field.
- R!** Replaces the current selection with the text specified by the replace field. If the current selection was set as the result of **S!** or **RS!**, the expression in the search field is available for replace-field variables. If the selection was set some other way, the replace field may only have literal text and may not contain any variables.
- RS!** Does an **R!** followed by an **S!**, thus replacing the current selection and searching for the next matching text.
- SR!** Does an **S!** followed by an **R!**, thus searching for the next matching text and replacing it.
- All!** Repeatedly does an **SR!**, thus replacing all text instances that match the search field. The repetition stops when the search fails to find a match.

For more information about the Editor Symbiote's search and pattern-matching facilities, see the section on Search and pattern matching.

If you press the **DOIT** key (**MARGINS**) when an Editor Symbiote has the input focus, the Editor Symbiote subwindow grows to two lines, with **All!**, **S!** and **RS!** on the top line and **SR!** and **R!** on the second line, giving more space to enter text. This two-line format is also useful for comparing search and replace strings, which may be quite simple or very complicated. Pressing the **DOIT** key again returns the symbiote subwindow to its original one-line configuration.

If the search field is empty when you invoke **S!**, the Editor Symbiote copies the current selection into the search field before starting the search.

3.2.1.2 Editor property sheet

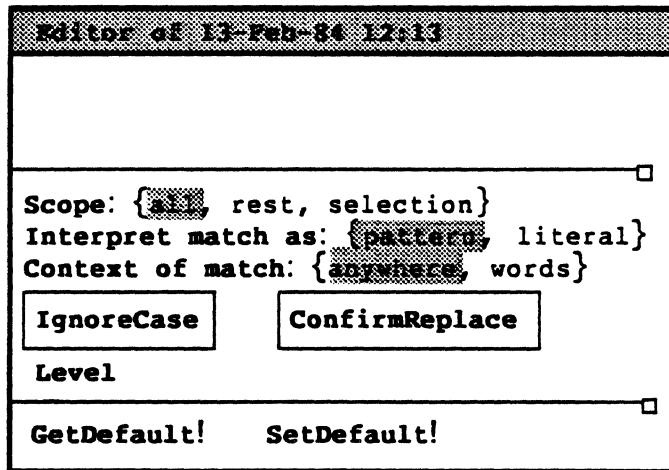


Figure 3.2: Editor property sheet

The Editor property sheet is a separate window named Editor. Its fields, which affect the Editor Symbiote's operation, are:

Scope: {all, rest, selection} specifies the scope of the **All!** command. **all** means the entire file, **rest** means "the rest of the file"--just like the **S!** command (*q.v.*)--and **selection** means "within the current selection."

Interpret match as: {pattern, literal} specifies the interpretation of the text in search field. **pattern** means to interpret the search field as a regular expression; **literal** means to use the search field as simple literal text.

Context of match: {anywhere, words} further limits the acceptable context in which a search may find a match. **anywhere** means that the pattern can match within a larger word. **words** only matches patterns that are surrounded by non-alphanumeric characters.

Ignore Case is a Boolean that will cause upper-/lower-case differences to be ignored during a search.

Confirm Replace is a Boolean that will cause the Editor Symbiote to request explicit user confirmation for each text replacement. A confirm cursor appears when confirmation is requested; use **Point to confirm**, **Adjust** to deny.

Level: is the number of space characters by which the indenting should be adjusted. This is used by the **Nest** and **UnNest** commands in the Edit Ops menu.

The property sheet also has a command subwindow with these commands:

- | | |
|--------------------|---|
| GetDefault! | sets the editor properties to the built-in default state. |
| SetDefault! | sets the default editor properties to be those currently set in the property sheet. GetDefault! may then be used to return the properties to that state. |

3.2.1.2.1 Editor property sheet accelerator

You can associate the Editor property sheet with any key on your keyboard for faster access to the editor's parameters. If the text subwindow TIP Interpreter sees the atom "Editor," it will make the Editor property sheet appear (become active if it is inactive, or normal if it is tiny). To associate the Editor property sheet with the **HELP** key, you would use the following entry in the <>TIP>TextSW.TIP file:

```
SELECT TRIGGER FROM
.
.
.
HELP Down => Editor;           --specifies which key to attach to
.
.
.
ENDCASE...
```

To get the TextSW.TIP file, look on the <Hacks>lx.0>Source>Editor> directory. It can be copied to the local file <>TIP>TextSW.TIP. After installing the file and rebooting, pressing the **HELP** key causes the Editor property sheet to appear.

3.2.1.3 EditOps menu

When an Editor Symbiote is attached to a subwindow, an EditOps menu is also placed on the window. The **All**, **Search**, **SearchReplace**, **ReplaceSearch**, and **Replace** menu items invoke the same commands as the Editor Symbiote's **All!**, **S!**, **SR!**, **RS!** and **R!** commands. Other menu commands, which only operate on text subwindows, are specific to formatting Mesa source code. They are:

- | | |
|---------------|---|
| Nest | shifts the lines that contain the current selection level characters to the right, where level is specified in the Editor property sheet. |
| UnNest | shifts the lines that contain the current selection level characters to the left, where level is specified by the Editor property sheet. |
| Match | identifies matching parentheses (), square brackets [], angle brackets < >, and braces { }. If one of these grouping characters is selected, Match extends the selection to the matching character. If a character that is not one of these is selected, Match extends the selection in both directions |

until it contains a match. Successively using **Match** will match larger scopes.

Count

gives a count of how many occurrences of a pattern are found in the text. The search expression and scope are specified in the Editor property sheet. The result is given in the message subwindow of the Editor property sheet.

3.3 Search and pattern matching

3.3.1 Search

The search operation accepts expressions in the search field. You can search for patterns or families of strings, as well as for simple literal strings. The syntax of a search expression is given below. First, some preliminary definitions:

<char>

a single literal character. Since the characters #, \$, [,], ~, *, and \ have special meaning within a search expression, you must prefix these characters with a backslash character. For example, * means a literal asterisk character. Following Mesa conventions, you may also use \n for carriage return, \t for tab, \ddd for the character whose code is octal ddd, where d is an octal digit and ddd \leq 377B. Escaping an ordinary character is harmless.

<char1>-<char2>

character range. For example, A-Z means all the capital letters.

<character class>

a set of characters, defined by naming the characters to be included. A character class specification consists of a sequence of characters and character ranges.

A *search expression* is an arbitrary sequence of the following five elements. Each element counts as one "variable" in replace expressions.

<string>

matches the given literal characters of the string.

#

matches any single character.

\$

matches the beginning of a line (for use when one is the first element in the pattern).

[<character class>]

matches any character in the character class.

[^<character class>]

matches any character *except* those in the character class.

In addition, any of the above five constructs can be qualified by appending either of the following *closures*, which are explained in the section on Character classes and closure. When a closure is applied to a <string>, it applies only to the last character of the string.

* short closure. Matches the least possible number, including zero, of occurrences of the previous construct.

** long closure. Matches the greatest possible number, including zero, of occurrences of the previous construct.

3.3.2 Replace

The replace field specifies the text that will replace the selection in a replace operation. This field may also contain an expression with variables denoting elements of the search field.

A *replacement expression* is an arbitrary sequence of the following elements.

<string> replaces with the given literal characters of the string. Since the character @ has special meaning within a replacement expression, you must prefix this character with a backslash character; e.g., \@.

@& replaces with the complete text found by the search.

@n@ replaces with the text that matched the *n*th element of the search expression. The first element of the search expression is "1," etc.

3.3.3 Character classes and closure

Character classes provide a way to match different characters as part of a pattern. For instance, either [a-c] or [abc] is a proper character class declaration that will match any of the letters a, b, or c. Usually, however, you will not want to match just a single character in a character class, but a word or a list of them. The short closure * and the long closure ** are used for this. * and ** match with zero or more members of the search expression element that immediately precedes the closure. * matches the shortest possible string of the pattern type, and ** matches the longest possible string. So an expression like [a-c]* will match strings of arbitrary length whose component letters are a, b, and c.

For example, given the text "Hello.bcd Goodbye.bcd":

H#*.bcd will match "Hello.bcd"

H#**.bcd will match "Hello.bcd Goodbye.bcd"

Caution: Be careful about using #* and #** if you are editing a large file,. Since # matches any character, #* and #** will be slow. Since #** matches the longest run of characters, it will be *very* slow.

3.3.4 Examples

1. To find words that start with an upper-case letter:

Find: [A-Z][a-z]**

Result: 'I', 'Hello', 'Prince' will all match, 'warthog' will not.

2. To find a word whose

first character is either a, b, c, d, s, x,y , z
 second character is either a, e, i, o, u
 third character is g, p, 4, 5, 6
 and reverse the order of the letters found:

Find: [a-dsx-z][aeiou][gp4-6]
Replace: @3@@@2@@1@
Result: dog => god

3. To delete the leading zeroes from numbers

Find: [~0-9][0]**[0-9]
Replace: @1@@@3@
Result: 000000B => 08, 00343B => 343B

4. To generate exec commands from a list of files (also see the example given in the next section):

Input: "Access.archivebcd Adobe.archivebcd Binder.archivebcd "
Find: #*
Replace: Copy <>Temp>@1@ ← @1@@n
Result:
 Copy <>Temp>Access.archivebcd ← Access.archivebcd
 Copy <>Temp>Adobe.archivebcd ← Adobe.archivebcd
 Copy <>Temp>Binder.archivebcd ← Binder.archivebcd

3.3.5 Editor as programmer's tool

The searching and pattern matching facilities of the editor can be used as a macro to generate sizeable chunks of code in a very short time, as in the following example:

Suppose you want to create a function that sends out simple error messages if there is an error while attempting to access a file. Because Mesa has such unique type-definition capabilities, you are likely to find an enumerated type such as **MFile.ErrorCode** lying around, a type that enumerates the different possible file access errors. Using the members of this type as a list of selection keys, you can trivially generate code that will send the name of the file access error message to your terminal. What follows is a dialog for doing just that.

First, you will want to get a list of all the error codes. Type the following command to the Executive window:

```
>Show type: MFile.ErrorCode
```

```
MFile.ErrorCode: TYPE = MACHINE DEPENDENT {noSuchFile, conflictingAccess,
insufficientAccess, directoryFull, directoryNotEmpty, illegalName,
noSuchDirectory, noRootDirectory, nullAccess, protectionFault,
directoryOnSearchPath, illegalSearchPath, volumeNotOpen, volumeReadOnly,
noRoomOnVolume, noSuchVolume, crossingVolumes, fileAlreadyExists,
```

```
fileIsRemote, fileIsDirectory, invalidHandle, courierError, addressTranslationError,
connectionSuspended, other(255)};
```

The list below was simply copied from the Executive window into an empty File window (using the copy key):

```
noSuchFile, conflictingAccess, insufficientAccess, directoryFull,
directoryNotEmpty, illegalName, noSuchDirectory, noRootDirectory, nullAccess,
protectionFault, directoryOnSearchPath, illegalSearchPath, volumeNotOpen,
volumeReadOnly, noRoomOnVolume, noSuchVolume, crossingVolumes,
fileAlreadyExists, fileIsRemote, fileIsDirectory, invalidHandle, courierError,
addressTranslationError, connectionSuspended
```

Now attach an Editor Symbiote subwindow to the File window and make the following entries into the find and replace fields (<->):

```
Find: #*,  
Replace: @1@ => Write["@1@"L];\n
```

Running that Replace function (R!) over the list above and adding the PrintError subroutine name and the SELECT statement yields the finished function below:

```
PrintError: PROC[code: MFile.ErrorCode] = {
    SELECT code FROM
        noSuchFile => Write["noSuchFile" L];
        conflictingAccess => Write["conflictingAccess" L];
        insufficientAccess => Write["insufficientAccess" L];
        directoryFull => Write["directoryFull" L];
        directoryNotEmpty => Write["directoryNotEmpty" L];
        illegalName => Write["illegalName" L];
        noSuchDirectory => Write["noSuchDirectory" L];
        noRootDirectory => Write["noRootDirectory" L];
        nullAccess => Write["nullAccess" L];
        protectionFault => Write["protectionFault" L];
        directoryOnSearchPath => Write["directoryOnSearchPath" L];
        illegalSearchPath => Write["illegalSearchPath" L];
        volumeNotOpen => Write["volumeNotOpen" L];
        volumeReadOnly => Write["volumeReadOnly" L];
        noRoomOnVolume => Write["noRoomOnVolume" L];
        noSuchVolume => Write["noSuchVolume" L];
        crossingVolumes => Write["crossingVolumes" L];
        fileAlreadyExists => Write["fileAlreadyExists" L];
        fileIsRemote => Write["fileIsRemote" L];
        fileIsDirectory => Write["fileIsDirectory" L];
        invalidHandle => Write["invalidHandle" L];
        courierError => Write["courierError" L];
        addressTranslationError => Write["addressTranslationError" L];
        connectionSuspended => Write["connectionSuspended" L];
    ENDCASE;
};
```

3.4 User.cm file entries

The typical Tajo tool parameters can be set for the Editor property sheet under [Editor] in the **User.cm** (i.e., WindowBox, InitialState, TinyPlace).

[Editor]

WindowBox: <put here the size of window box you prefer>

InitialState: <put here the initial state you want, particularly Tiny or Active>

TinyPlace: <put here the coordinates of the desired location of the Tiny window on your screen>

In particular, fix the **User.cm** entry for **[FileWindow]** to "Setup: Always Menu Edit" to get the Editor Symbiotes to attach themselves by default to text windows.

[FileWindow]

Setup:Always Menu Edit



Executive

The Executive is a tool for loading and running Mesa programs.

4.1 Files

The Executive is built into Tajo and CoPilot; no extra files are needed.

4.2 User interface

The Executive runs as a TTY window, so the standard editing functions are not available. The insertion point is always at the end of the text and cannot be moved elsewhere in the Executive window. In the following descriptions, word refers to a sequence of alphanumeric characters; token refers to a sequence of non-blank characters.

4.2.1 Editing functions

The Executive interprets certain characters as editing characters on the current command line, as follows:

BS	deletes the last character.
BW	deletes the previous word; any non-alphanumeric characters to the right of the previous word are also deleted.
CONTROL-X	expands the command line (defined below) and prints the expanded command line.
CONTROL-C, DELETE	aborts the current command line and prompts for a new command.
COMPLETE	treats the last token on the command line as the beginning character string of a file name or registered command and attempts to complete it. If the token starts more than one file name or command, the screen flashes. The Executive extends the command line with as many unambiguous characters as it can.

TAB	treats the last token on the command line as the beginning character string of a file name and lists all files or registered commands it starts. The token is deleted from the command line and the command line is retyped.
?(question mark)	treats the last token on the command line as the beginning character string of a file name and lists all files or registered commands it starts. The token is not deleted from the command line and the command line is retyped.
RET, :(semicolon)	terminates the command line. RET terminates the command line and causes it to be interpreted, while the semicolon permits more command lines to be typed before interpretation begins.

4.2.2 Command line expansion

The Executive expands a command line using the following for these special interpretation characters:

' (single quote)	quotes the following character so that the Executive does not interpret it. The following character, but not the quote, becomes part of the expanded command line. For example, use a single quote to pass a semicolon in a command line to the Compiler.
↑ (UpArrow)	quotes the following character so that the Executive will not interpret it. Neither the UpArrow nor the following character is part of the expanded command line. ↑ is typically used to insert carriage returns into long command lines to make them more readable.
* (star)	interprets the token containing the star as a pattern; replaces this token by the list of files and registered commands that match the pattern. The * in the pattern may match zero or more instances of a character. A single star only matches within one level of subdirectory, that is, it will not match the character > in a file name. Multiple stars will cross subdirectories. Hence, the pattern * matches all the files in the current subdirectory, while the pattern ** matches all the files in or below the current subdirectory.
#(pound sign)	same as *, but matches only one character.
@ (at-sign)	interprets the following token as a command file. The token may be terminated by another at-sign, by a space, a CR, or a semicolon. The token is interpreted as the name of a file, and the token is replaced by the contents of that file. If the token is not a file name, the Executive tries to complete it by appending .cm. If that fails, it appends *.cm, and if that fails, it prompts you for the contents of the file.

\\" (backslash) or -- denotes the characters that follow as a comment. The comment can be terminated by a matching pair of delimiters (\\" or --) or by >>.

4.2.3 Command line interpretation

The Executive assumes that the first token in a command line is the unique prefix of one of its registered commands. (Commands may be registered by programs.) If the first token is the prefix of more than one command, the Executive reports that it cannot find the subsystem and prompts for a new command, discarding all previous input.

If the first token is not the prefix of any command, the Executive assumes that there is a program that would register that command if it were run. The Executive attempts to find and run a likely program. First, it checks to see if the token is the name of a file. If not, it strips any extension from the file and appends the following suffixes: .archivebcd, *.archivebcd, .bcd, *.bcd. If any of these patterns match exactly one file, the Executive runs that program. After running the program, the Executive checks to see whether the program has registered the command that should have been present to correspond with the first token on the command line. If not, it skips the current command line and starts processing the next command.

4.2.4 Built-in commands

The commands listed below are built into the Executive and are automatically loaded and started when the Executive is created. Some of the built in commands take file names or directory names for arguments.

AliasCommand

AliasCommand OldCommandName NewCommandName

provides a mechanism for giving a particular command more than one name. Subsequent invocations of the command by its original name or any of its aliases will always invoke the same procedure that was registered with the original command. This is useful for commands which have identical beginning letters, such as Compare and Compiler, since the user must enter at least five letters of either command in order for command completion to work.

ClientRun

has the same semantics as the Run! command of CommandCentral, except that its arguments come from the command line instead of the Run: line of CommandCentral. (Also see SetClientVolume.) For example, the following command runs the program Test1.bcd on the current client volume:

ClientRun Test1.bcd

CloseVolume

takes a list of volume names and closes each volume. The volume to be closed should not be on the current search

path (see the Search Path Tool chapter). The following command closes the logical volumes named Tajo and User.

CloseVolume Tajo User

ChangeCommandName

OldCommandName NewCommandName

is used for renaming commands registered with the Executive (not to be confused with **Rename**, which renames files). After executing **ChangeCommandName**, the operations previously invoked by typing **OldCommandName** to the Executive can only be started by typing **NewCommandName**; **OldCommandName** will no longer be registered.

Clearinghouse

prompts you for your domain and organization. An example of the use of the Clearinghouse is:

```
Clearinghouse
Domain: OSBU North
Organization: Xerox
```

Copy

expects an argument of the form

TargetFile ← file1 file2 ...

If the left arrow is omitted, the Executive asks the you to confirm that the first file is the target file. After the **Copy** command, the target file will contain the concatenation of the contents of the source files. If there is only one source file, the target file will have the same creation date as the source file; otherwise, it has the current time as its creation date. As an example, the following command copies the file **MyFile.mesa** and **MyOtherFile.mesa** into the file **Temp.mesa**:

```
Copy Temp.mesa ← MyFile.mesa
MyOtherFile.mesa
```

CreateDir

creates a directory with the name you type.

CreateDir <CoPilot>NewDir

CWD

replaces the current working directory with the one you type. The facility for changing the current working directory also exists in the **SearchPathTool**:

CWD <CoPilot>Temp

Delete

takes a list of file names or directories and deletes each one. If the specified directory is not empty, or if it is on the current search path, the Executive will abort the deletion and print an error message. As an example, the following

command deletes the files **MyFile.mesa** and **MyOtherFile.mesa**:

Delete MyFile.mesa MyOtherFile.mesa

Filestat

takes a list of file names or directories and prints out the file ID, number of bytes in each file, the file type, the times at which the file was created, last read, and last written, and whether the file is delete-protected, read-protected or write-protected. As an example, the following command requests file information on file **MyFile.mesa**. Typical output is listed below.

```
Filestat MyFile.mesa
MyFile.mesa FileID: 0,
125000B,601B,64150B,15144B
11520 bytes
type: text
create: 5-Jan-82 15:30:25 write: 11-Jan-82
17:42:06
read: 14-Jan-82 19:41:41
```

If you have the file ID of a file rather than the file name, **Filestat** can still be used to obtain file information. Instead of the file name, use the file ID, preceded by the **s** switch. Numbers must be separated by spaces.

Filestat /s 0, 125000B, 601B, 64150B, 15144B

Floppy

recognizes commands that allow you to store and retrieve files on floppy disks using the floppy disk drive in your workstation. (For a detailed discussion of the commands, arguments and switches recognized by **Floppy**, see the chapter on floppy commands.)

Floppy command arguments.

Load

interprets each token on the command line as a file name and loads that program. Prints the load handle of each program loaded. You can specify the following switch, either locally or globally:

l: use codelinks when loading

As an example, the following command will load the programs **MyProgram.bcd** and **MyOtherProgram.bcd**

Load MyProgram.bcd MyOtherProgram.bcd

LogIn

prompts you for your name and password. An example of the use of **LogIn** is:

LogIn**User: YourName Password: YourPassword****OpenVolume**

takes a list of volume names and opens each volume.

You can specify the /w switch (open the volume for read-write instead of readOnly) either locally or globally.

OpenVolume Tajo User/wopens the logical volume **Tajo** for reading and the logical volume **User** for read-write.

A volume being opened should not be on the current search path. For example, if you wanted to open your Library volume for read/write, you would type **OpenVolume Library/w**. If <Library> were on the current search path, the feedback message would say **Unexpected MFile error**. However, if you take all <Library> references out of your current search path, things work as advertised.

PopWD

pops the working directory, eliminating it from the current search path, and leaving the next directory in the search path as the working directory.

PushWD

pushes the directory named to the front of the current search path, making it the current working directory.

ProcessInBackgroundcauses the compiler and binder to run at background priority when run from **CommandCentral**. This command does not take parameters. The default priority is **normal**.

ProcessInNormalPriority causes the compiler and binder to run at normal priority when run from **CommandCentral**. This command does not take parameters. The default priority is **normal**.

Rename

expects a command line in one of two forms:

TargetFile ← SourceFile

or

SourceFile TargetFile

If the target file already exists, the command will fail. Otherwise, the source file will be renamed to the target file. As an example, either of the following commands will rename the file **MyFile.mesa** to be called **NewFile.mesa**:

Rename NewFile.mesa ← MyFile.mesa
Rename MyFile.mesa NewFile.mesa

Run

interprets each token on the command line as a file name and runs that program. You can specify the following switches, either locally or globally:

- l use codelinks when loading
- d call the debugger after loading but before starting the program
- a start any tools created by the program in the active tool state
- i start any tools created by the program in the inactive tool state
- t start any tools created by the program in the tiny tool state

As an example, the following command will run the programs **MyProgram.bcd** and **MyOtherProgram.bcd**. After **MyProgram.bcd** has been loaded, but before it has been started, the system will break to the debugger.

Run MyProgram.bcd/d MyOtherProgram.bcd

SetClientVolume

sets the client volume that will be used by the **Run!** command in **CommandCentral** (and by **ClientRun**). As an example, the following command sets the client volume to the logical volume named **Tajo**:

SetClientVolume Tajo

SetErrorLevel

Outcome/switch <outcome/switch>

This command allows you to indicate whether processing should proceed, wait or abort following an error or warning. The outcome can be either warning or error. Switches can be either p for proceed, w for wait or a for abort. The default is to abort whenever a warning or error occurs. If you decide to wait following a particular outcome, processing will continue only after you type any character, except "q," which will halt rather than continue processing. The switches can be ordered according to their severity as follows: p < w < a. The switch chosen for errors must be greater than or equal to that for warnings; that is, **warning/a error/p** is not a legal combination since it violates the ordering constraint.

SetErrorLevel warning/p error/a

SetPriority

level (1, 2 or 3)

sets the priority at which the Executive will run. The priority must be specified in terms of a number: 1 is the lowest priority and stands for background; 2 is for normal priority; and 3 is the highest, meaning foreground priority. Default is 2, normal priority. The priority may be initialized by adding the appropriate a **User.cm** entry (see below).

SetPriority 2

SetSearchPath

sets the search path to the list of directories in the command line. The user can specify the following local switch:

- r readOnly search path entry.**

As an example, the following command sets the search path so it contains the directories <Tajo>Temp, <Tajo>Defs, and <Tajo>.

SetSearchPath <Tajo>Temp <Tajo>Defs <Tajo>

ShowSearchPath

displays the current search path in the Executive window.

Snarf

expects a list consisting of volume and file name. It copies a file from the source volume onto the current volume. The default source volume is CoPilot. The user can specify the following local switches:

- c** interpret the next argument as a command. The permissible commands are **SourceDir** and **DestDir**. These commands have been added so that the user can specify the source or destination directory of a snarf. The name of the directory is the next name on the line.
- s** rename this file when copying it; the target name is the next name on the line.
- u** copy the file only if the source file is newer than the target file, or if the target file does not exist.

As an example, the following command copies the files **MyFile.mesa** and **MyOtherFile.mesa** from the logical volume **Tajo**, renaming **MyOtherFile.mesa** to **Temp.mesa**. **MyFile.mesa** will be copied only if the source files is newer than the target file or the target file does not exist.

**Snarf SourceDir/c <Tajo> MyFile.mesa/u
MyOtherFile.mesa/s Temp.mesa**

Start

interprets each token on the command line as the load handle of a loaded program and starts that program. You can specify the following switches, either locally or globally:

- a start any tools created by the program in the active tool state
- i start any tools created by the program in the inactive tool state
- t start any tools created by the program in the tiny tool state

As an example, the following command starts the program with load handle **4063700B** in the tiny state:

Start 4063700B/t

Type

takes a list of file names and displays the contents of each in the Executive window. As an example, the following command types the files **MyFile.mesa** and **MyOtherFile.mesa**:

Type MyFile.mesa MyOtherFile.mesa

Unload

Command₁ ... <Command_n>

unloads the specified command and the module or configuration implementing it, provided it has been previously registered with the Executive. **Unload** will also unload commands that have been aliased using **AliasCommand**, or renamed using **ChangeCommandName**. Since the Executive keeps track of all original command names as well as those that have been renamed, both the original and alias or rename may be supplied to **Unload**.

Zap

takes a list of file names and causes them to be deleted, or, if they are currently in use, to be deleted when they are no longer in use. It is usually used to permit the retrieval of copies of programs that are already loaded, or to delete files that have accidentally been left locked by another program. As an example, the following command zaps the files **MyProgram.bcd** and **MyOtherProgram.bcd**.

Zap MyProgram.bcd MyOtherProgram.bcd

The file name always disappears immediately from the file system, so a new file of that name may be created right away.

4.2.5 Exec Ops menu

The Exec Ops menu is available outside all windows and contains the following commands:

FileWindow	creates a new Source window.
Run	runs the file that is the current selection.
Load	loads the file that is the current selection.
Start	starts the load handle that is the current selection.
New Exec	creates a new Executive window.
Quit	does a physical volume boot.
Power Off	shuts off the power.
CoPilot	boots your CoPilot volume.

4.3 User.cm processing

The Executive section of a **User.cm** file can contain the following entries:

CompilerSwitches:	the default switches to be used by the compiler.
BinderSwitches:	the default switches to be used by the binder.
ClientSwitches:	the default boot switches to be used by the Executive's built-in Run command as well as the Run! command in CommandCentral.
ClientVolume:	the volume to be used by the Executive's built in Run command as well as the Run! command in CommandCentral.
Priority:	the priority that the Executive should run in. Choices are 1 for background priority, 2 for normal priority, or 3 for foreground priority. The default is 2, normal priority.
UseBackground:	if TRUE , then the compiler and binder will be run at background priority from CommandCentral.
CodeLinks:	if TRUE , codelinks will be used by default when loading programs.
WindowBox:	location of the Executive's window box.
TinyPlace:	location of the Executive's tiny box.
InitialState:	initial state of the Executive (Active, Tiny, or Inactive).



HeraldWindow

CoPilot and Tajo have a banner called the HeraldWindow appearing at the top of the screen. It displays the name and version of the boot file, the date on which it was built, the current user, the current time and date, a logical volume name, and the number of free pages on that volume. It allows other tools to display messages in its window and has a menu that allows you to boot any of the bootable volumes.

5.1 Files

The HeraldWindow is built into CoPilot and Tajo.

5.2 User interface

A **Boot from:** menu is available through the HeraldWindow. It is invoked by positioning the cursor in the window and pressing **MENU**.

5.2.1 Boot from: menu

Besides containing the names of the volumes on your workstation, the **Boot from:** menu lists the following options:

File Name: uses the current selection as the name of a boot file on the current logical volume to be booted.

Set Switches: uses the current selection as a string of Pilot booting switches for a subsequent booting command. The scanner recognizes the following syntax: The characters ~ and – change the sense of the immediately following switch. Each character of the selection is the character representation of a switch. \ is an escape character. If it is followed by a three-digit octal number, the switch is the character with that octal representation. If \ is followed by the characters N, n, or R, or r, the switch is the Ascii CR character. If \ is followed by B or b, the switch is the Ascii BS character. If \ is followed by F or f, the switch is the Ascii FF character. If \ is

followed by L or l, the switch is the Ascii LF character. If \ is followed by ', ", ~, or -, the switch is that character.

Reset Switches uses default switches for a subsequent booting command.

Boot Button automatically pushes the boot button.

Set Priority Up sets the priority of the clock process to foreground, making it a good stopwatch.

Reset Priority resets the priority of the clock process to normal.

There may be other volume names in the menu. Invoking any of these causes the volume to be booted after confirming with a mouse click.

When the HeraldWindow is made tiny, it can display the current date and time, the Pilot logical volumes, and their free page counts. Move the cursor into the tiny HeraldWindow and it will display the date and time. Each successive click with **POINT** will display the name and free page count of a Pilot logical volume, starting with the system volume. If the information about all the volumes has been displayed, the HeraldWindow will redisplay the date and time. The HeraldWindow will stop displaying this information when you move the cursor out of its window. If you wish to have the HeraldWindow continue to display after the cursor is moved out of the window, click **ADJUST**. To cause the HeraldWindow to revert to its normal state, click the right button in the window again.

The name and free page counts of volumes other than the system volume may also be obtained when the HeraldWindow is active, by clicking the mouse over the volume name in the right side of the window. Each successive click with **POINT** will display the name and free page count of a Pilot logical volume, starting with the system volume. If the volume is not the system volume, it will have an asterisk appended to its name. Clicking **ADJUST** over the volume name will cause the HeraldWindow to continue displaying information for that volume after the cursor has moved out of that region of the window.

5.3 User.cm processing

The HeraldWindow initializes its window box, tiny position, and its initial state from entries in the [HeraldWindow] section of the **User.cm**:

```
WindowBox: [x: 362, y: 628, w: 662, h: 150] -- location of tool's
                                                window box

TinyPlace: [x: 720, y: 778]                      -- location of tool's
                                                tiny box

InitialState: Active                            -- initial state of tool
```

Profile Tool

The Profile Tool, which is built in, allows you to edit information used by other tools running in the development environment.

6.1 User interface

The Profile Tool interacts with you through a form subwindow, which contains the following fields:

Profile Tool as of 24 Sep 94 14:39:45			
Apply!	User:	Password:	Registry:
Abort!	Domain:	Organization:	Debugging
	Librarian:	Prefix:	Suffix:

- User** is a text form item for your login name. This field is normally initialized by a value specified in the **User.cm**.
- Password** is your password.
- Registry** contains the mail registry to which you belong. This field is normally initialized by a value specified in the **User.cm**.
- Domain** contains the clearinghouse domain you wish to use. It is needed when communicating with NS servers, such as printers and file servers. This field is normally initialized by a value specified in the **User.cm**.
- Organization** contains the clearinghouse organization you wish to use. It is needed when communicating with NS servers, such as printers and file servers. This field is normally initialized by a value specified in the **User.cm**.
- Debugging** is a Boolean form item that some tools read. When a tool detects an error situation, it may go to the debugger if **Debugging** is **TRUE** and print out a message to the user if **FALSE**. If you are not prepared to go to the

debugger, you should set the Boolean to **FALSE**. This field is normally initialized by a value specified in the **User.cm**.

Librarian contains the network address or name of the default Librarian service. This field is normally initialized by a value specified in the **User.cm**.

Prefix: is used to expand libject names into full libject names. **Prefix:** is a string of one or more tokens, each of which represents a project identity (e.g., **Tools** > **<Pilot>**, etc.) This field is normally initialized by a value specified in the **User.cm**.

Suffix: is used to expand the libject name you supply into a full libject name (e.g., **mesa**, **config**, etc.). This field is normally initialized by a value specified in the **User.cm**.

The Profile Tool displays the following commands only when the values of one or more of the data items have been edited so that the values displayed in the window are (potentially) different from the values of the underlying system variables. When the values are the same, these commands will not be displayed:

Apply! is a command form item that enters the information in the Profile Tool's subwindow into the system, making the information available to other tools. Note that no changes take effect until you invoke **Apply!**

Abort! is a command form item that resets the information in the Profile Tool's subwindow from the system variables.



Tool Driver

The Tool Driver extends the facilities of the Xerox Development Environment by providing a mechanism for automatically performing repetitive, routine batch tasks. It does this by acting as a simulated user that interprets simple command sequences. The Tool Driver uses only the functions available through the XDE's user interface, rather than accessing special hooks in various low levels of the Development Environment and the attendant common collection of tools.

The power of the Tool Driver is constrained only by the power of the set of tools that are loaded and accessible to it. However, the flexibility and sophistication of the commands understood by the Tool Driver is low. It is not intended to meet all your non-interactive needs, but instead tries to provide simple catalogued procedures.

The Tool Driver has the potential to completely destroy large, permanent user data structures such as Action Request databases. For this reason, certain tools may place extra restrictions on the operations that they will allow while under the control of the Tool Driver. Any such restrictions will be discussed in the documentation for the individual tools.

7.1 Files

Three files are required to use the Tool Driver. The first is the Tool Driver's code, **Tools>ToolDrivers.bcd**; the second is a list of the tools that you might want the Tool Driver to manipulate, **Tool.sws**; and the last is a set of instructions for the Tool Driver (a script for the simulated user).

If you wish to make tools available for use through the Tool Driver or are interested in extending the Tool Driver, retrieve <**Mesa>Doc>ToolDriverClient.memo**.

7.2 User interface

The Tool Driver communicates via the Tool Driver Executive window. This tool allows you to specify the name of the script files and the options to be used by the Tool Driver during execution of the scripts.

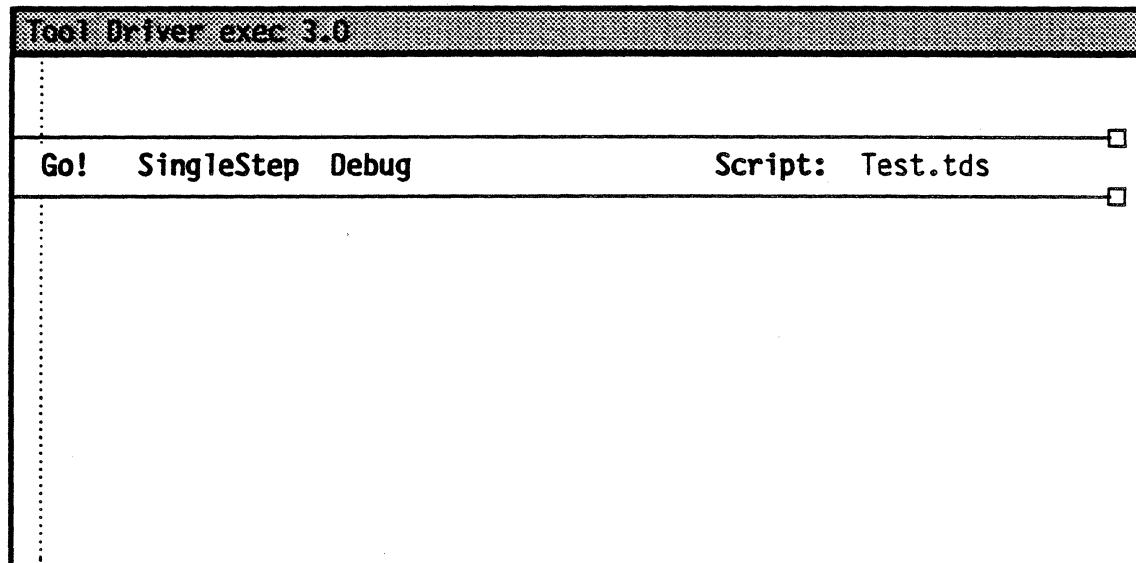


Figure 7.1: Tool Driver executive window

The Tool Driver executes scripts until it either runs out of input, is aborted, or encounters an error. A script can cause the Tool Driver to temporarily interrupt its execution and return to the user; except for these breaks, the Development Environment's Notifier is completely tied up by the execution of the Tool Driver.

7.2.1 Message subwindow

Messages that are a result of calls on the function pause are displayed in the message subwindow.

7.2.2 Form subwindow

The form subwindow contains the following items:

Go! causes the Tool Driver to execute using the specified file as the input script. Use **ABORT** to abort the execution.

SingleStep is a Boolean which, if **TRUE**, causes the Tool Driver to pause after it executes each statement in the script. Otherwise, execution does not halt unless either the script is finished, the user or a tool aborts, or an error occurs.

Debug is used for debugging the Tool Driver itself. Its value should normally be **FALSE**.

Script: is a string item that lists names of the input script files. It is defaulted to **Test.tds** (the extension **.tds** for script files is an acronym for tool Driver script). If a script is aborted, either by the user or by one of the tools being driven, the rest of the scripts will not be executed (see the Script files section).

7.2.3 File subwindow

The file subwindow is used to log messages of more than transient interest, such as the name of the script file currently being executed, **Done** or **Abort**, or other status messages indicating how or why the script file finished. The root log name for this tool is **TDE.log**.

7.3 Script files

A script file is a text file containing a series of statements. A statement is either an assignment to a variable, a command, a loop or exit loop, a simple conditional, or a function call.

7.3.1 Script file format

There is no inter-statement separator, optional or otherwise. White space is not significant, except that it delimits atoms in the script. The commenting conventions are those used in Mesa. Occasionally it may be necessary to quote an arbitrary character in the script by preceding the character by a ' character. The \ is treated as an end-of- file signal, and should not appear unquoted in a script unless you want the Tool Driver to ignore the following part of the script.

7.3.1.1 Constants and variables

Delimited strings (must be preceded and followed with double quotes), unsigned numbers, or one of the set of reserved words **NIL**, **TRUE**, and **FALSE**, are constants. Whether a constant is semantically valid depends on the context in which it is used.

Variables reference items in form subwindows. The format of a variable reference is **ToolName.SubwindowName.Tag**; e.g., **AREditTool.CommandSW.UseQL**. If **ToolName** is omitted, then the value of the reserved variable **TOOL** is used. If **SubwindowName** is also omitted, then the value of the reserved variable **SUBWINDOW** is used. The tag trailer provided by the **FormSW** package must not be present in **Tag**.

All other available facilities are invoked by function calls.

7.3.1.2 Assignment to variables

A variable is assigned to by

Form item \leftarrow *Expression*

where *Expression* is either a constant, a variable, or a function call.

7.3.1.3 Function calls

Function calls are positional and do not allow defaulting. Provision has been made for the Tool Driver's set of functions to be dynamically increased. A function call must always have the form:

Function[*ExpressionList*]

where an *ExpressionList* is one or more *Expressions*, separated by commas.

These are the function calls currently allowed:

ActivateTool[*Expression*].

The *Expression* must specify the name of an entry in the Tajo Inactive Tools menu. This entry might not match the tool's herald, its tiny name, or its name as known to the Tool Driver for variable referencing purposes. If the name is found in the menu, then the Tool is activated; otherwise this call is a no-op.

AppendCommand[*ToolName.SubwindowName, Expression*].

This calls **UserInput.Stuff** string with the subwindow handle and string value.

AppendString[*ToolName.SubwindowName, Expression*].

This calls **Put.Text** with the subwindow handle and string value.

CallDebugger[*Expression*].

This calls the debugger with the *Expression* as the message to be printed by the debugger.

FileCreated[*Expression, Expression*].

The first *Expression* is the name of the file to check on. **TRUE** is returned if the file exists and was created within the number of seconds specified by the second *Expression*.

InvokeMCR[*ToolName.SubwindowName, Constant, Constant*].

The *ToolName* may be omitted, in which case the default will be used. The first constant is the name of the menu; the second is the keyword in that menu.

IsVisible[*Form item*].

TRUE is returned if the specified form subwindow item's invisible flag is **FALSE**.

LastMessage[*ToolName.SubwindowName*].

This returns the last message posted in the message subwindow specified. The *ToolName* may be omitted, in which case the default will be used.

ModifyItem[*Form item, Expression, Expression, Expression*].

This allows you to insert, delete, or replace characters in the specified form subwindow item. The first *Expression* specifies the position at which to start the modification, beginning with 0 at the left edge of the body of the item (i.e., the item's tag and tag trailer are not accessible). The second *Expression* specifies the number of characters to be affected, and the last *Expression* is the new characters (if any). Thus *pos, length, NIL* for the three *Expressions* specifies a deletion beginning at *pos* of *length* characters. *pos, 0, "new string"* specifies the insertion of the nine characters "new string" at *pos*. *pos, length, exp* specifies a replacement. For convenience, all starting positions off the right edge of the item are trimmed back to the right edge, so appending new text to the item can be achieved by

using the expression (100000B, 0, *newText*). For further details, see the description of the Tajo procedure **FormSW.ModifyEditable** in the *Mesa Programmer's Manual*.

Pause[*Expression*, *Expression*].

This allows you to intervene and interrogate while a script is being executed. It prints the first argument in the Tool Driver exec's message subwindow and then enables the Notifier, allowing you to interact with the development environment again. The second argument indicates whether the **Pause** is simply trying to ask a question. It must be either **TRUE** or **FALSE**. If **TRUE**, the Tool Driver Exec adds two new items to its command subwindow, named **Yes** and **No**. If you invoke **Yes**, **Pause** returns **TRUE**; if you invoke **No**, **Pause** returns **FALSE**. If the second argument is **FALSE**, the Tool Driver exec adds a new item to its command subwindow named **Proceed**, and **Pause** returns an undefined value when you invoke **Proceed**.

SetSelection[*Expression*].

This sets the current selection. There is no feedback to show what the selection has been set to.

SetWindowBox[*Tool/Name*, *Expression*, *Expression*, *Expression*, *Expression*].

This sets the tool's window to the size specified. The order of the arguments (from the left) is **x**, **y**, **w**, and **h**.

SubString[*Expression*, *Expression*, *Expression*].

This returns the value of the the subportion of the first expression that begins at the second expression and has a length specified by the third expression.

Wait[*Expression*].

This causes the Tool Driver to relinquish the processor for the specified number of seconds. During the wait, the Notifier is still disabled, but periodic notifications occur (although perhaps not as quickly as they normally would).

WindowOnTop[*Tool/Name*].

This brings the specified tool window to the top of the window stack.

7.3.1.4 Control structure

The Tool Driver allows for some forms of control structure. They are:

- 1) **DO**
...
IF BooleanExpression THEN EXITLOOP Label;
...
EXITLOOP Label;
ENDLOOP Label;

The *Label* after the **EXITLOOP** specifies the label on the **ENDLOOP** to which you are exiting and is optional. However, the semicolon after the *Label* is mandatory in both places. These are the only places in a script file where a semicolon appears.

- 2) **IF BooleanExpression THEN Statement**
- 3) **IF BooleanExpression THEN**
BEGIN
...
END
- 4) **IF BooleanExpression THEN**
BEGIN
...
END
ELSE Statement
- 5) **IF BooleanExpression THEN**
BEGIN
...
END
ELSE
BEGIN
...
END

The *BooleanExpression* has one of two forms:

Expression
or *Expression Relational Expression*

The *Relational* is one of the set $\{=, \#\}$.

7.3.2 Sample script

The following sample script would produce a query list of all the AR's submitted against the Ether subsystem of Mesa that has been marked **Fixed** in 6.0z. Then, by using this query list, it would edit each of the AR's so that their **In/By** field now reads 6.0m.

```

TOOL ← "AdobeQuery"
SUBWINDOW ← "formSW"
Number ← ""
System ← "Mesa"
Subsystem ← "Ether"
Status ← "Fixed"
In'/By ← "HAS 6.0z"
cmdsw.Query

TOOL ← "AdobeEdit"
SUBWINDOW ← "cmdSW"
UseQL ← TRUE
Next
Checkout
DO

    formSW.In'/By ← "6.0m"
    Next
    IF LastMessage[msgSW] = "Query List exhausted!" THEN EXITLOOP;
    Checkin'&out
    IF LastMessage[msgSW] = "Can't check out AR: must do update before
        further editing!" THEN
        BEGIN
        ARUpdateTool.CommandSW.Update
        Checkout -- Remember we are here because "out" part of "in&out" failed
        END
    ENDLOOP;
    Checkin -- don't forget to put the last guy back

```

7.4 BNF for script files

goal	::= statements \
statements	::= statements statement statement
statement	::= assignment formCmd loop semiSuffix ifStatement EXITLOOP loopLabel ; semiSuffix functionCall
assignment	::= formSWItem ← expression
formCmd	::= formSWItem
formSWItem	::= idList
idList	::= idList . id id

expressionList	:: = expressionList , expression expression
expression	:: = variable constant
expressionTail	:: = variable constant
variable	:: = formSWItem functionCall
constant	:: = delimStr num NIL TRUE FALSE
functionCall	:: = id [expressionList] functionName [expressionList]
functionName	:: = ActivateTool AppendCommand AppendString CallDebugger FileCreated InvokeMCR IsVisible LastMessage ModifyItem Pause SetDispState SetSelection SetWindowBox SubString Wait WindowOnTop
loop	:: = do statements ENDLOOP loopLabel ;
do	:: = DO
ifStatement	:: = ifExp block ifExp blockElse block
ifExp	:: = IF boolExp THEN
block	:: = statement BEGIN statements END
blockElse	:: = BEGIN statements END ELSE
boolExp	:: = expression relational expression expression

```

loopLabel      ::= id
|              

semiSuffix    ::= 

relational     ::= =
| #

```

Note: The *FormItem* must be a **command** item in the Form subwindow.

Note: The semantic restrictions on the *ExpressionList* depend on the *Id*.

7.5 The subwindows file

The Tool Driver will not function unless the subwindows file, **Tool.sws**, is present on the local disk. The format of this file is:

```

...
[ToolName1]
SubwindowName1, ..., SubwindowNamen

[ToolName2]
SubwindowName1, ..., SubwindowNamen

...

```

The opening [must be the first character on the line. Everything after the closing] on that line is simply ignored. If a tool that is not in the subwindows file attempts to publicize subwindows (i.e., calls **ToolDriver.NoteSWs**), it is ignored, as are all subwindows not present in the list of subwindows for that tool. The individual documentation for each tool should list the tool and subwindow names that the tool publicizes. There must be no extra subwindows declared by the user. If there are, the Tool Driver will halt with an error.

7.6 Running the Tool Driver

The procedure for running the Tool Driver is as follows:

- Start the Tool Driver.
- Start other tools.
- Run the script.

Note: Tools started before starting the Tool Driver are not accessible to the Tool Driver. Tools that are inactive are also inaccessible to the Tool Driver. However, inactive tools can be accessed indirectly via the **InvokeMCR** function applied to the Executive menu.

File-related tools

This chapter discusses the XDE tools for manipulating files. The first part explains file naming conventions, since file names are used by many of the tools as field values. The rest of the chapter briefly describes each tool's function.

II.1 File system conventions

Once you have written your text onto a file window or text subwindow, you will probably want to save it as a file. This section describes the XDE local file system's structure and naming conventions, which are used for searching for files as well as for creating new files.

Many of the tools in the development environment have parameters that are file names, such as the File Tool and the Executive. Some tools are prepared to deal with either local or remote file names. The syntax of remote file names is determined by the remote file system. Consult the documentation for your remote file system for the definition of legal remote file names.

II.2 File names

The local file system provides a tree-structured directory. The top-level directory, the root of the tree, has the same name as the logical volume. All directories can contain directories and non-directory files. A file has a simple name (that is, its name within a directory) and a fully qualified name (its name within the directory structure). The legal characters that can be used in the simple name of a file are the alphabetics (a - z, A - Z), digits (0 - 9), period (.), dollar sign (\$), plus (+), and minus (-).

The fully qualified name of a file, whether directory or non-directory, describes the path from the top-level directory of the volume containing that file to the file. The name starts with the character <, and all subdirectories on the path are separated by the character >. No file names end with the character > with the exception of the top-level directory, which always ends with >. Some examples of fully qualified file names are:

<CoPilot>

<CoPilot>MyFile.mesa

```
<CoPilot>SubDirectory>MyFile.mesa
```

```
<CoPilot>SubDirectory
```

Certain operations, such as the File Tool's and the Executive's list commands may print the names of directory files followed by a > to distinguish them from non-directory files. This is an output convention; don't confuse it with the name of the directory file.

The top-level directory of the current volume can also be specified by <>; that is, if the name of the top-level directory is omitted in a fully qualified name, the top-level directory of the current volume is used. Hence, the following names are equivalent to the above examples to a user on the volume CoPilot:

```
<>
```

```
<>MyFile.mesa
```

```
<>SubDirectory>MyFile.mesa
```

```
<>SubDirectory
```

A file name can also be specified relative to the current search path. If a file name does not start with the character <, it is a relative name. In this case, a fully qualified name is formed by appending the relative name to each entry of the search path until a match is found (refer to the chapter on the SearchPath Tool). If the search path contained the single entry <CoPilot>, the relative file name MyFile.mesa would be resolved to the fully qualified name <CoPilot>MyFile.mesa

Directories on the search path may be *write-protected*, in which case it is not possible to change any of the files in the directory or add or delete files from it. If a file name is relative to the search path and it is to be created or written into, two problems can occur: no match could be found on the search path, or the first match might occur in a directory that is write-protected. In either case, the file will be created in the first directory that is not write-protected in the search path. This directory acts somewhat like a working directory. If the first directory in the search path is write-protected, anomalies may result; for example, if you write into the file MyFile, and then subsequently try to read file MyFile, you may not read the information that you just wrote. This could happen if the first directory in the search path is write-protected but contains a file named MyFile. When you write into file MyFile, the system notices it is in a write-protected directory and creates a new file MyFile in the first writeable directory. When you later read the file MyFile, the system returns the first file named MyFile on the search path, which was the file MyFile in the write-protected directory.

II.3 File-related tools

Brownie helps distribute software and maintain consistent copies of archive directories on file servers.

Compare examines two pairs of source files and summarizes the differences between each. The files can be either local or remote.

A *File window* is used to view and edit a text file.

The *File Tool* provides a means for you to work with the files on your local disk as well as on remote file systems. It allows you to retrieve, delete, list, rename, and copy files. It is like FTP except that it has a window interface instead of an Executive command.

Find searches for a pattern in a list of files and displays the lines in which the pattern occurs.

Floppy commands allow you to store and retrieve files on floppy disks using the floppy disk drive in your workstation.

FTP is a file transfer program that runs in the Executive. It is used for moving files to and from a file system, which can be on a file server or on another workstation.

Print generates press format files and sends them to a printer on the network.

The *SearchPath Tool* is used to inspect and change the file system search path.



Brownie

Brownie aids in the problem of how to distribute software and maintain consistent copies of master or archive directories on several file servers. It may also be helpful in moving files among private directories during the software development process.

8.1 Files

Retrieve **Brownie.bcd** from the Release directory.

8.2 User interface

Brownie is invoked by typing a command of the following form to the Executive:

>Brownie file

where **file.brownie** is a Brownie script file with the format described below. Brownie will prompt for login and connect names and passwords for the hosts and directories involved in the transfer. It will also log messages to the Executive, informing the user of its progress.

8.3 Script file

The script file describes the operations Brownie is to perform. It consists of a parameter section and a command section separated by a comment line. The comment is ignored, but the // must appear. In the script below, the first **QualifiedFilename** is the target and the second **QualifiedFilename** is the source.

```
[level]
start: [time]
stop: [time]
// comment
copy/switches  QualifiedFilename/<-QualifiedFilename/
...
...
```

```

rename/switches QualifiedFilename ← QualifiedFilename
...
delete/switches QualifiedFilename

```

8.3.1 Parameters

All parameters are optional, and if present their order is not important.

The amount of information logged is controlled by the **level** parameter. The choices are **verbose** and **terse**. **verbose** mode will post the name of each source and destination file as it is being copied (or deleted), along with their creation dates. **terse** mode will post directory names only, and a dot for each file as it is copied. **terse** mode is normally recommended for large copies, to keep the **Executive.log** file from getting too large. **level** defaults to **terse**.

The **start** parameter allows you to specify a start-up time. This allows lengthy transfers that tie up a lot of network resources to be delayed until nighttime. Brownie processes the script file before doing any transfers so that any syntax errors may be discovered immediately. The **stop** parameter allows you to specify a stopping time. Brownie periodically glances at the stop time and aborts processing if the current time becomes larger than this value. **time** may be in any of the formats: HH:MM, HHMM, H:MM, or HMM. **time** defaults to **start** *immediately* for **start** and *when finished* for **stop**.

8.3.2 Commands

A **QualifiedFilename** (*QFN*) of a Brownie command has the general form:

[host]<directory>filename

Where **filename** is optional. The Profile domain and organization are appended to **host** if none are specified. If a **QualifiedFilename** contains spaces, it must be surrounded by double quotes.

8.3.2.1 Copy

The **copy** command transfers the files described by the source *QFN* to the target *QFN* according to the constraints of **switches**. If **filename** appears in both the source and the target, the single file is transferred. If **filename** is omitted from the source *QFN*, it must also be omitted from the target *QFN*, meaning copy all files from the source directory to the target directory. If **filename** is not omitted from the target in this case, all files from the source will be copied to the single target file.

"*" wildcards may appear within the source *QFN*. (See the FileTool section: Wildcard/expansion characters for an explanation of wildcards.) A "*" may also appear as the only character of the final subdirectory, instructing Brownie to recursively search through the specified directory. All files matching the *QFN* will be copied. If a "*" appears, the target *QFN* as in the previous case must be a directory. A "*" may *not* appear in the target *QFN*.

8.3.2.2 Copy switches

- /c Connect to target directory; prompt for credentials. Default is FALSE. (Not implemented)
- /s Connect to source directory; prompt for credentials. Default is FALSE. (Not implemented)

The Update (/u) and Always (/a) switches have identical meaning to those of FTP.

- /u Copy the files specified by the source QFN only when the creation date of the source file is greater than the creation date of the target file and the target file exists. Default is FALSE.
- /a Copy the files even if those files of the target QFN don't exist. Default is TRUE.

8.3.2.3 Rename (Unimplemented)

The **rename** command renames single files or complete directories on a single file server. Only the latest versions of files are renamed, unless the /a switch is specified. If **filename** is omitted from both QFNs, the entire source directory is renamed to the target directory; otherwise, the single file is renamed. A "*" may not appear in either QFN.

8.3.2.4 Rename switches

- /c Connect to (source) directory; prompt for credentials. Default is FALSE.
- /a Rename all versions of the source QFN. Default is FALSE.
- /u Update (Unimplemented).

8.3.2.5 Delete

The **delete** command deletes one or more files on a file server. Only the oldest versions of files are deleted, unless the /a switch is specified. A "*" may appear in a QFN. (See the FileTool section: Wildcard/expansion characters for an explanation of wildcards.)

8.3.2.6 Delete switches

- /c Connect to directory: prompt for credentials. Default is FALSE. (Not implemented)
- /a Delete all versions of the source QFN. Default is FALSE.

8.4 Example

This is an example of a script file:

```
[terse]
start: [20:30]
// Start at 8:30PM; commands follow
copy/ua "[RatTail:OSBU North]<emerson>doc"<
```

```
[Rasp]<emerson>doc>*>*!*
copy/u [Igor]<emerson>defs> ← [Idun]<int>tajo>public>*.mesa
copy [Sun]<newInt>brownie>Brownie.bcd ←
[Igor]<emerson>brownie>Brownie.bcd
copy [Sun]<newInt>brownie>Brownie.doc ←
[Igor]<emerson>brownie>Brownie.doc
delete/ca [Bad]<Movies>*
delete [Mediocre]<Movies>*
```

To execute Brownie with the above example script, **Example.brownie**, type the following command to the executive:

>Brownie Example

and log in according to the prompts for each host and directory.



FTP

FTP is a file transfer program used for moving files to and from a file server.

The File Tool serves the same purpose as FTP. (For more information, see the File Tool chapter.)

Transferring a file from one host to another over a network requires the active cooperation of programs on both machines. In a typical scenario, a human user (or program acting on the human's behalf) directs FTP (or the File Tool) to establish contact with a file server.

9.1 Files

Retrieve **FTP.bcd** from the Release directory.

9.2 User interface

FTP runs in the Executive.

9.2.1 Command line syntax

The two basic file transfer operations are **Retrieve** and **Store**. The **Retrieve** command causes a file to move from server to user, whereas **Store** causes a file to move from user to server.

Other commands are often used in conjunction with the basic **Retrieve** and **Store** commands. Commands are of the form:

<Keyword>/<SwitchList> <arg> ... <arg>

Unambiguous abbreviations of command keywords (which in most cases amount to the first letter) are legal. A command is distinguished from arguments to the previous command by having a switch on it, so every command must have at least one switch.

9.2.2 Command line switches

In the descriptions that follow, the terms *local* and *remote* are relative to the machine on which the FTP user program is active (that is, you type commands to your local user

program and direct it to establish contact with a file server.) A **Retrieve** command copies a file from the remote file system to the local file system, whereas a **Store** command copies a file from the local file system to the remote file system.

Local and *remote* also refer to file names. Files on your workstation are local, and files on a server are remote.

Most commands take local switches. These switches have default values used if the switch is not mentioned. The switches are listed below with their defaults and functions:

- /C [Command]** a null switch that tells the command line parser that this token is a command (no default).
- /S [Selective]** used if the remote and local file names differ; for example, if you retrieve a file listed under one name but want to bring it to your workstation under a different name (**FALSE**).
- /V [Verify]** requests confirmation from the keyboard before the file transfer takes place. Confirm with **V** (not **CR**); deny with **N**. **S** (for **STOP**), **DELETE**, or **CONTROL-C** will terminate all further commands (**FALSE**).
- /Q [Query]** specifies that a password be requested interactively from the user instead of being read from the command line (**FALSE**).

If FTP can unambiguously decide that a token is a command, you do not need to append any switches to the command word. Otherwise, you must append some switch; use the **/C** switch if there are no other switches desired. This means that if a command (such as **Retrieve**) takes a list of files and the list is followed by another command, that command must have some switch appended.

Some switches affect transfers conditioned upon comparison of the creation dates of corresponding local and remote files. The comparison is **<source file> <operator> <destination file>**. For **Store**, the source file is the local file; for **Retrieve**, the source file is the remote file:

- /# [NotEqual]** transfers the file if the destination file exists and the creation dates are not equal. This must be quoted (**'#'**) to keep it out of the clutches of the Executive.
- /= [Equal]** transfers the file if the destination file exists and the creation dates are equal.
- [Greater] transfer the file if the destination file exists and the source's creation date is greater than the destination's.
- [Less] transfers the file if the destination file exists and the source's creation date is less than the destination's.
- /U [Update]** same as (for backward compatibility).
- [All] modifies the action of **#**, **=**, **>**, **<**, **/U** to transfer the file even if no corresponding file exists in the destination file system.

If more than one switch is present, they are ORed together, so, for example, "/>=" means "transfer the file if the source's creation date is greater than or equal to the destination's."

The sense of a switch is inverted if it is preceded by a minus sign; the minus sign inverts the sense of the immediately following character, not the entire operator expression.

9.2.3 Commands and examples

In the examples below, the /C switch has been included, even though it may not be necessary.

Open/C <HostName>

opens a connection with the host. The first token after FTP in the command line is assumed to be a host name, so no subsequent Open command is required. The Profile domain and organization are appended to <HostName> if none are specified.

Close/C

closes the currently open FTP connection.

Login/C <UserName> <password>

supplies any login parameters required by the remote server before it permits file transfers. FTP will use the user name and password in your Profile (see the Profile Tool chapter), if they are there. Logging into FTP will set the user name and password in your Profile, if they have not already been set.

When you issue the Login command, FTP will first display the existing user name in your Profile. If you now type a space, FTP will prompt you for a password. If you want to provide a different user name, you should first type that name (which will replace the previous one) followed by a space. The command may be terminated by a carriage return after entering the user name, to avoid entering the password. The parameters are not immediately checked for legality, but rather are sent to the server for checking when the next file transfer command is issued. If a command is refused by the server because the name or password is incorrect, FTP will prompt you as if you had issued the Login command and then retry the transfer request. Typing CONTROL-C aborts both the request for login information and the rest of the FTP command line.

Login/Q <UserName>

causes FTP to prompt you for the password. This form of Login should be used in command files, because including passwords in command files is bad practice.

Directory/C <DefaultDirectory>

causes <DefaultDirectory> to be used as the default remote directory in data transfer commands (essentially it prefixes the directory name to remote file names that do not explicitly mention a directory). The default directory can be overridden at any time by fully specifying a file name within a particular command ([Host]<Dir>filename). Do

not include punctuation that separates the directory name from other parts of the remote file name; thus, type *Directory Mesa*, not *Directory <Mesa>*.

LocalDirectory/C <DefaultDirectory>

causes the default directory to be used as the default local directory in the transfer. For example, if you want to retrieve files onto a local directory in your Tajo volume without having to specify the destination name each time, you can specify a default local directory and it will be prepended to all file names.

Retrieve/C <RemoteFilename> ... <RemoteFilename>

retrieves each *<RemoteFilename>*, constructing a local file name from the actual remote file name as received from the server. FTP will overwrite an existing file. If the remote host allows "*" (or some equivalent) in a file name, a single remote file name may result in the retrieval of several files. You must quote the "*" to get it past the Executive's command scanner.

Retrieve/S <RemoteFilename> <LocalFilename>

retrieves *<RemoteFilename>* and names it *<LocalFilename>* in the local file system. This version of **Retrieve** must have exactly two arguments. The remote file name should not cause the server to send multiple files.

Retrieve/> <RemoteFilename> ... <RemoteFilename>

retrieves *<RemoteFilename>* if its creation date is greater than that of the local file. If the corresponding local file doesn't exist, the remote file is not retrieved. This option can be combined with **Retrieve/S** to rename the file as it is transferred.

Retrieve/>A <RemoteFilename> ... <RemoteFilename>

is the same as **Retrieve/>** except that if the corresponding local file does not exist, the remote file is retrieved anyway.

Retrieve/V

requests confirmation from the keyboard before retrieving a file. This option is useful in combination with the Update option (/U), because the creation date is not a foolproof criterion for updating a file.

Store/C <LocalFilename> ... <LocalFilename>

stores each *<LocalFilename>* on the remote host, constructing a remote file name from the name body of the local file name. A local file name may contain "*", because it will be expanded by the Executive into the actual list of file names before the FTP subsystem is invoked.

Store/S <LocalFilename> <RemoteFilename>

stores <LocalFilename> on the remote host as <RemoteFilename>. The remote file name must conform to the file name conventions of the remote host. This version of **Store** must have exactly two arguments.

Store/> <LocalFilename> ... <LocalFilename>

stores each <LocalFilename> on the remote host if the local file's creation date is later than the remote file's. If the corresponding remote file does not exist, the local file is not stored. This option can be combined with **Store/S** to rename the file as it is transferred.

Store/>A <LocalFilename> ... <LocalFilename>

is the same as **Store/>** except that if the corresponding remote file does not exist, the local file is stored anyway.

Store/V

requests confirmation from the keyboard before storing a file. This option is useful in combination with the **Update** option when creation date is not a foolproof criterion for updating a file.

List/C <RemoteFileDesignator> ... <RemoteFilename>

lists all files in the remote file system that correspond to <RemoteFileDesignator>. The remote file designator must conform to file-naming conventions on the remote host. The following subcommands request printout of additional information about each file. They are specified by local switches:

- /t type,
- /l length in bytes,
- /d creation date
- /w write date,
- /r read date,
- /a author (creator),

f<date> - from<date>. Lists only files with write date greater than <date>. This must be the last entry on the command line before the file name. Example:
list/f10-Dec-79-11:00:04 *.mesa.

b<date> - before<date>. Lists only files with read or write date less than <date>. This must be the last entry on the command line before the file name.

Note: The file system keeps creation, read, and write dates with each file. FTP treats the read and write dates as properties describing the local copy of a file; i.e., when the file was last read and written in the local file system. FTP treats the creation date as a property of the file contents; i.e., when the file contents were originally created, not when the local

copy was created. Thus, when FTP makes a file on the local disk, the creation date is set to the Creation date supplied by the remote FTP, the Write date is set to 'now' and the Read date is set to 'never read.'

Delete/C <RemoteFilename>

deletes *<RemoteFilename>* from the remote file system. The syntax of the remote file name must conform to the remote host's file system name conventions. This **Delete** is an irreversible act. It is therefore unwise to use the "*" in the *RemoteFilename* to specify deletion of multiple files.

Delete/V <RemoteFilename>

asks you to verify that you want to delete *<RemoteFilename>* from the remote file system. If the remote file name designates multiple files (the remote host permits "*" or some equivalent in file names), FTP asks you to confirm the deletion of each file. Type **V** to delete the file; **N** if you don't want to delete it.

Compare/C <RemoteFilename>...<RemoteFilename>

compares the contents of *<remote filename>* with the file by the same name in the local file system. It tells you how long the files are if they are identical, or the byte position of the first mismatch if they are not.

Compare/S <RemoteFilename> <LocalFilename>

compares *<RemoteFilename>* with *<LocalFilename>*. The remote file name must conform to the file name conventions of the remote host. This version of **Compare** must have exactly two arguments.

Rename/C <OldFilename> <NewFilename>

renames *<OldFilename>* in the remote file system to be *<NewFilename>* in the new file system. The syntax of the two file names must conform to the remote host's file system name conventions, and each file name must specify exactly one file.

9.2.4 Command line errors

Command line errors fall into three groups: syntax errors, file errors, and connection errors. FTP can recover from some of these.

Syntax errors, such as unrecognized commands or the wrong number of arguments to a command, cause FTP's command interpreter to lose its place in the command file. FTP recovers from syntax errors by ignoring text until it encounters another command (i.e., another token with a switch).

File errors, such as trying to retrieve a file that does not exist, are relatively harmless. FTP recovers from file errors by skipping the offending file.

Connection errors, such as executing a **Store** command when there is no open connection, could terminate the command.

When FTP detects an error, it displays an error message and aborts the rest of the command.

9.3 Tutorial

The following are examples of how to use FTP:

- To transfer files **FTP.bcd** and **FTP.symbols** from the Dandelion called Chocolate to the Dandelion called Vanilla, you might start up the **STP** server on Chocolate, then walk over to Vanilla and type:

```
FTP Chocolate:OSBU' NORTH Retrieve/C FTP.bcd FTP.symbols
```

Alternatively, you could start an FTP server on Vanilla; then issue the following command to Chocolate:

```
FTP Vanilla Store/C FTP.bcd FTP.symbols
```

The latter approach is recommended for transferring large groups of files such as "***.bcd**" (since expansion of the "*" will be performed by the Executive).

- To retrieve **<System>Network.txt** from the server and store it on your disk as **Directory.bravo**, and store **RTP.mesa**, **lb.mesa**, and **BSPStreams.mesa** on **<DRB>** with their names unchanged:

```
FTP server Connect/C drb MyPassword Retrieve/S <System> Network.txt  
Directory.docStore/C RTP.mesa lb.mesa BSPStreams.mesa
```

- To retrieve the latest copy of all **.bcd** files from the **<Mesa>Defs>** directory, overwriting copies on your disk:

```
FTP server Retrieve/C <Mesa>Defs>'*.bcd
```

(The single quote is necessary to prevent the Executive from expanding the "")

- To update your disk with new copies of all **<Mesa>** files whose names are contained in file **UpdateFiles.cm**, requesting confirmation before each retrieval:

```
FTP server Directory/C Mesa Ret/>V @UpdateFiles.cm@
```

- To store all files with extension **.mesa** from your local disk to **<my directory>** on the file server (the Executive will expand "***.mesa**" before invoking FTP):

```
FTP server dir/c <my directory>Store/C *.mesa
```


File Tool

The File Tool provides a means for you to manipulate files on your local disk as well as on remote file systems. It allows you to retrieve, delete, list, and copy files.

10.1 Files

The File Tool is built in. You will find it in your Inactive menu, unless specified elsewhere in your **User.cm**.

10.2 User interface

The File Tool communicates through a form subwindow, a command subwindow, and a List Options window. Below is an illustration of a File Tool with the List Options window displayed:

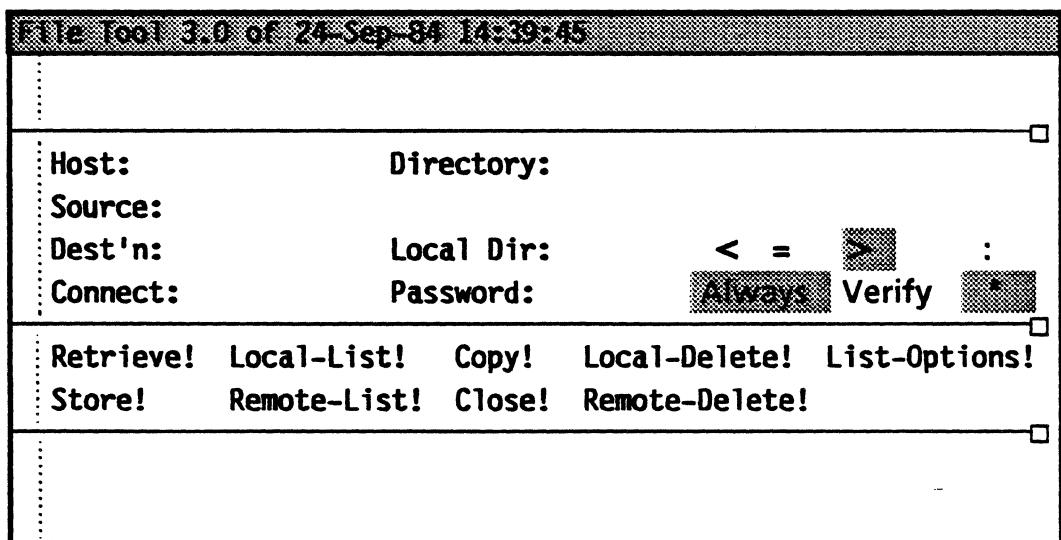


Figure 10.1: File Tool window

10.2.1 Form subwindow

The fields that can be used as arguments to a command are listed in the form subwindow:

Host:	is the name of the host to be used for remote files and operations. The Profile domain and organization are appended to Host if none are specified.
Directory:	is the default remote directory.
Source:	is a list of files (separated by spaces or returns) for the next command to act upon. File names may include wildcard/expansion characters (see the Wildcard/expansion characters section). Any files appearing in this field should conform to the syntax of file names for the file system that is the source of the transfer.
Dest'n:	is the file name for the destination of a transfer. It should conform to the syntax of file names for the file system that is the destination of the transfer.
LocalDir:	means that all references to the local disk will only occur within this directory. If the directory is not a complete path name (i.e., if it does not begin with <), it is assumed to have a <> prepended.
Connect:, Password:	this feature is not implemented.
'*	means that in remote commands (Retrieve , RemoteList , RemoteDelete), * characters in Source should be treated as if they were quoted (i.e., they should be expanded remotely instead of locally). The default is TRUE .
>	means "only store or retrieve the file if the destination exists and the source is newer than the destination (comparing creation dates)." The default is FALSE .
<	means "only store or retrieve the file if the destination exists and the source is older than the destination (comparing creation dates)." The default is FALSE .
=	means "only store or retrieve the file if the source is the same as the destination (comparing creation dates)." The default is FALSE . "Not equal" can be specified by turning on both < and >.
Always	conditions the above three commands (>, <, =) to also act if the destination file does not already exist.
Verify	requests confirmation for each file transfer. The default is FALSE .

10.2.1.1 Wildcard/expansion characters

The File Tool interprets some of the characters in **Source** as wildcard or other expansion characters. It uses the same mechanism as the Executive in expanding these characters. (See the Executive: Command line expansion section for a further explanation of local wildcard/expansion characters.)

- ' (single quote): treats the character following the single quote as if it were not a file name expansion character. The single quote is removed from the file list.
- @ (at-sign): takes the file to be an indirect file and uses its contents as a list of files if @ is the first character of the file name. This list of files replaces the indirect file in the list of files. Indirect files may nest.
- ↑ (up-arrow): removes the up-arrow character and the character following it from the file list.

The wildcard * matches zero or more characters in a file name. For example, *.mesa matches all file names ending with the extension .mesa in the specified local or remote directory. # matches any *single* character in a file name.

The * can also be used to expand across directory boundaries. In the remote case, a * as the only character of the final subdirectory in the Directory field directs the search down through all subdirectories. For example, **Directory:** <Mesa>* and **Source:** *.bcd matches all .bcd files in or below <Mesa>. In the local case, ** in the Source name achieves this. For example, **LocalDir:** <>Tools> and **Source:** **.archiveBcd finds all .archiveBcd files in or below the <>Tools> directory.

10.2.2 Command subwindow

The fields in the command subwindow are as follows:

- | | |
|----------------------|--|
| Retrieve! | transfers the file name specified in Source from the remote file system to the local disk. You may designate multiple files by the use of '*' only to the extent that the remote server supports it. If Dest'n is blank, the file name of the copy made on the local disk is the source file name stripped of all host and directory qualifiers. |
| Store! | transfers the file name specified in Source from the local disk to the remote host. Development environment file name conventions apply to the local file. |
| Local-List! | lists all files on the local disk corresponding to the name in Source . |
| Local-Delete! | deletes the files specified in Source from the local disk. If for any reason a file cannot be deleted, that file is skipped and processing continues with the rest of the files in the list. |

Remote-List!	lists all files on the remote file system corresponding to the name in Source . This must conform to the file-naming conventions on the remote host. You may designate multiple files by the use of '*' only to the extent that the remote server supports it.
Remote-Delete!	deletes the file name specified in Source from the remote file system. You may designate multiple files by the use of '*' only to the extent that the remote server supports it.
Copy!	copies the local file in the Source: field to the local file in the Dest'n: field. The Copy! command operates only on the local disk. Only single files can be specified.
Close!	closes any currently open connection, freeing any resources needed to maintain it.
List-Options!	creates a List Options window if one does not already exist.
If Verify is TRUE , then for each file that might be transferred, the following commands are displayed:	
Confirm!	do the operation.
Deny!	don't do the operation.
Stop!	don't do the operation and terminate the command. This may take some time while the termination is negotiated with the server.

10.2.3 List Options window

The List Options window is created by the **List-Options!** command. The properties that will be displayed, in addition to the file name, by a **Local-List!** or **Remote-List!** are governed by the Booleans in this window. After changing the options, invoke **Apply!** to effect those changes. The **Abort!** command will restore the options to what they were before the **List-Options!** command was invoked. Both **Apply!** and **Abort!** perform the appropriate actions and then destroy the **List-Options** window.

10.3 User.cm

The **User.cm**, in addition to the standard **InitialState**, **TinyPlace**, and **WindowBox** entries, includes:

[FileTool]

SetOptions:	A list of the Boolean options to be initialized to TRUE . Any option not appearing will initially be FALSE . The following desired options must be separated by one or more spaces and may appear in any order: QuotedStar Greater Less Equal Always Verify Type Create Bytes Write Author Read
--------------------	--

10.4 Operational notes

The actual file transfer takes place in a background process, so you are free to issue other commands or even change the values in the parameter subwindow without affecting the command currently executing. The command subwindow is cleared so that a second command cannot be invoked while one is under way. Changing a field while the File Tool is waiting for **Confirm!** will not affect the name of the **Dest 'n:** file; you should abort the transfer and re-issue the command with the desired field already set. It is important to remember that the commands are postfix; for example, fill in the **Host:** and **Source:** fields before invoking the **Retrieve!** command.



Floppy commands

The Floppy commands allow you to store and retrieve files on floppy disks using your workstation's floppy disk drive. Files larger than a single floppy disk may be written as several pieces on several disks and later put back together.

11.1 Files

The Floppy commands are built in; no additional files are needed.

11.2 User interface

The Floppy commands run in the Executive. The Executive command **Floppy.~** has several subcommands, each of which takes arguments. The command line format is

Floppy.~ <command> <arguments>.

11.2.1 Common argument definitions

Several of the commands take lists of files as arguments. The following definitions will simplify the explanations of these commands:

<fileList> consists of a list of file names to be operated upon, separated by spaces. If a file name is followed by the /s switch, the next name is used as the destination of the file transfer.

<wildList> consists of a list of file names separated by spaces. The names may contain * and # characters to match multiple files. Remember that * and # must be quoted to avoid being expanded by the Executive.

11.2.2 Commands

There are six Floppy commands. They may be abbreviated to any unique initial substring.

Delete <wildList>

deletes the specified files from the floppy disk.

Format <name>/n <number>/f

prepares a new disk for storing data. This command must be used on new disks before any data can be stored on them. It may also be used to erase all the data on a disk. The name and number arguments are optional and may be specified in either order. *<name>* specifies the name to be assigned to the floppy; you may include special characters (such as a space) in a name by enclosing it in double quotes. *<number>* specifies the maximum number of files that you may store on the floppy; the default value is 64. The Format command will ask for confirmation if there appears to be valid data on the floppy.

Info

gives information about the floppy. This consists of the name of the floppy, the number of free pages, and the size of the largest contiguous group of free pages. Since files on the floppy must be written on contiguous pages, this last number is the size of the largest file that may be written on the floppy. One extra page is added to each file to hold system information, such as the creation date.

List/<switches> <wildList>

displays the names of the specified files on the floppy. If the *<wildList>* is omitted, all files on the floppy are displayed. The *<switches>* specify additional information to be included for each file as follows:

- /d** displays the creation date of each file.
- /l** displays the length of each file in bytes.
- /t** displays the **File.Type** of each file as a decimal number.
- /w** displays the write date of each file.
- /v** (verbose) displays all of the above information.

Read <names>

copies files from the floppy to your rigid disk. *<names>* may be either a **fileList** or a **wildList**.

Write <number>/t <fileList>

copies files from your rigid disk to the floppy. You can get the effect of a *<wildList>* using the Executive's file name expansion. If *<number>/t* is present, subsequent files will be written on the floppy with **File.Type** equal to *<number>* (see the *Pilot Programmer's Manual* for a discussion of **File.Type**). You cannot overwrite an existing file on the floppy; you must delete the old copy before writing a new one.

11.3 Partial files

A double-sided, double-density, eight-inch floppy can store about 2200 pages (512 bytes each) of data. Larger files must be broken into several pieces and written on several disks and then put back together later. To specify partial files, the **Write**, **List**, and **Read** commands use an interval notation similar to that of the Mesa language and debugger. These intervals are appended to the names of files for a **Write** command and are shown by the **List** and **Read** commands. The **Read** command automatically writes data into the correct pages of the destination file on the rigid disk. Three forms of the interval are allowed:

[firstPage..lastPage] gives the inclusive range of pages.

[firstPage!count] is equivalent to **[firstPage..firstPage+count-1]**

[firstPage] defaults **lastPage** to be the end of the file.

11.4 Examples

Floppy Format "Backup Disk"/n 100/f -- name of disk and number of files specified.

Floppy Write User.cm *.mail *.mesa -- write files using Executive to expand <fileList>.

Floppy Write HugeFile[0!2000] -- write the first 2000 pages.

Floppy Write HugeFile[2000] -- write the rest of the file.

Floppy Write 4290/t Gacha/s Xerox.XC82-0-0.Gacha
-- prepare a font for a print server.

Floppy Read Foo.mesa/s OldFoo.mesa -- retrieve and rename file.

Floppy List/dl '*.mesa -- list all ".mesa" files with creation date and length.

11.5 Error messages

Most of the error messages from the Floppy commands are self-explanatory; however, two messages need further explanation:

unexpected Floppy.Error[code]

means that the floppy software raised `Floppy.Error`. See the description of the `Floppy` interface in the *Pilot Programmer's Manual* for the meaning of `code`; most of the values are self-explanatory.

unexpected AccessFloppy.Error[code]

means that the floppy software raised `AccessFloppy.Error`. The `AccessFloppy` interface is not documented, but the values of `code` are self-explanatory.

Search Path Tool

The Search Path Tool, which is built into CoPilot and Tajo, is used to inspect and change the file system search path. The introduction of this section explains how to construct legal file names. The *Mesa Programmer's Manual* documents the XDE file system.

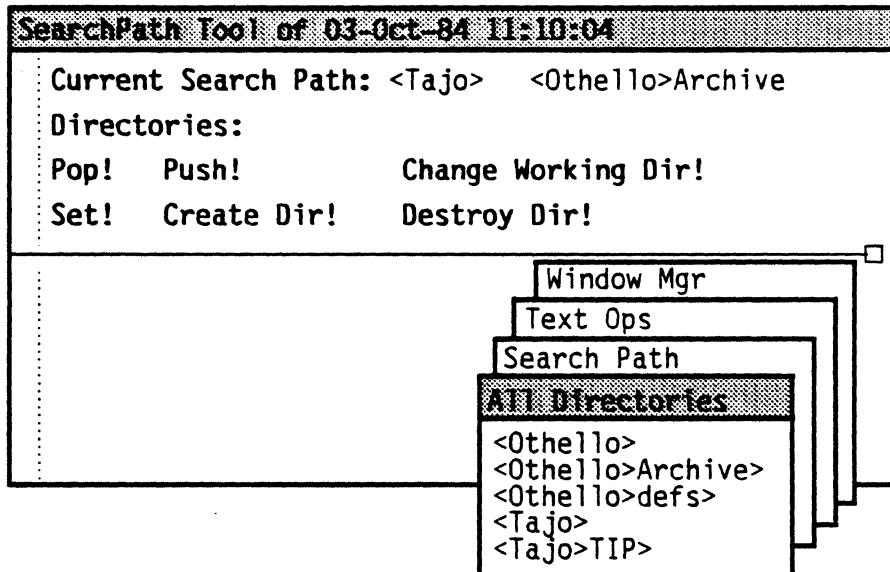


Figure 12.1: Search Path Tool window

12.1 User interface

The Search Path Tool consists of two subwindows: a form subwindow and a log subwindow.

12.1.1 Form subwindow

Arguments to Search Path Tool commands are either single directories or an entire search path. In either case, it is not necessary to qualify subdirectories fully if the corresponding root directory is on the current search path. If subdirectory names are not fully qualified, they will be interpreted in the context of the current search path.

CurrentSearchPath:	is the field where the current search path is displayed.
Directories:	is the argument field for search path commands. Create! , Destroy! , Pop! and Push! expect a single directory; Set! expects a search path, which is specified by one or more directories.
Set!	sets the search path to the list of directories appearing in the Directories: field.
Create!	creates the directory appearing in the Directories: field.
Destroy!	deletes the directory appearing in the Directories: field.
Pop!	pops the working directory, eliminating it from the current search path, and leaving the next directory in the search path as the working directory.
Push!	pushes the directory in the Directories: field to the front of the current search path.
Change Working Dir!	substitutes the directory in the Directories: field for the directory in front of the current search path.

Note: Commands for manipulating the search path are also registered by the Executive (see the chapter on the Executive).

12.1.2 Directories menu

The Directories menu is a list of all existing directories on currently open volumes. It is automatically maintained and reflects the creation and deletion of new directories, as well as opening and closing of volumes. When an item is selected from this menu, its value is pushed onto the current search path.

12.1.3 Search Path menu

The Search Path menu is a list of the directories that make up the current search path. Selecting an item from this menu removes it from the current search path.



Compare

Compare examines a pair of text files and summarizes the differences between them. The files can be either local or remote.

13.1 Files

Retrieve > **Compare.bcd** from the Release directory.

13.2 User interface

Interaction with Compare is available via the Compare Tool window or the Executive window.

13.2.1 The Compare tool window

The Compare Tool communicates through a message subwindow, where information and error messages are posted; a form subwindow, where the **Compare!** command and options are listed; and a file subwindow, where the results of the comparison are displayed. Figure 13.1 is an illustration of the Compare Tool with the switches set to the default values.

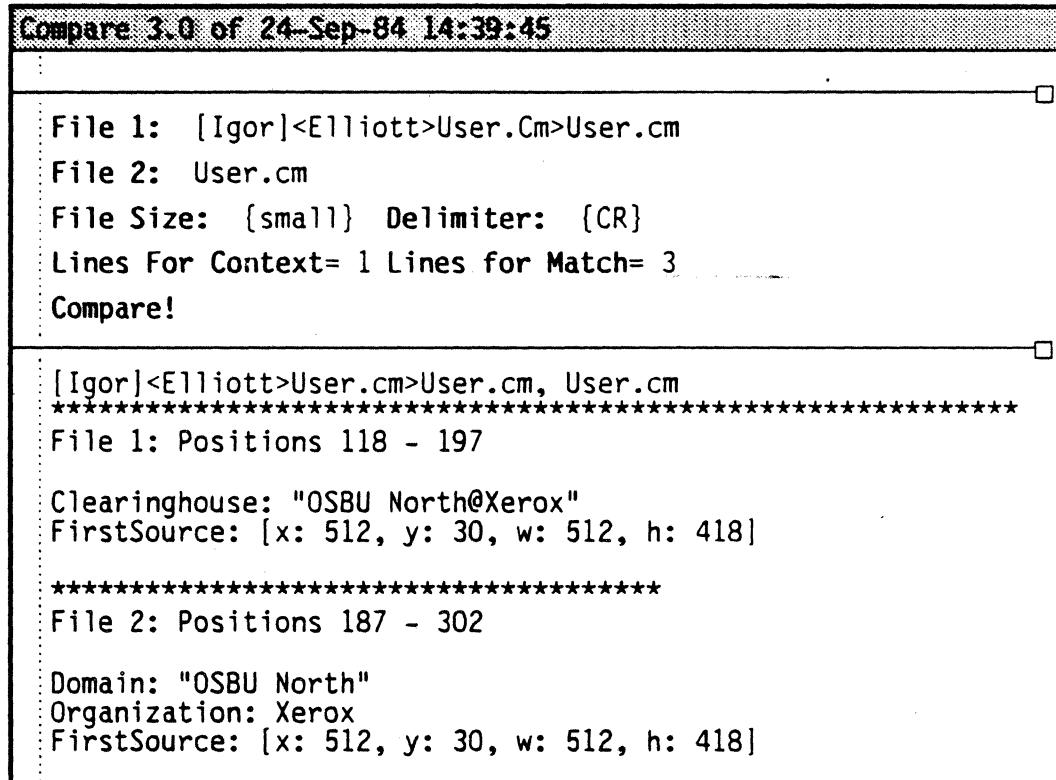


Figure 13.1: Compare Tool window

13.2.1.1 Form subwindow

The **Compare!** command and the fields that can be used as arguments are listed in the form subwindow.

Compare! compares the source files specified in the **File1** and **File2** fields and displays the difference summary in the file subwindow.

File1 , File2: files to be compared.

File Size = approximate size in pages of the source files to be compared. **File Size** is an enumerated type: {**small**(10 pages), **medium**(30 pages), **large**(50 pages)}. **Medium** is the default file size.

Delimiter: determines whether a statement will be defined to terminate with a carriage return (**CR**) or a semicolon (;). For example, if **Delimiter** is set to semicolon, then

```
index ← index + 1; GOTO exit; (CR)
```

will match

```
index ← index + 1; (CR)
```

```
GOTO exit; (CR)
```

Delimiter is an enumerated type: {**CR**, **semicolon**}. **CR** is the default delimiter.

Lines For Match = minimum number of lines to define a match. Default = 3.

Lines For Context = number of trailing lines to output for context. Default = 1.

13.2.1.2 File subwindow

The File subwindow displays the differences between the text files specified in the **File1** and **File2** fields. The difference file contains the names of the two files being compared and a list of lines in which they differ. The differing lines are reported in context and are preceded by a character position range that encompasses the character positions of the differing line(s) and the adjacent contextual line(s). Note that blank and empty lines are ignored during the comparison. The file associated with this window is **Compare.log**.

13.2.2 Compare via the Executive window

Compare also runs in the Executive. Here, a list of text file pairs may be given. The differences between each pair of text files are recorded in files created by Compare. The name of each difference file is obtained by appending .dif to the name of the first file in the pair, excluding its extension. If the two files of a pair are identical, or if one of them is empty, no difference file is generated. If the first file of a pair is an editor back up file, the \$ will be incorporated into the name of the difference file before the .dif extension.

The difference file contains the names of the two files being compared and a list of lines in which they differ. The differing lines are reported in context and are preceded by a character position range that encompasses the character positions of the differing line(s) and the adjacent contextual line(s). Note that blank and empty lines are ignored during the comparison.

13.2.2.1 Command line

Compare is invoked by typing a command of the following form to the Executive:

```
>Compare /FilePairSwitches file1 file2.../FilePairSwitches filen-1  
filen
```

13.2.2.2 File pair switches

The optional switches are a sequence of zero or more letters preceded by a slash(/). Each letter is interpreted as a separate switch designator and each may optionally be preceded by - or ~ to invert the sense of the switch. # denotes a decimal number. The switches are:

#m minimum number of lines to define a match. Default = 3.

#c number of trailing lines to output for context. Default = 1.

#b approximate size in pages of the source files to be compared. Default = 30 pages.

s determines whether a statement will be defined to terminate with a carriage return (**CR**) or a semicolon (;). **CR** (/s) is the default delimiter. For example, if **Delimiter** is set to semicolon (/s), then

```
index ← index + 1; GOTO exit; (CR)
```

will match

```
index ← index + 1; (CR)
```

```
GOTO exit; (CR)
```

13.2.2.3 Examples

```
>Compare file1 file2 file3 file4
```

Compare **file1** to **file2** and **file3** to **file4** using default switches.

```
Compare /5m3c file1 file2
```

Compare **file1** to **file2** using five lines as the criterion for a match and output three trailing lines for context.

```
Compare /15b file1 file2
```

Compare **file1** to **file2**; both files are approximately 15 pages in length.



Find

Find is a program that looks for a pattern in a list of files and prints the position within the file and the line in which the pattern occurs. Remote files are specified using the standard **[server]<directory>filename** notation.

14.1 Files

Retrieve **Find.bcd** from the Release directory.

14.2 User interface

Find is invoked by typing a command of the following form to the Executive:

```
>Find pattern/global-switch file1/local-switch...filen/local-switch
```

where each **pattern** is a string of characters not containing a blank, tab, or slash (/). If any of these special characters is to appear within a pattern, the pattern must be enclosed within double quotes. Certain other characters have special meanings within a pattern, as described below.

Note: Because the Executive recognizes *, #, ?, TAB, CR, ↑, @, ; and ' to have special meaning, any of these characters within patterns or remote file names must be preceded by a single quote (see the Executive chapter).

14.2.1 Switches

If there is more than one pattern, each but the first *must* be given a switch (either /c or /~c), since **file₁** is taken to be the first string, following the first pattern, that has none of the pattern switches listed below. The pattern switches are:

- c Ignore upper- and lower-case distinction when pattern matching (default FALSE). This is the only switch that may be negated.
- i Interpret the string not as a pattern, but as a set of characters to be ignored throughout the input file(s). For example, -/i would cause all hyphens to be

ignored, thereby letting you search for one or more words that may or may not be hyphenated within the files. The default is that no characters are ignored.

- o Interpret the string not as a pattern, but as the name of a file in which to write the matches. The test of the command line-up through the first file name is included at the beginning of the output file. If the file already exists, overwrite it. The file named must be local.

14.2.2 Switches on file names

- h Use this name as a default host name for all subsequent file names, until either the end of the command is reached or another default host is specified. If this switch appears without a host string, no default is applied to subsequent names.
- d Use this name as a default directory name for all subsequent file names, until either the end of the command is reached or another default directory is specified.

14.2.3 Special characters

Within a pattern, the following special interpretations apply. All but the last also apply within the text accompanying a /i switch.

- [xyz] Matches any characters x, y, and z (or X or Y or Z, if contained within a pattern that has the /c switch).
- # Matches any single character.
- | Matches any "white space" character (CR, LF, TAB, SP, or FF).
- ~x Matches any character except x, where x can in turn be one of the special forms. For example, ~[0123456789] matches any non-digit, and ~| matches anything except a white space character.
- =x Matches the character x, even if x is one of these special characters. Thus =[matches a left bracket, and == matches a single equals sign. Also, =Q matches 'Q' but not 'q', even if the pattern is given a /c switch.
- \n,etc. Matches a single character as defined for Mesa strings. Thus, \n matches a CR, \t matches a TAB, and so forth. If the character following the \ is not one of the recognized forms, the \ has the same effect as an =.
- x* Matches any number (including zero) of repetitions of x. Again, x can be one of these special constructs; thus, ~[0123456789]* matches zero or more non-digits. Note that only single-character patterns can be repeated; there is no way to match "zero or more iterations of the string 'abc'."

14.3 Examples

```
>Find system user.cm [server] <doc> spiffy.cm
```

Print the lines containing "system" (ignoring case distinction) and the corresponding character positions within the local files **user.cm** and within the remote files **[server]<doc> spiffy.cm** and **[server]<doc> crulty.cm**.

```
>Find OPEN/~c HackOpens/o Oldhack.mesa Newhack.mesa
```

Determine the lines and positions within **Oldhack.mesa** and **Newhack.mesa** that contain the pattern "**OPEN**" (all capitalized) and write them to the file **HackOpens**.

```
>Find ": CARDINAL" DudleyDriver.mesa
```

Print all declarations of long and short cardinals within **DudleyDriver.mesa**.

```
>Find Allocate'*NodeStorage*.mesa [server] <defs> '*.mesa
```

Print the lines and character positions matching the pattern **Allocate**-anything-**Node** from the local files matching **Storage*.mesa** and the remote files matching **[server]<defs> '*.mesa**. Note that this pattern would in fact produce a match against something of the form

```
AllocateStuff[zone: myZone, node: myNode];
```

or even

```
AllocateBins[...];
<<several lines of stuff>>
FreeNode[...]
```

The position and line containing the *end* of the match are printed. If what you really wanted was to see calls to procedures named **AllocateNode**, you could use the pattern **Allocate~=['*Node**.

```
>Find .NEW/~c .FREE/~c MakeNode/~c FreeNode/~c GarbageImpl.mesa
```

Show all heap allocations and deallocations with **GarbageImpl.mesa**

```
>Find RECORD~';'*FooType MumbleDefs.mesa MumbleImpl*.mesa
```

Show all record declarations that contain an element of type **FooType**. (You might miss some if a record declaration includes a comment containing a semicolon.)

```
>Find |/i ~[=[==,<-]"~:c`BadGuys/o [server]<StarSource>'*.mesa
```

Produce a file containing all instances of non-local string literals in a set of remote files, assuming that all string literals are preceded by a left bracket, an equals sign, a comma, left arrow, or a colon, possibly with some intervening white space. The pattern says to search for a quote character *not* preceded by any of those characters, and ignoring white space, thereby matching only closing quotes. Thus, the result is to find closing quotes that are not followed by 'L' or 'I'. (The /c switch is used to save having to remember whether lower-case 'l' is accepted by the compiler.) Note that this pattern will overlook strings in which the last non-white space character is a left bracket, equals sign, etc. (The syntax has its limits.)



File window

A File window is used to view and edit a text file.

15.1 Files

The ability to create File windows is built into the Xerox Development Environment.

15.2 User interface

The File window interacts through a text subwindow. It can be opened by choosing **FileWindow** in the ExecOps menu. The ExecOps menu is available from the root window, outside all other windows. The window name frame contains useful information about the state of the File window. For example, when the File window comes to the screen, the window name frame says **Empty Window**. When a file is retrieved into the empty window, the text in the window name frame changes to display the name of the file.

15.2.1 DebuggerOps menu

The DebuggerOps menu belongs to a File window. The DebuggerOps menu contains the following commands. (For more information, refer to the Debugger chapter.)

- Attach** tells the debugger to ignore the time stamp in the source file when setting breaks.
- Break** uses the current selection to set a breakpoint. If you select **PROCEDURE** or **PROC**, a breakpoint is set on the entry to the procedure; if you select **RETURN**, a breakpoint is set on the exit of the procedure; otherwise a breakpoint is set at the closest statement enclosing the selection.
- Clear** clears the breakpoint or tracepoint at the specified location.
- Trace** sets a tracepoint at a specified location. Confirmation is given by moving the selection to the place at which the tracepoint is actually set.

15.2.2 FileWindow menu

The FileWindow menu belongs to a File window. The commands available in the menu depend on the state of the File window. The File window may be in one of three states: empty, non-editable, and editable. The menu commands available for each state and a description of each command are:

Empty: **Create Destroy Load Store Time**

Non-Editable: **Create Destroy Edit Reset Load Store**

Editable: **Create Destroy Reset Load Store Time Save**

Create makes a new File window at the place selected by clicking **POINT**. There is no explicit maximum number of File windows.

Destroy removes the File window in which the command was invoked. When you invoke **Destroy**, a symbol of a mouse appears. Clicking **POINT** confirms the command; **ADJUST** aborts it. Invoking **Destroy** will not remove a File window when a file is being edited.

Edit enables editing of the currently loaded file. The **Edit** command is available only if a file has been loaded into the window. The window name frame changes to read **Editing: filename**. A scratch file, **filename\$\$**, is created during the editing as the edit log; this file is not automatically deleted when the editing has been completed.

Load displays a file in the window, using the current selection as a file name. An accelerator for loading files is provided: typing the **DOIT** key in an empty window causes the file named by the contents of the window to be loaded. If a file name extension is not provided, the system first looks for the file name without the extension; if this is not found, it looks for **filename.Mesa**, **filename.Config**, then **filename.cm**. The **Load** command fails if the file is not found, and the display blinks. **Load** will not work while you are editing, as you would lose your edits.

Reset resets the window back to a previous state; confirmation is required only if you are editing. If you have been editing, all edits to the file are discarded and the original file is left in the window. If the file loaded in the window is not editable, then the File window is set back to an empty window.

Save stores the contents of the window that is being edited to its current file; confirmation is required. A backup "\$" file is created that is a copy of the unedited version. After the **Save** command completes, the File window is no longer editable. This command is available only when the file loaded in the window is editable.

Store creates a file whose name is the current selection and stores the contents of the window to it; confirmation is required. After the file has been stored, the file is not editable.

Time replaces the current selection with the current date and time.

Note: An empty File window can only contain up to 60,000 characters.

15.3 User.cm

The following **User.cm** entries are available to create initial File windows and for symbiote initialization. Typical entries for the System and FileWindow sections are:

[System]

FileWindow: [x: 0, y: 457, w: 512, h: 321] []

FileWindow: [x: 512, y: 60, w: 512, h: 448] [x: 300, y: 778]

FileWindow: [x: 512, y: 30, w: 512, h: 247] [x: 904, y: 778] Calendar/t

[FileWindow]

Menu: Create Edit Load Position Reset Save Split Store Time Wrap

SetUp: Always Menu Edit

FileWindow: An arbitrary number of File window entries is permitted in the System section. Each specifies a file window to be created. The first set of bracketed values indicates the position of the window when it is active. **x** and **y** are the horizontal and vertical bitscreen coordinates of the upper left corner of the window. **w** and **h** are the width and height of the window in bitscreen coordinates. Any or all of these fields may be omitted, in which case they have the following default values: [**x: 0, y: 0, w: 512, h: 400**]. The second set of bracketed values indicates the position of the window when it is tiny. **x** and **y** are the horizontal and vertical bitscreen coordinates of the upper left corner of the window. Any or all of these fields may be omitted, in which case they have the following default values: [**x: 0, y: 0**]. The next item in the line, which is optional, is the name of the file to be loaded into the window. If there is a switch on the file name, it specifies the initial state of the window (**a** for active, **t** for tiny, and **i** for inactive). Note that you must always specify the active box and tiny box position, even if they are defaulted by specifying **[]**.

Menu: specifies the commands that will be available in an editable menu symbiote.

SetUp: specifies when symbiotes are to be applied and which are desired. The entry can contain either the keywords **Always** or **Initial**, **Edit** and **Menu**. The meanings of the keywords are:

Initial Add specified symbiotes to all existing File windows.

Always Initial plus add specified symbiotes whenever a File window is created.

Edit User wants an edit symbiote.

Menu User wants an editable menu symbiote.



Print

Print converts text files to Interpress masters for printing and sends the result to a printer, such as an 8044 printer. Switches in the Print program allow you to specify how the output will look or to produce a master file without sending it to a printer.

16.1 Files

Retrieve **Print.bcd** from the Release directory. You will also need **Fonts.widths** from the **Fonts** directory.

16.2 User interface

Print runs in the Executive. The command line format is **Print <filename1>/switch <filename2>/switch <filename3>/switch....** The special filename **\$\$\$** instructs Print to print the current selection rather than a file. This is useful for printing parts of your debugger log or other small pieces of text.

Files are converted and sent to the printer; multiple files are batched and sent together to the printer. The Interpress master is written on the file **Print.scratch\$**. If the transmission to the printer fails or is aborted, you may save this file and send it later to the same or a different printer. You may specify remote files using normal remote filename syntax (**[Host]<Directory>File.ext**). Both local and remote file names may contain asterisks (*) to permit expansion to all file names that match the string provided. An * must be preceded by a quote if you are printing remote files instead of local ones.

If a local file specified in the command line is already an Interpress master, it will be sent to the printer without further conversion. Remote files are not checked for being Interpress masters, so instructing Print to print a remote Interpress master will not produce what you want.

16.2.1 Switches

Local switches (i.e., those appended to an input file name) affect the printing of that file only. Global switches affect all subsequent input files.

- /a prints headings on each page (default true; -a disables).
- /z prints footings on each page (default true; -z disables).
- <host>/h directs the output to the print server named <host> for the files that follow. The server name is qualified by your default domain and organization (from the **ProfileTool**), if necessary.
- <output>/o creates an Interpress master in <output> (extension defaults to .**interpress**) and disables transmission to the printer.
- /f changes the font to for the files that follow. The default fonts are Gacha8 in portrait mode and Gacha6 in landscape mode. (See the next section on Naming fonts.)
- /c<n> sets number of copies to be printed to <n> (default 1).
- /t<n> changes the tab stops to <n> spaces (default 8).
- /l<n> specifies landscape orientation (long edge of paper horizontal). <n> is the number of columns (default 2).
- /p<n> specifies portrait orientation (long edge of paper vertical). <n> is the number of columns (default 1).
- /s<n> specifies number of sides. <n> can be 0, 1, or 2; 1 and 2 request single- and double-sided printing respectively; 0 means let the printer decide how to print the document.

Examples:

```
Print filename          -- produce a master for filename in
                        the default mode and send it to the
                        default printer.

Print filename/l        -- print filename in landscape mode.
                        /l is a local switch.

Print /l filename1 filename2 -- print two files in landscape mode.
                        /l is a global switch.

Print filename1 filename2/l3c3 Classic10BI/f filename3/l
                        -- print filename1 in the default
                        mode; then three copies of filename2
                        in three-column landscape; then one
                        copy of filename3 in two-column
                        landscape using font Classic10BI.
```

Print \$\$\$/p

-- print the current selection in portrait mode.

16.2.1.1 Naming fonts

Font names consist of three parts: family, point size, and face. Families are spelled out, point sizes use digits, and faces are encoded. Print has no knowledge of which fonts are available; contact your System Administrator to find out what fonts are available on your printers.

Examples:

Families: **Classic, Modern, Gacha, Titan**

Point size: **10**

Face: **B** (bold), **I** (italic), **BI** (bold italic)

Thus **Classic10BI** specifies the 10 point size of the Classic font with bold italic face.

16.2.2 Defaults

The following defaults may be overridden by switches or **User.cm** entries:

1-column portrait

Font = Gacha8

1 copy

Headings and footings printed on each page

TAB stops set at multiples of 8 spaces (Note: space width is a function of the font)

Use printer's default for number of sides

Some settings that cannot be changed are:

Portrait mode margins = $\frac{3}{4}$ inch on all sides

Landscape mode margins = $\frac{3}{4}$ inch top, $\frac{1}{2}$ inch others

Space between columns = $\frac{1}{2}$ inch

Heading and footing text = file name, creation date , and page number

Page number location = at right margin when heading or footing specified.

16.3 Formatting

Print automatically determines line, column, and page breaks (only $8\frac{1}{2} \times 11$ inch paper is supported). Long lines are broken at white space, and the continuation line is indented the same as the original line up to a maximum of half the column width. To force a new column, put a form feed character (**CONTROL-L**) in your text. Print will begin each file on a new sheet of paper. Note that files formatted for single-sided printing and later printed on both sides may not start on new sheets.

16.4 User.cm entries

Print initializes several of its parameters from the [Hardcopy] section of your **User.cm**.

[Hardcopy]

Interpress: "My Printer"

-- name of your Interpress printer; quotes are necessary if the name contains spaces.

PrintedBy: Deliver to \$, room 123

-- This string is sent to the printer to appear on the banner page. The "\$" is replaced by your name (from the **ProfileTool**); the remainder of the text is literal.

LandscapeFont: Gacha6

-- default font for landscape printing.

PortraitFont: Gacha8

-- default font for portrait printing.

Orientation: Portrait

-- or Landscape

Columns: 1

-- number of columns in your default orientation.

Your default domain and organization from the **ProfileTool** will be used to qualify the name of your printer, if necessary.



III

System-building tools

This chapter describes how a typical program might be built in the Xerox Development Environment. It describes and illustrates common applications of the most common functions: compiling, binding, running, and debugging a system. It also briefly discusses the concepts of packaging a system and making bootable files. This chapter should be viewed as a base point from which to build familiarity and expertise with programming in the development environment. The last part of the chapter briefly describes each of the program-building and analysis tools.

III.1 Files

Many of the examples in this chapter are based upon two Mesa modules, Lexicon and LexiconClient, which are roughly equivalent to those found in chapter 7 of the *Mesa Language Manual*. These modules are part of a simple string management system called Lex. They can be retrieved from the release directory along with several other files that are needed to complete the Lex system.

```
LexiconDefs.mesa (.bcd) -- interface source and object file  
Lexicon.mesa (.bcd) -- source and object file for Lexicon  
LexiconClient.mesa (.bcd) -- source and object file for the Client  
Lex.config -- binding configuration file  
Lex.pack -- packaging specification file  
Lex.bcd -- object file for complete system
```

III.2 Creating a source file

Creating a source file is similar to creating a text file. The code can be typed into any file window and saved. Conventions for how this code should be ordered and how comments should be notated are described in the *Mesa Language Manual*.

Mesa source code is easier to read when appropriately formatted. Please refer to the chapter about the Formatter for more information about how to format source code files.

Note: Remember that Mesa has both description modules and implementation modules. Compiled descriptions and implementations must be bound together before they can be

executed. Later sections describe the compiling and binding processes in more detail, as do the chapters on these individual tools.

III.3 Creating an object file

After creating an executable object file, the first step is building a component. The next two steps are usually compiling and binding. Though they may have to be repeated many times to create a large system, the way they are used is relatively invariant, as is described in the following subsections.

III.3.1 Compiling a program

Invoking the compiler is normally done in one of two ways. The first is to enter a command line to the Executive:

>Compiler source1 {source2 source3...}*

This command causes sources listed to be compiled into separate object files.

The second way to invoke the compiler uses Command Central (refer to the chapter on Command Central). After selecting the **Compile:** item, a type-in point appears and the source file name(s) may be entered:

Compile: source1 {source2 source3...}

To run the compiler, invoke **Compile!** The compiler always assumes .mesa extensions to the file names if no extension is given.

A successful compilation results in object files named **sourcename.bcd**. If the Compiler discovers a syntax error in the source, it will logically insert or delete what it thinks is appropriate text so that it can continue compiling the program. A summary of the errors and warnings is written in a file named **sourcename.errlog**, and no object file is produced. Errors and warnings are reported in the form *procedure[character-position-in-file]*, with an indication of the type of error and what text has been logically inserted or deleted. A program will run with warnings, but it is not recommended.

You may specify operational options in the form of Compiler switches. For example, the "b" switch specifies that the compiler should generate code to do array and subrange bounds checking. In general, switches turn on or off some runtime checking or optimization feature. The switch set **/-b-ej-np-u** is commonly used to compile programs that have already been debugged and are ready to use; the switch set **/-ep** is more common for programs during development. The first set disables most runtime error checking and enables some optimization; the second set enables all the runtime checking code and disables some of the optimization. The switch sets given above are merely suggestions, not rules. (For a complete list and definition of the switches, as well as their default values, see the Compiler chapter's section on Compiler Switches.)

There are several ways to set these switches, depending on how you invoke the compiler. If the Executive is used, switches may be specified either with each file name, globally for all files, or both:

```
>Compiler /-e source1/bej source2/-n-u
```

The global switches are in effect appended to each of the local switch sets; if a conflict arises, the local switches take precedence. In the example above, the effect is to apply the switch set **/bej** to source1 and the set **/-e-n-u** to source2. If Command Central is used, the switches may be given along with each file name as above. The global switches are set via the Options window, invoked by selecting **Options!**. The same precedence rules apply. In either case, defaults for any switch may be set in the **User.cm** file. (These default entries have lowest precedence; refer to the section on **User.cm** entries in the Compiler chapter).

III.3.2 Binding a configuration

Though the Binder performs a number of tasks, its main tasks are matching the **IMPORTS** of one program to the **EXPORTS** of another and binding the result. Specifically, the Binder combines modules and possibly previously bound configurations, according to the specifications in a configuration file to produce a new object file. This file may be loaded into a running system or be processed by a later invocation of the Binder or Packager. The following subsections describe a simple configuration file and show how to use the Binder.

III.3.2.1 Configuration description files

A configuration description file describes to the Binder which modules to bind and how they are to be put together. The binding configuration shown below is merely a list of the modules to be bound: Lexicon and LexiconClient. The names listed need not be of single modules but can refer to previously bound configurations.

Note that the names given are module names, not file names. Unless a **DIRECTORY** statement is used (described in the Binder chapter), the Binder assumes that the module **modulename** can be found in the file **modulename.bcd**.

```
Lex: CONFIGURATION
IMPORTS Process, Storage, String, TTY
CONTROL LexiconClient =
BEGIN
    Lexicon;
    LexiconClient;
END.
```

The **CONTROL** statement indicates which module should be started when the resulting configuration is loaded. Other modules may be explicitly started by the module specified in the **CONTROL** statement or be implicitly started when any one of its procedures is called. For the example given above, LexiconClient is started explicitly and Lexicon is started only when one of its procedures is called.

Because Lexicon relies on certain operating system support, it must have access to the interfaces through which they are provided. This is accomplished by the **IMPORTS** statement. It gives the Binder a list of interfaces that will be referenced by the modules being bound. It is an error to omit a necessary interface, and a warning results if an imported interface is never referenced.

III.3.2.2 Using the Binder

As with the Compiler, the Binder is normally invoked using either the Executive or Command Central. To use the Executive, type:

```
>Binder source1 {source2 source3...}
```

Each of the sources represents a distinct configuration description, and the command creates distinct object files. To invoke the Binder through Command Central, simply select **Bind:** and enter the source name(s):

```
Bind: source1 {source2 source3...}
```

Now that the arguments are listed, the Binder may be invoked by selecting the **Bind!** command. Regardless of the method used to invoke the Binder, a **.config** file name extension is assumed if no extension is given. Also in either case, all error messages are written to a file named **sourcename.errlog**.

You may specify options to the Binder by Binder switches. In most cases the **/c** switch is used to specify code copying. Often the **/s** switch is also specified, but there are different policies about whether to use **/s** (for a complete list and definition of the switches, see the section on Binder Switches in the Compiler chapter). If the Binder is invoked using the Executive, the switches may be given along with each file name, globally at the beginning of the line, or both:

```
>Binder /c source1/s source2 source3/s
```

When using Command Central, the switches may also be given on the command line along with each file, or global switches may be set via the Options window, invoked by selecting **Options!**. In either case, defaults for these switches may be set in the **User.cm** file, and these defaults have the lowest precedence (see the section on User.cm processing in the Executive chapter).

For more details, see the Binder chapter.

III.3.3 Summary

Summarizing the operation of the Compiler and Binder:

- Both the Compiler and the Binder can be invoked with either the Executive or Command Central, and both recognize various switches.
- The Compiler assumes input file names have an extension of **.mesa** if no extension is given, while the Binder assumes **.config**.
- Both the Compiler and Binder produce an object file if processing was successful. Otherwise, a file named **source.errlog** is created containing the errors or warnings that were issued.

- Names specified in a configuration file refer to modules, not files. It is assumed that a module name **Mod** exists in a file named **Mod.bcd**. This association can be changed with a **DIRECTORY** statement.

III.4 Running a program in the Tajo environment

Once you have created an executable object file, it has to be loaded into the runtime environment for execution to begin. This section describes how to get object files to the runtime environment (typically Tajo), and how to run them once they are there.

Often the object file to be run will already be resident on the Tajo volume, which is the case for tools that have already been developed and are present as utilities. However, while performing development tasks, programmers often work in a volume different from Tajo (usually CoPilot) for debugging convenience. When this is the case, you must move the object file to be run to Tajo using either the **Snarf** command (see the chapter on the Executive) or Command Central.

III.4.1 Snarfing and running

To snarf a file to Tajo, first get to Tajo by booting or proceeding from CoPilot. (To proceed, CoPilot must have been entered from Tajo using **CALLDEBUG**.) If you were in Othello, then Tajo must be booted.

Once in your Tajo volume, the object file can be retrieved using Snarf in the Executive:

>Snarf source.bcd

Snarf does not move the file from volume to volume but makes a new copy on the Tajo volume. At this point the program can be run by typing its name to the Executive. The **.bcd** extension is not necessary.

>source

Selecting commands from the Exec Ops menu is an alternative to use the Executive. After typing and selecting the name of the object file, you may load, start, or run it by selecting the appropriate menu item. **Load** loads but does not start an object file, **Start** starts a previously loaded object file, and **Run** loads and then starts an object file.

III.4.2 Using Command Central

The Command Central **Run!** command is roughly equivalent to the one described above. To use it, activate Command Central in the CoPilot volume and select the **Run:** item. A type-in point will appear, indicating where to enter the object file name.

Run: Lex

After entering the name, select **Run!** and Command Central does the rest: the Tajo volume is booted, and the object file is copied and then run. Like the Executive, Command Central does not require the **.bcd** extension to be entered. Various switches may be

specified to modify the operation of this command. `User.cm` entries may also be set (see the Command Central chapter's `User.cm` section).

III.4.3 Summary

To recap, to get an object file from a development volume (normally CoPilot) to a client volume (Tajo) and to run it, you may:

- Boot or proceed to get to Tajo, use the Snarf command in the Executive to copy the object file, and run the object file by typing its name to the Executive.
- Use the Command Central `Run!` command, which boots the Tajo volume, copies the object file, and runs it.

Once the object file has been copied to the client volume, it need not be recopied unless it is changed. Thus future invocations can be made directly using the Executive--no copying is required.

III.5 Making boot files

As with any program to be executed, the operating system itself requires an object file that can be loaded into memory and started. Such a file is called a *boot file*. Along with the Pilot image, the boot file also contains one or more Pilot clients, such as Tajo and the compiler. This file, containing the entire runtime environment plus the initialization code needed to start it, is loaded at boot time by a boot loader called the *germ*. There are several steps to creating a typical boot file. Some of these require familiar actions such as using the Compiler and Binder, while others require less-familiar tools such as the Packager and MakeBoot. The following subsections describe these less-familiar tools.

III.5.1 Packaging a system

The Mesa Packager can be used to improve the swapping performance of Pilot-based programs. The Packager allows you to specify the swap units for your program's code (code packs) and global frames (frame packs). For example, the Packager allows a code pack to be defined that includes the code for a collection of procedures from several different modules and a frame pack to be defined that collects the global frames of a number of modules (for example, you might pack together procedures from different modules that are not commonly used, such as initialization routines or catch code). This prevents a seldom-used procedure from remaining resident just because it is in the same module as a commonly used procedure. Similarly, commonly used procedures from many modules can be grouped together so that they have a better chance of remaining resident. Packaging a system requires detailed knowledge of the software in question and careful consideration of the packaging specification.

III.5.2 Packager operation

The Packager is a post-processor that reads a single object file and a packaging description and writes a new object file with the code rearranged as specified. Its operation resembles that of the Binder. To work correctly, all symbol files corresponding to the input object file must be on the disk. The Packager needs these files to identify procedures and frame packs and to locate the code for procedures. The output file contains the reorganized code, but not

the symbols, of the input object file (that is, the code is copied; symbols are not). The output file also contains information about the global frame packs for later use by MakeBoot and the Pilot loader. Finally, the Packager can produce detailed listings and maps of the placement of code and frame packs, as well as other information (see the Packager chapter).

III.5.3 Using MakeBoot

As stated earlier, MakeBoot converts an object file into a file that can be boot loaded; namely, a boot file. To use MakeBoot, you need the base object file from which the boot file will be built and at least one parameter file containing information about certain data structure sizes and initial memory configurations. MakeBoot allows you to specify information such as the length of the Global Frame Table and the number of processes allowed to coexist. Unlike the Packager, MakeBoot does not require any symbol files to be present on disk (see the MakeBoot chapter).

III.5.4 Summary

Summarizing the operation of the Packager and MakeBoot:

- The Packager and MakeBoot are normally used in conjunction to create a file that can be boot-loaded. Such a file typically contains the operating system (Pilot) and one or more clients.
- Run as a post-processor, the Packager provides a level of fine tuning on an object file to improve its swapping characteristics.
- MakeBoot converts an object file into a boot file according to specifications given in a separate specifications file. Parameters include Global Frame Table length and the number of coexisting processes.

If specification files (`.pack` for the Packager, and `.bootmesa` for MakeBoot) already exist, which is normally the case, using these tools is fairly simple. However, it is worth restating that both MakeBoot and the Packager are not as commonly used as either the Binder or Compiler, and creating a good specification file for either requires careful thought.

III.6 Using the Debugger

This section describes the Pilot-based interactive Mesa Debugger, CoPilot. CoPilot supports source-level debugging: it allows you to set breakpoints, trace program execution, display the runtime state, and interpret Mesa statements. CoPilot is intended for experienced programmers familiar with Mesa. The annotated examples in this section are both examples of form and suggestions for dealing with situations that commonly arise while debugging. (The Debugger chapter describes CoPilot in detail).

III.6.1 Invoking CoPilot

There are several ways to invoke the Debugger. For example, in Tajo or CoPilot, pressing `CALLDEBUG` interrupts your program. In the course of running your program, you may also

enter the Debugger for several other reasons. There is a different cursor icon for each reason.

- Some currently running module generates a **SIGNAL** or **ERROR** that no procedure catches. The *Unc Sig* cursor is displayed, representing **Uncaught Signal**.
- A module explicitly requests to go to the Debugger. Pilot makes such a **Call Debugger** request when handling address and write-protect faults. The cursor displayed is *Call Debug*.
- The Debugger has been used to specify a point in the source program where execution should be stopped and the Debugger entered. Such a point is called a **BreakPoint** and is denoted by the cursor, *Brk pt*.
- To maintain a consistent map of the client's virtual memory, CoPilot must be invoked periodically to update internal data structures. Called **Processing VM Map**, it is automatic and requires no user intervention. The cursor says *Map Log*.
- If CoPilot is entered due to a **CALLDEBUG**. The cursor will be *Int -- (Interrupt)*.

III.6.2 Talking to the Debugger

The user interface to the Debugger controls a command processor that invokes a collection of procedures for managing breakpoints, examining user data symbolically, and setting the context in which user symbols are referenced. The command processor accepts character input and extends the input to the maximal unique string that it specifies. For instance, an **L** in response to the **>** prompt will be extended to **List**, just as a **P** will be extended to **Proceed**. Typing a question mark during command entry will result in a list of the valid options with the command characters shown in upper case. Typing a space in response to the **>** prompt invokes the CoPilot interpreter, which will be described later. (For further information on debugger I/O conventions and the CoPilot interpreter, see **Debugger I/O Conventions** in the Debugger chapter.)

III.6.3 Debugging a client program

The following sample session demonstrates CoPilot commands commonly used in debugging a client program. The component files of **Lex**, the configuration in our example, are listed at the beginning of this chapter. The sample configuration **Lex** consists of two modules, **Lexicon** and **LexiconClient**. Let us assume that the configuration has been bound, loaded, and started in Tajo, and you have interrupted the program and entered CoPilot for the first time (by holding down **CALLDEBUG** after the program started). You get the current date and time, a message indicating why you entered CoPilot (in this case, interrupting the program), and a prompt for the first command:

```
6-Jan-82 14:59
*** interrupt ***
>
```

III.6.3.1 Setting the context

CoPilot allows you to specify a referencing environment, or context, in terms of Mesa configurations and modules. To get to a context from which breakpoints may be set in one

of the modules in **Lex**, let's first check to see which configurations have been loaded by typing:

```
>List Configurations
```

which responds with:

```
Lex
Print
CommComSoft
    CourierConfig
FloppyCommands
XComSoft
MailStubConfig
AuthStub
CHStub
NSStringConvertConfig
NSStringConfig
NSDataStreamConfig
NSSessionImpl
NSFilingRemoteConfig
    NSFilingCommonConfig
NSFileTransfers
FileTransfers
RightsNotice
StartIncludedBcds
BasicHeadsDLion
Tajo
    HideIntermediateExpRecs
    PilotKernel
        Control
        MesaRuntime
        Misc
        Store
            ResMemMgr
            VMDriver
            FileBasics
            FileMgr
            VMMgr
        DiskDrivers
        UserTerminalDriver
Loader
Communication
    Level0
    Level1
    Level2
SubTajo
Wisk
    TajoBasics
    ToolWindows
    UserInputs
    Windows
```

```

TajoExtras
  TextDisplays
TextSWs
  BaseTextSWs
  TTYSWs
FormSWs
TajoTools
  Editor
BuiltInTools
  Executive
WiskSupport
  DontExportPilotRun
DevComSoft
  RealImpl
  FloppyImpl
MesaBasics
FileSystemex

```

CoPilot also allows you to see what the context was before going to the Debugger. Checking the context at this point, you find that the current module is **PilotNub** in the **MesaRuntime** configuration. This will always be the context after a **CALLDEBUG**:

```

>Current context
Module: PilotNub, G: 14544B, L: 4700B, PSB: 115B
Configuration: MesaRuntime

```

We are interested in the configuration **Lex**, so we make it our root configuration:

```
>SEt Root configuration: Lex*
```

and find out which modules are in this configuration:

```

>Display Configuration Lex
Lexicon, G: 70410B~
LexiconClient, G: 70434B

```

Notice the **~**, indicating that Lexicon hasn't been started yet. Now we can set the module context to be **Lexicon**, so that we can set some breakpoints:

```
>SEt Module context: Lexicon*
```

If you know which module is of interest, you need not search through the configurations to find it. A **SEt Module context** command works even if no root configuration is specified explicitly (this assumes that the module name is unique; if it isn't, an error message results). You could have responded to the first **>** prompt with a **SEt Module context** command if you knew that Lexicon was the module of interest.

III.6.3.2 Setting breakpoints

If the source text for **Lexicon** is loaded into a window, so you can set breakpoints by pointing at the text in two ways. First, you can display the stack and ask to see the source

(this loads and positions the source file for the current module into the source window of the Debugger):

```
>Display Stack
Lexicon, G: 70410B~ >s Cross jumped!
--Lexicon.mesa
>q
```

Second, you can load the file into a source window by selecting the file name **Lexicon** (the extension defaults to **.mesa**), moving into a source window (there is always at least one), and selecting the **Load** command from the menu. Note the message warning that Lexicon was compiled with the cross-jumping switch turned on.

To set a breakpoint on the exit of the procedure **NewNode**, scroll the window until this procedure is visible; then select the word **RETURN** inside it. Hold down the **MENU** key and choose the **Break** command. This sets a breakpoint on the exit of the procedure (selecting the word **PROCEDURE** or **PROC** sets a breakpoint on the entry to the procedure).

To set a breakpoint in the end of one of the **IF-THEN-ELSE** statements in the procedure **InsertString**, select any place in the statement **ELSE n.llink ← NewNode[];** and select **Break**. Where the breakpoint has been set is confirmed by the selection moving to the first character of the statement: **ELSE <>n.llink ← NewNode[];**. In all cases, the breakpoint is set to the beginning of the selected Mesa statement. You may also set entry and exit breakpoints in the program using keyboard commands. If, for instance, you wish to set a breakpoint on the entry to the procedure **FindString**, type:

```
>Break Entry procedure: FindString Breakpoint #3.
```

For any breakpoint, you may specify a condition that must be satisfied for the breakpoint to be taken. If, for example, a breakpoint is set on the statement **FOR i IN [0..n) DO** in the **LexicalCompare** procedure, you may attach the condition that **n** be greater than 10 for the breakpoint to be taken:

```
>ATtach Condition #: 4, condition: n > 10.
```

III.6.3.3 Proceeding

It is now time to proceed and run the program, but saving some comments along with the commands makes it easier to remember what happened when you review a log of the session. For instance, you might say:

```
>--this breakpoint was set to find a comparison of
<--lexemes longer than 10 characters>
```

Proceeding is now easy, as shown by the following command:

```
>Proceed [confirm]
```

If the lexeme "xxxxx" is subsequently added to the tree, one of the breakpoints is reached and CoPilot is reentered.

III.6.3.4 Examining and changing the state

The Debugger is next entered with the message:

```
Break #1 at exit from NewNode, L: 3760B, PC: 244B (in Lexicon, G:  
70410B)
```

to indicate from where and why CoPilot was entered. At this point you might display the stack and look at the variables:

```
>Display Stack  
NewNode, L: 3760B, PC: 244B (in Lexicon, G: 70410B) >v  
n = 4043126B ↑  
>q
```

or look at the several levels of the stack:

```
>Display Stack  
NewNode, L: 3760B, PC: 244B (in Lexicon, G: 70410B) >n  
InsertString, L: 3700B, PC: 137B (in Lexicon, G: 70410B) >n  
AddString, L: 3420B, PC: 115B (in Lexicon, G: 70410B) >n  
CommandProc, L: 6410B, PC: 506B (in LexiconClient, G: 70434B) >q
```

or ask to see what the node n (in NewNode) looks like (invoke the interpreter by typing a space):

```
>n ↑  
[llink: NIL, rlink: NIL, string: 4043120B ↑ (5,5)"xxxxx"]
```

It might be advantageous to set both the left link and right link of n to point to n itself and then check the value of n by typing:

```
>n.llink ← n; n.rlink ← n; n; n ↑
```

which responds with:

```
n = 4043126B ↑  
[llink: 4043126B ↑, rlink: 4043126B ↑, string: 4043120B ↑ (5,5)"xxxxx"]
```

If the value of the variable **root** in the module **Lexicon** is important, and the module context has been changed to **LexiconClient**, you may obtain the value using the **Find** command. **root** is a variable in the current configuration, but not the current module.

```
>Find variable: root NIL (in Lexicon, G: 70410B)
```

III.6.3.5 More breakpoint commands

To review all of the breakpoints, do the following:

```
>List Breaks
1 -- Break at exit from NewNode (in Lexicon, G: 70410B).
2 -- Break in InsertString (in Lexicon, G: 70410B).
Cross jumped!
ELSE <>n.llink ← NewNode[];
3 -- Break at entry to FindString (in Lexicon, G: 70410B).
4 -- Break in LexicalCompare (in Lexicon, G: 70410B). Condition: n >
10
Cross jumped!
<>FOR i IN [0..n) DO
```

If the breakpoints are no longer interesting, they may all be cleared simultaneously:

>Clear All Breaks

Individual breakpoints may be cleared either using the **Clear Break** command or by selecting the source code of the line containing the breakpoint and then selecting the **Clear** menu item from the Debugger menu.

III.6.3.6 Looking at the user screen

You may often be thrown into CoPilot without warning and without a chance to take stock of what was being displayed. The CoPilot **Userscreen** command provides for this situation. Entering the following command repaints the display with the contents of the client-world screen as it was before entering CoPilot:

>Userscreen [confirm]¶

In this mode, CoPilot accepts no commands and performs no client-world operations. After 20 seconds, the CoPilot display is restored automatically. To review the user screen for longer than 20 seconds, hold down the **ABORT** key, which maintains the display. Pressing **ABORT**, then releasing it, returns you to CoPilot.

III.6.3.7 Setting tracepoints

Suppose the user screen indicates that it is worthwhile to breakpoint the entry to the procedure **LexicalCompare**. When you set a breakpoint on entry to a procedure, you will often want to see the input parameters by typing:

>Trace Entry procedure: LexicalCompare Breakpoint #5.

If you **Proceed** and enter the lexeme *yyy*, the tracepoint will be reached. A message indicating why CoPilot was entered, the context, and a dump of the input parameters is then displayed:

```
Trace #5 at entry to LexicalCompare, L: 3760B, PC: 246B (in Lexicon,
G: 70410B)
>Display Stack
```

```
LexicalCompare, L: 3760B, PC: 246B (in Lexicon, G: 70410B) >P
s1 = 406412B↑(3,80)"yyy"
s2 = 4043120B↑(5,5)"xxxxx"
>q
```

This leaves CoPilot in the **Display Stack** command. You can terminate the command by typing **q** or continue to perform **Display Stack** functions.

III.6.4 Pilot symbols files

Symbolic access to Pilot structures is often essential in debugging Pilot client programs. In particular, such access is useful in interpreting Pilot **SIGNALS** and essential if you are to break entry or exit to a Pilot procedure.

The Pilot symbols files (found in **<Pilot>Symbols**) should satisfy most client debugging needs for access to Pilot structures. To determine which **Pilot .symbols** file pertains to the module in question, perform a **Current context** command, which displays the current configuration (you may wish to set module context or set octal context before this). The configuration name is prepended to the **.symbols** suffix to arrive at the symbol file name. The exceptions are listed in the table:

<u>Displayed Name</u>	<u>Symbols File</u>
HConfig	VMMgr.symbols
PConfig	VMMgr.symbols
Level0	Communications.symbols
Level1	Communications.symbols
Level2	Communications.symbols

III.6.5 Interpreting signals

If you go to CoPilot with an uncaught signal, you will often find a message of the form:

```
***uncaught signal[nnnnnB] msg = ?[mmB] (in module MumbleImpl, G:
pppppB)
```

This virtually useless message usually occurs because CoPilot did not have the necessary symbols files available to interpret the signal. To get useful information, find the file that contains the symbols for **MumbleImpl** and retrieve it. (It may also be necessary to retrieve the object file for an interface module so that signal parameters can be interpreted correctly.) Once the appropriate files have been fetched, type a space to invoke the CoPilot interpreter and then a **LOOPOLE** expression (the % is the loophole operator). This tells CoPilot to interpret the number **nnnnnB** as a **SIGNAL** from the current context. CoPilot will reply with a message similar to the one above, except it will have signal names instead of a number, and an ASCII message. For example, assume a simple module named **Test** has been loaded and started, and subsequently a word swap to CoPilot occurs:

```
13-Jan-82 15:01
*** uncaught SIGNAL [1005B] msg = ?[5423B] (in module Traps, G:
20624B)
```

The symbol file to be retrieved can be determined by finding the **Current context**:

```
>Current context
Module: Traps, G: 20624B, L: 11754B, PSB: 101B
Configuration: MesaRuntime
```

Once the context has been established to be MesaRuntime, retrieve the file **MesaRuntime.symbols** and re-interpret the signal.

```
>1005B%(SIGNAL)
SIGNAL DivideCheck (in module Traps, G: 20624B)
DivideCheckTrap, L: 11754B, PC: 1503B (in Traps, G: 20624B) >n
SDIV, L: 5240B, PC: 156B (in ProcessorHeadDLion, G: 21454B) >n
Test, L: 21214B, PC: 15B (in test, G: 11414B) >q
a = 0
b = 3410
>s Cross jumped!
a←0; <>b←5/a;
>q
```

It seems that there has been some sort of invalid division operation. To get more information, look at the call stack as illustrated above. It shows that **Test** tried to perform a divide-by-zero instruction, which ended in a signal being raised.

III.6.6 Address and write-protect faults

Pilot permits programs to access only those locations in virtual memory contained within mapped spaces. Furthermore, a space in virtual memory can be designated read-only (or equivalently, write-protected). Programs that try to write to such locations or that try to reference unmapped spaces will enter CoPilot with the message **WriteProtect Fault** or **Address Fault**, respectively. In addition, programs that attempt to reference a location beyond the end of the processor's virtual memory will enter CoPilot with the message **Address Fault (address past end of processor VM)**. These are not signals; Pilot has detected the fault and explicitly called the Debugger.

A write-protect fault is a fatal error, so neither Pilot nor the client program can be successfully restarted in this case. Conversely, address faults are not fatal errors, except to the process in which they occur. Pilot and the remaining client processes are still healthy and will continue to run if a **proceed** command is issued. The address-faulted process will be effectively blocked forever, waiting for pages to get swapped into real memory (which will never happen). As long as this process holds no vital monitor locks, everything should be fine. In addition, you may freely interpret procedures from CoPilot after an address fault. Since Pilot will be healthy, its facilities may also be used freely. Making address faults non-fatal allows you to clean things up after faulting but is not meant to provide a way to continue operation for an extended period of time. There is little or no experience with that kind of use, so its limitations and problems are largely unknown.

III.6.7 Tracing an address fault

When an Address Fault occurs, the Debugger is entered with the *Call Dbug* cursor, and displays the message **Address Fault**. No indication of which process caused the fault is given. Suppose that Lex had been running for a while and an address fault occurred. The first thing to do is list the set of processes and look for one that has page faulted (it will be clearly labeled).

Address Fault

```
>List Processes
PSB: 20B, page fault, address: 2515217B↑, L: 21304B, PC: 360B (in
StorageImpl, G: 32404B)
PSB: 75B*, ready, InitializeAwaitDebuggerRequest, L: 12144B, PC:
553B (in PilotNub, G: 14544B)
PSB: 77B, ready, L: 11374B, PC: 2364B (in UserInputsA, G: 26004B)
PSB: 100B, waiting CV, L: 4010B, PC: 3446B (in HeraldWindowsB, G:
35100B)
PSB: 101B, waiting CV, L: 3650B, PC: 377B (in TTYSWsB, G: 31500B)
PSB: 102B, waiting CV, L: 11410B, PC: 22316B (in TextSWsD, G:
32020B)
PSB: 103B, waiting CV, L: 12760B, PC: 6316B (in MFileImplA, G:
36214B)
PSB: 104B, waiting CV, L: 3440B, PC: 45464B (in UserInputsC, G:
30034B)
PSB: 105B, waiting CV, L: 37214B, PC: 14624B (in UserTerminalImpl,
G: 20010B)
PSB: 106B, waiting CV, L: 22370B, PC: 5325B (in UserInputsB, G:
26724B)
PSB: 107B, waiting CV, L: 3460B, PC: 2732B (in UserInputsA, G:
26004B)
PSB: 110B, waiting CV, L: 3470B, PC: 2667B (in UserInputsA, G:
26004B)
PSB: 111B, waiting CV, L: 3500B, PC: 2641B (in UserInputsA, G:
26004B)
PSB: 112B, waiting CV, L: 3520B, PC: 2641B (in UserInputsA, G:
26004B)
PSB: 113B, waiting CV, L: 11454B, PC: 344B (in SocketImpl, G:
23360B)
PSB: 114B, waiting CV, L: 21134B, PC: 1331B (in RoutingTableImpl, G:
23404B)
PSB: 115B, waiting CV, L: 37144B, PC: 2641B (in UserInputsA, G:
26004B)
PSB: 116B, waiting CV, L: 37360B, PC: 1232B (in EthernetDriver, G:
23060B)
PSB: 117B, waiting CV, L: 37234B, PC: 2271B (in EthernetDriver, G:
23060B)
PSB: 120B, waiting CV, L: 4044B, PC: 325B (in EthernetDriver, G:
23060B)
PSB: 121B, waiting CV, L: 21224B, PC: 306B (in DispatcherImpl, G:
23304B)
PSB: 122B, waiting CV, WriteFaultProcess, L: 11754B, PC: 65166B (in
SpaceImplB, G: 20464B)
PSB: 123B, waiting CV, L: 11424B, PC: 37B (in SwapperExceptionImpl,
G: 17570B)
```

```

PSB: 124B, waiting CV, L: 4550B, PC: 44B (in FilerExceptionImpl, G:
14104B)
PSB: 125B, waiting CV, L: 3430B, PC: 4460B (in MStoreImpl, G:
17270B)
PSB: 126B, waiting CV, L: 11644B, PC: 1053B (in CachedRegionImplA,
G: 15110B)
PSB: 127B, waiting CV, L: 3774B, PC: 37B (in PageFaultImpl, G:
17404B)
PSB: 130B, waiting CV, L: 21320B, PC: 7446B (in FileTaskImpl, G:
14200B)
PSB: 131B, waiting CV, L: 21334B, PC: 1463B (in DiskChannelImpl, G:
13220B)
PSB: 132B, waiting CV, L: 11550B, PC: 2031B (in PilotDiskImpl, G:
13324B)
PSB: 133B, waiting CV, FrameFaultProcess, L: 11530B, PC: 123B (in
FrameImpl, G: 14524B)

```

In this example, only one process has page faulted (20B), but if there are more than one, the **Octal Read** command will indicate which is the culprit. For each page-faulted process, an octal read should be performed on the associated address. CoPilot will respond with the message **!Invalid Address [nnnnnB]** for the process that is to blame for the address fault. The following verifies that process 20B is the culprit in the Lex example.

```

>Octal Read: 2515217B, n(10): 1
2515217B/ !Invalid Address [2515000B]

```

Once you have laid blame for the fault on a particular process, you may examine it more closely by setting the process context:

```

>SEt Process context: 20B

```

At this point you may look at the call stack using the **Display Stack** command, or at a particular frame using the **Display Frame** command. The latter command is very useful in many situations. For instance, suppose you have displayed and climbed the call stack:

```

>Display Stack
CopyString, L: 21304B, PC: 360B (in StorageImpl, G: 32404B) >n
NewNode, L: 3730B, PC: 333B (in Lexicon, G: 67410B) >n
InsertString, L: 3760B, PC: 240B (in Lexicon, G: 67410B) >n
AddString, L: 3410B, PC: 223B (in Lexicon, G: 67410B) >n
CommandProc, L: 12214B, PC: 746B (in LexiconClient, G: 70020B) >q

```

Suppose that sometime later you wish to look at variables or interpret statements in the context of **AddString**. Rather than climbing back through the stack using **Display Stack**, you may directly display that frame, as illustrated below:

```

>Display Frame: 3410B
AddString, L: 3410B, PC: 223B (in Lexicon, G: 67410B) >y
s = 412216B↑(5,80)"xxxxxx"
>q

```

Display Frame offers all of the functions available with **Display Stack** (including **n**). Hopefully there will be enough state attainable using CoPilot to track down the cause of the address fault.

III.7 Program-building tools

The *Binder* combines modules and previously bound configurations to produce a new configuration. The output of the Binder is a binary configuration description (object file) that may be loaded into a running system or later be input to the Binder.

CommandCentral is a tool that supports the compile/bind/run program development loop. It permits you to compile and bind programs on a development volume and run them on a client volume.

The *Compiler* translates Mesa source files into corresponding object files. An object file contains the executable code for the module, tables for use by the Binder and Loader, and symbols for use by the Debugger.

The *Formatter* transforms Mesa source files into a standard format. It establishes the horizontal and vertical spacing of the program text to reflect its logical structure.

MakeBoot transforms an object file containing Pilot and its client into a memory image that can be run on any machine conforming to the Mesa Processor Principles of Operation. The resulting boot file is later boot-loaded to get it started.

The *MakeDLionBootFloppyTool* creates Dandelion-bootable floppies.

The *Packager* explicitly groups procedures together into swap units.

III.8 Program analysis tools

The *Debugger* is CoPilot, the interactive Mesa debugger.

The *DebugHeap* Tool is used in CoPilot to debug the client, or in Tajo to do client-side debugging. It aids debugging by showing the layout of memory.

The *IncludeChecker* examines a collection of local or remote source and object files for consistency. It produces an output listing that gives a compile and bind order for the files and the dependencies among them. Inconsistencies are flagged. The IncludeChecker will also generate compile and bind commands to correct any inconsistencies.

The *Lister* produces listings of information in object files, such as dates of the definitions files used by an object file and cross-reference listings of procedure calls within the object file.

Performance Tools are five tools that aid in the study of the behavior of Mesa programs: the CountPackage, PerfPackage, Spy, Ben, and Willard.

Spy can measure the amount of time spent executing in a module, certain procedures, or even source statements within a procedure. It is especially useful for top-down

analysis of a program; thus, Spy can be used to first identify the hottest modules, then the hottest procedures within those modules, and so forth.

The CountPackage gathers information on the flow of control between groups of modules.

Willard produces a list of the control transfers executed during some interval of client activity.

The PerfPackage allows you to collect timing and frequency statistics of program execution.

Ben produces a list of the page faults that occur during some interval of client activity and tells what caused the fault to occur.

The *Statistics* tool gathers statistics about Mesa source and object files, such as number of characters and frame size.



Binder

This chapter discusses the operation of the Binder, including its switches and error messages. The Mesa Binder combines modules and previously bound configurations to produce a new configuration. The output of the binder is a binary configuration description (object file) that may be loaded into a running system or processed by a later invocation of the Binder. The configuration description language C/Mesa is used to describe desired configurations to the Binder. It is documented in the *Mesa Language Manual*.

To understand the Binder options described below, it is necessary to understand something about how configurations exist in files. The object file produced by the Binder contains a compiled description of the configuration; it may also contain copied code or symbols. For each module instance in the configuration, the object file specifies the location of the code and symbols by file name (and version stamp), starting page, and number of pages. Thus the code and symbols for a configuration may be scattered over a large number of files. The default is for the configuration's code to be copied to the object file, while its symbols are left in the original compiler object files. It is also possible to put the object file, the code, and the symbols in the same file (this is the way object files are generated by the Mesa compiler).

Copying the code or symbols for a configuration's modules is controlled by switches and parameters on the Binder's command line. Code is usually copied into the same file containing the object file. It is also possible to copy code into a file other than the object file, but this is not very useful. Symbols may be copied into the object file, but they are usually written to a separate file.

It is a good idea to package the symbols of a released subsystem into a separate file, so that they will not take up disk space when they are not in use. This also makes it easier to keep track of a consistent set of symbols for all of the modules. Because the Binder and Loader deal only with interfaces, symbol tables are not required for binding or loading. Of course, they are required for meaningful debugging.

17.1 Files

Retrieve `Binder.bcd` from the Release directory.

17.2 User interface

The Binder runs in the Executive and in Command Central. A summary of the Binder's commands is written on the file `Binder.log`. The error and warning messages from binding, say `Foo.config`, are found on `Foo.errlog` (unless the `/e` switch is in effect; see the Command line section below).

17.2.1 Command line

The Binder accepts a sequence of one or more commands, each of which usually has one of the following forms:

`inputFile/switches`

`outputFile ← inputFile/switches`

`[key1: file1, ... keym: filem] ← inputFile/switches`

In the third form the valid names for `key1` are `code`, `symbols`, and `bcd`. The string `inputFile` names the file containing the text of the configuration description, and its default extension is `.config`. There is a principal output file, the name of which is determined as follows:

If you use the first command form, it is `inputRoot.bcd`, where `inputRoot` is the string obtained by deleting any extension from `inputFile`.

If you use the second form, it is `outputFile`, with default extension `.bcd`.

If you use the third form and `key1` is `bcd`, it is `file1`, with default extension `.bcd`; otherwise, it is obtained as described for the first form.

If the Binder detects any errors, the principal output file is not written, and any existing file with the same name is deleted. You may also request that the code or symbols of the constituent modules be copied to an output file by specifying the `/c` switch or by using the third command form with keyword `code`. Code is copied to the principal output file unless you use the third form and `key1` is `code`, in which case the code is copied to a file named `file1`, with default extension `.code`.

You may request copying of symbols by specifying the `/s` or by using the third command form with keyword `symbols`. Symbols are copied to the file named as follows:

If you use the first command form, it is `inputRoot.symbols`.

If you use the second form, it is `outputFile`, with default extension `.symbols`.

If you use the third form and `key1` is `symbols`, it is `file1`, with default extension `.symbols`; otherwise, it is obtained as described for the first form.

Unless the `/e` switch is in effect, any warning or error messages are written on the file `outputRoot.errlog`, where `outputRoot` is the string obtained by deleting any

extension from the name of the principal output file. If there are no errors or warnings, any existing error log with the same name is deleted at the end of the bind.

When more than one Binder command is given on the command line, the commands are separated by semicolons. Usually the semicolon can be omitted. It cannot be omitted, however, if the second of the two successive commands is a global switch. For example:

```
>Binder /cs MySystem'; /c AnotherSystem
```

The semicolon can be left out between two successive identifiers (file names or switches), or between a | and an identifier. Any required semicolon in an Executive command must be quoted.

17.2.2 Switches

The optional switches are a sequence of zero or more letters. Each letter is interpreted as a separate switch designator, and each may optionally be preceded by - or ~ to invert the sense of the switch.

The Binder recognizes these switches:

- c copy code (default)
- e merge the .errlog file into the **Binder.log** file
- p pause if there are errors, or if there are warnings and the /w switch is specified
- s copy symbols
- w also pause on warnings if /p is specified (default)

Global switches are set by a command with an empty file name. Each of the switches listed above can be specified as a global switch. Note that unless a command to change the global switch settings comes first in the sequence of commands, it must be separated from the preceding command by an explicit semicolon (see Examples section).

The /p switch is unusual in that its meaning is slightly different, depending on whether it is a global or local switch. As a global switch, it means report (p) or don't report (-p) errors or warnings to the calling Executive. The Executive will typically terminate (pause) if errors or warnings are reported. The global default is to pause. As a local switch, it specifies pausing just after compiling the specified file if that file or any preceding file contained errors; moreover, any remaining commands are ignored. The local default is not to pause but to continue with the next input file.

17.2.3 Associating files with modules and configurations

The Binder lets you control the association between file names and the modules or configurations included in a configuration when you call it. This is done by specifying a list of component identifier-file name pairs inside brackets after the input file name. Such a list can be thought of as augmenting or replacing a **DIRECTORY** clause in the configuration description. For example, the command line

```
>Binder MySystem[Test: UnpackedTest]
```

will bind **MySystem.config** using the previously bound configuration **Test** that is stored on the file **UnpackedTest.bcd**.

A command that includes one of these optional component-file name lists will have one of the forms:

```
inputFile[id1: file1, ... idn: filen]/switches  
outputFile ← inputFile[id1: file1, ... idn: filen]/switches  
[key1: file1, ... keym: filem] ← inputFile[id1: file1, ... idn: filen]/switches
```

The module or configuration named by **id₁** in the configuration description will be read from the file **file₁**. The extension **.bcd** is assumed for the file names.

17.3 Examples

```
>Binder MySystem
```

Read **MySystem.config** and write the resulting object file on **MySystem.bcd**. Copy all code segments to **MySystem.bcd**. Symbol segments are not copied, but are left in the original input files. This is the normal mode because the loader will only load object files that have code copied into them.

```
>Binder MySystem/-c
```

Read **MySystem.config**; write **MySystem.bcd**. Leave all code and symbol segments as they were in the input files. This might be done if an intermediate level configuration were being bound, and code or symbols were going to be copied later when a higher-level configuration was bound.

```
>Binder MySystem/s
```

Read **MySystem.config** and write the resulting object file on **MySystem.bcd**. Copy all code segments into **MySystem.bcd**, and copy all symbol segments into **MySystem.symbols**. By packaging all of the symbols in a single file, you minimize the risk of getting an incorrect version of some symbol table. This is a possible distribution mode, if debugging will be required.

>Binder MySystem[SubSystem: ExperimentalSubSystem]

Read **MySystem.config**; write **MySystem.bcd**. Read the included subconfiguration **SubSystem** from the file **ExperimentalSubSystem.bcd**.

>Binder MySystem ← NewSystem.config/s

Read **NewSystem.config**; write **MySystem.bcd**. Copy all code segments into **MySystem.bcd** and all symbol segments into **MySystem.symbols**. Commands with "left-hand sides" allow renaming of the output (**bcd**, **symbol**, and **code**) files.

>Binder [bcd: MySystem.bcd, symbols: MySystem.bcd] ← NewSystem/c

Read **NewSystem.config**; write **MySystem.bcd**. Copy all code and all symbol segments into **MySystem.bcd**.

>Binder SubSys ← MySystem/cs

Read **SubSystem.config**; write **SubSys.bcd**. Then read **MySystem.config**; write **MySystem.bcd**; copy code into **MySystem.bcd** and symbols into **MySystem.symbols**.

>Binder /-c SubSystemA'; /c SubSystemB MySystem

Bind **SubSystemA**, **SubSystemB**, and **MySystem**, but only copy code for the last two configurations. Note that a semicolon is required before the second global switch.

17.4 Error messages

If possible, the Binder will indicate the offending source line and configuration name with each error. Some of the common error messages are:

Errors detected, Bcd not written

The Binder has produced no output.

Exported type clash

Only one implementation of an opaque type may appear in a configuration. This is true even if the interface defining the opaque type is "hidden" in a nested subconfiguration by not being exported by that subconfiguration.

Fatal Binder Error

Fatal errors are reported in a fashion similar to the Compiler; the signal and message are given in octal, and should be included in any change request reporting a fatal Binder error.

file could not be opened to copy symbols

Warning: When copying symbols, the file containing the symbol segments for a module could not be opened. The copied symbols file will still be produced, but will not

contain symbols for the module; thus limited debugging will still be possible using the symbols file.

file is referenced in two versions: (version1) and (version2)

Warning: Two different versions of the named file are referenced by the modules being bound. This will produce an error if you attempt to match the two versions as import and export.

id does not match the module or configuration name in the object file

The identifier used to name a module or configuration in a configuration description must exactly match (including capitalization) the name used inside that module or configuration.

id is not valid as a CONTROL

A control list item must be a module or subconfiguration in the configuration.

**item from interface is unbindable (imported by module)
(item nnn) from interface is unbindable (imported by module)**

Warning: An item from *interface* has no implementation. If symbols for the importer or the interface can be found, the item's name is printed. Otherwise, the item's interface number is printed, and you can count (from 0) the interface items in *interface* or use the Lister's **Interface** command to get more information.

**interface is not imported by any modules
interface is not exported by any modules**

A configuration must tell the truth about what it **IMPORTS** and **EXPORTS**; i.e., everything imported or exported by a configuration must actually be imported or exported by a contained module or configuration.

interface is undeclared

An attempt is being made to import the interface (or program) *interface*, but *interface* is neither imported from a higher-level configuration nor exported by any module or configuration at the same level.

interface1 (version1) is required for import, but only interface2 (version2) is available

interface2 is available for import (or being passed as a parameter), but the importer requires *interface1*. The source line shows the importer.

interface₁ (*version₁*) is being exported, but *interface₂* (*version₂*) is required

The source line shows an exporter of *interface₁* who is trying to assign the interface (implicitly or explicitly) to *interface₂*. This may be a version problem (if the interface names are the same) or an error in an assignment.

The right hand side exports more interfaces than required by the left hand side.

The left hand side requires more interfaces than exported by the right hand side.

An explicit list of interfaces or module instances was given as a result or argument list, and either too few or too many were given.

17.5 Current limitations

The **DIRECTORY** clause in a configuration description should be used *only* when the name of a module or configuration differs from the name of its file. Do not make **DIRECTORY** entries for interface (**DEFINITIONS**) files.

The output object file can be renamed; the symbols file cannot (since the object file contains the name of this file in its internal tables).

Multiple instantiations of nested configurations are not implemented. You can get around this by binding the nested configuration in a separate step.

CommandCentral

CommandCentral is a tool that supports the compile/bind/run program development loop. It permits you to compile and bind programs on a development volume and run them on a client volume. Because the functions provided by CommandCentral overlap with those of the Executive, also see the chapter on the Executive.

18.1 Files

CommandCentral is built into Tajo, so no files need be retrieved..

18.2 User interface

CommandCentral interacts through a message subwindow, a command subwindow, and a log subwindow.

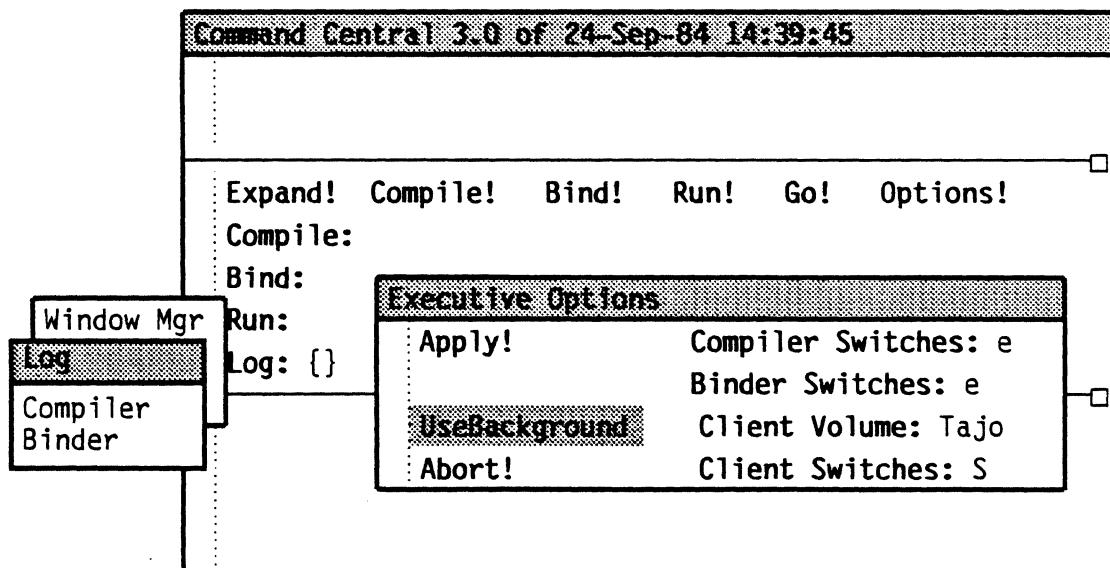


Figure 18.1: Command Central tool window

18.2.1 Message subwindow

The message subwindow is used to display error and status messages.

18.2.2 Command subwindow

The command subwindow contains the following fields and commands:

Expand! expands any file names listed containing #, *, or @ in the usual way (i.e., # matching one character, * matching zero or more, and @file@ expanded to the contents of file).

Compile! invokes the compiler, taking its arguments from the **Compile:** field.

Bind! invokes the binder, taking its arguments from the **Bind:** field.

Run! takes a list of file names with switches from the **Run:** field, transfers the corresponding files to the client volume, and (possibly) runs them.

Fine point: The commands **Compile!**, **Bind!**, and **Run!** each run in a separate process. This means that for example, invoking **Compile!** immediately followed by **Bind!** will run the compiler and binder simultaneously, which is probably not what is intended. The **Go!** command should be used to sequence through compilation, then binding, then execution.

Go! executes the **Compile!**, **Bind!**, and **Run!** commands, in that order. If a command fails, the subsequent commands are not executed.

Fine point: The command line to a subsystem is copied when the subsystem starts. The contents of the command lines can be changed until the corresponding system starts running, e.g., the Binder line can be edited while the compiler is running.

Options! allows switches to be specified for the **Compile!**, **Bind!**, and **Run!** commands (see the chapters on the Binder and Compiler). The client volume may also be specified in the Options window. Each of these items override those taken from the User.cm or the default if no User.cm exists. The Boolean item **UseBackground**, if set to **TRUE**, runs the Compiler and Binder at background priority.

Compile: contains a list of file names and optional compiler switches. The file names and switches are passed directly to the compiler as if they had come from the command line of the Executive.

Bind: contains a list of the file names and optional switches that are passed as input to the binder.

Run: is the input field used to list the files to be run on the client volume. The following switches are recognized by the **Run!** command:

- a Start with active initialToolStateDefault rather than inactive.
Default **FALSE**.
- c Copy from development volume, default **TRUE**
- e Executable (i.e., load the object file), default **TRUE**

- s** Start after loading, default **TRUE**
- l** Load with code links, default **FALSE**
- d** Debug: call debugger after loading, default **FALSE**

The default is to copy, load, and start each file named. (The default extension is **.bcd**; files without extensions may not be used.) To copy but not load a file, use **/-e** (i.e., don't execute). To run a file already on the client volume, use **/-c** (i.e., don't copy).

Log{ allows you to explicitly load the desired **.log** file into the bottom message subwindow. The **.log** file is selected by depressing the menu button over the tag and selecting either **compiler** or **binder**.

18.2.3 Log subwindow

After completion of a Compile or Bind, the bottom subwindow is loaded with the corresponding **.log** file. Any time **Compiler.log** or **Binder.log** is changed (e.g., if you edit one of them and save it), it will be loaded into the window. Also, if the current search path changes to one not containing **Compiler.log** or **Binder.log**, the log subwindow will automatically be cleared if it contains one of the log files.

18.3 Communication between client and development volumes

When the **Run!** command is invoked, CommandCentral creates a file in the root directory of the client volume that consists of a list of the file names (converted to file ids), and switches that were on the **Run** line. When the client volume is booted, a check is made in its root directory, and if CommandCentral's run file is found, the listed object files will be executed. Once CommandCentral's run file has been read, the client volume destroys it, so that subsequent booting of the client volume will not cause a re-run of the same programs.

Since the run file created by CommandCentral is not a development environment file, it cannot be accessed, deleted, or read from the development environment, but instead is fully maintained by the client volume and CommandCentral. If for some reason a boot initiated from CommandCentral were aborted or interrupted, the client volume may be in an inconsistent state in relation to the existence of CommandCentral's run file. The next time the client volume is booted, it may or may not produce the desired results, depending on whether the file actually got created. For example, if the file were created before the interrupt, and the client volume is subsequently booted from the HeraldWindow menu, an attempt will be made to execute the object files in the run file most recently created by CommandCentral. This is not what one expects when booting from the HeraldWindow. If the client volume is rebooted from CommandCentral, a check will be made to see if the file already exists. Since it does in this example, no attempt will be made to create a new one, so the old one will be used. If the list of files in the **Run** line did not change and at least one file in the list was re-compiled, the results will be particularly confusing since the file id recorded in the previous run file on the client volume will not match the id for the latest object file on the development volume.

18.4 User.cm

The `User.cm` fields used by CommandCentral are:

[Executive]

<code>Compiler:</code>	<code>NameOfCompiler</code> (default extension is <code>.bcd</code>); default is <code>Compiler.bcd</code> .
<code>Binder:</code>	<code>NameOfBinder</code> (default extension is <code>.bcd</code>); default is <code>Binder.bcd</code> .
<code>CompilerSwitches:</code>	default global switches for compiler.
<code>BinderSwitches:</code>	default global switches for binder.
<code>ClientVolume:</code>	<code>VolumeLabelString</code> ; default is first volume of type below CommandCentral's system volume.
<code>ClientSwitches:</code>	Pilot switches used for booting client volume.
<code>CodeLinks:</code>	<code>TRUE FALSE</code> for compiler/binder loading; default <code>TRUE</code> .
<code>UseBackground:</code>	<code>TRUE FALSE</code> if <code>TRUE</code> , the compiler and binder run at background priority. Otherwise, they run at normal priority. Default <code>FALSE</code> .

The name of the development volume is set in the client volume `User.cm`:

[System]

`CommandVolumeName:` `VolumeName`

If no development volume is specified, the volume is defaulted to `CoCoPilot` if the client volume is of type debugger, and to `CoPilot` otherwise.

CommandCentral's window size, tiny place, and initial state can be set as for any other tool:

[CommandCentral]

`WindowBox:`

`TinyPlace:`

`InitialState:`



Compiler

The Mesa compiler translates Mesa source files into corresponding object files. An object file contains the executable code for the module (if any), a binary configuration description (for use by the binder or loader), and a symbol table (for inclusion by other programs or for use by the debugger). By convention, an object file has a name with extension `.bcd`.

The *Mesa Language Manual* describes the syntax and semantics of the Mesa source language. This chapter describes the operation of the Compiler, including the compile-time options and messages.

19.1 Files

Retrieve `Compiler.bcd` from the Release directory.

19.2 User interface

The Compiler runs in the Executive and takes commands from the command line. The simplest form of command is a list of file names, such as

```
>Compiler sourcefile1 sourcefile2 ... sourcefilen
```

If you supply the command `sourcefile` with no period and no extension, the Compiler assumes you mean `sourcefile.mesa`.

During compilation, the Compiler gives feedback by giving the name of the file, any non-default switches, and a dot at the beginning of each major pass (six dots in all). It also shows code size if successful, or number of errors/warnings if not.

The Compiler reports the result of each command on the file `Compiler.log` with a message having one of the following forms (each * is replaced by an appropriate number; bracketed items appear only when relevant):

`Command:/switches`

```
Command: file
file.mesa
[lines: *, code: *, links: *, frame: *, time: *]
```

Compilation was successful. The object file is `file.bcd`. For a **DEFINITIONS** module, the code and links are not meaningful and are omitted. Otherwise, "links" is the number of items imported by the module, and "frame size" is the size of the global frame (in words), exclusive of the links. A third line appears only if warning messages were logged. The Compiler issues warnings for certain constructs that are technically correct but nonsensical or likely to be unintended. Warnings do not prevent writing a valid object file, but you should usually investigate them.

```
file.mesa -- aborted, * errors [and * warnings] on file.errlog
```

Compilation was unsuccessful. You will find the error messages (and warning messages, if any) in the indicated file. If the errors were detected during the early phases of compilation, no object file was written (and any existing object file with the same name was deleted).

File error

The Compiler could not find the specified file.

Fine point: **ABORT** will cause the Compiler to return at the end of the current pass, ignoring any other files to compile.

19.2.1 Command line

The Compiler allows you to control the association between modules and file names at the time you invoke the Compiler. The Compiler accepts a series of commands, each of which has the form

```
outputFile ← inputFile[id1: file1, ..., idn: filen]/switches
```

Only `inputFile` is mandatory; it names the file containing the source text of the module to be compiled, and its default extension is `.mesa`. Any warning or error messages are written on the file `outputRoot.errlog`, where `outputRoot` is the string obtained by deleting any extension from `outputFile`, if given, otherwise from `inputFile`. If there are no errors or warnings, any existing error log with the same name is deleted at the end of the compilation.

If a list of keyword arguments appears between brackets, each item establishes a correspondence between the name `idi` of an included module, as it appears in the **DIRECTORY** of the source program, and a file with name `filei`; the default extension for such file names is `.bcd`. (If the name of an included module is not mentioned on the command line, its file name is computed from information in the **DIRECTORY** statement).

The optional switches are a sequence of zero or more letters. Each letter is interpreted as a separate switch designator, and each may optionally be preceded by `-` or `~` to invert its sense.

If *outputFile* and *←* are omitted, the object code and symbol tables are written on the file *inputRoot.bcd*, where *inputRoot* is *inputFile* with any extension deleted. Otherwise code and symbols are written on *outputFile*, for which a default extension of *.bcd* is supplied. If the Compiler detects any errors, the output file is not written and any existing file with the same name is deleted.

The Compiler accepts a sequence of one or more commands from the Executive's command line. Commands are separated by semicolons, but you may omit a semicolon between any two successive identifiers (file names or switches), or between a *|* and an identifier (but not between an identifier and a */*). Note that any required semicolon in an Executive command must be preceded by a single quote ('').

You can set global switches by a command with an empty file name. In the form */switches*, each letter designates a different switch. Unless a command to change the global switch settings comes first in the sequence of commands, you must separate it from the preceding command by an explicit semicolon.

19.2.1.1 Examples

```
>Compiler ReadOldFormat ← ReadData[DataFormat: OldFormat]
```

Compile the program **ReadData.mesa** that has the included interface **DataFormat** in its **DIRECTORY** statement. Use the file **OldFormat.bcd** (which contains the declaration **DataFormat: DEFINITIONS = ...**) as the source of this interface. Put the object program in the file **ReadOldFormat.bcd**.

```
>Compiler/~j SymStuff[Table: LongTable]/n SymExtra[Table: LongTable]
```

Compile the files **SymStuff.mesa** and **SymExtra.mesa**, getting the definition of Table from **LongTable.bcd**. Produce object files **SymStuff.bcd** and **SymExtra.bcd**. Don't cross-jump either module and generate **NIL** checks for **SymStuff** only (switches explained below).

19.2.2 Switches

Switches allow you to modify command input. A command has the general form

file[/s]

where [] indicates an optional part and **s** is a sequence of switch specifications. A switch specification is a letter, identifying the switch, optionally preceded by a '**'**' or '**~'** to reverse the sense of that switch. The valid switches are

- b** bounds checking
- e** .errlog file is merged into **Compiler.log**
- j** cross-jumping optimization (default)
- n** NIL pointer checking
- p** pause after compiling file if there are errors or warnings
- s** sort global variables and entry indices (default)
- u** uninitialized variable checking

- w report warning messages (default)
- y warning on runtime calls

Each switch has a default setting. The command **sourcefile** is equivalent to **sourcefile/~b~ej~n~ps~uw~y** if you use the standard defaults (i.e., if the Compiler cross-jumps the code, does not pause after compiling file, sorts variables, and logs warning messages). It does not do bounds, **NIL** pointer, or uninitialized variable checking, and does not warn about runtime calls.

You can change the default setting of the switches by having an entry

compilerSwitches: <your defaults>

in the [**Executive**] section of the file **User.cm**

You can also change the default setting of any switch by using a global switch. Any switch given with no file name (i.e., just a slash and switches) establishes the default setting for that switch. Unless overridden or reset, that default applies to all subsequent commands.

Fine point: Any global switches given at other than the beginning of the command line must be preceded by a semicolon (quoted to the Executive), or the command parser will assume that they are local switches on the previous file. The command parser only allows a single slash after a given file, so some cases of missing semicolon are flagged.

Here is some information about the options:

b[ounds]

If bounds checking is specified, the Compiler inserts code to check that values are within range for all assignments to subrange variables and all indexing operations. Checking is also inserted for all assignments of signed values to unsigned variables and vice versa. If the value is out of range, the signal *BoundsFault* is raised (see the *Pilot Programmer's Manual*). The Compiler performs some bounds checking during compilation and does so independently of the setting of the /b switch. If it can deduce that no bounds failure is possible, the runtime check is omitted; if a bounds failure is unavoidable, it reports the error during compilation. Compile-time bounds checking assumes that all variables are initialized before use.

Fine point: Bounds checking in indexing operations is suppressed if the declared index type is empty, e.g., [0..0].

error to log

Errors are appended to **Compiler.log** rather than onto a separate **file.errlog**.

j[umped]

Cross-jumping is a peephole optimization technique that potentially shortens the object code. The reduction in code size ranges from negligible to 20%, depending upon coding style. If cross-jumping is specified, the correspondence of source to object is no longer one-to-one. This affects the debugger's ability to set breakpoints and identify code locations (see the Debugger chapter.) However, you can still set entry and exit breaks on all procedures. This switch also enables tail recursion elimination. If the

last operation in a procedure is a call of itself, the call can often be turned into a jump and the old frame reused.

n[i1]

If **NIL** checking is specified, the Compiler inserts code to check for a null value prior to any operation that dereferences a pointer. Note that indexing operations using an array descriptor or a string also imply dereferencing and are checked. If the pointer value is **NIL**, the signal *PointerFault* from interface *Runtime* is raised. No compile-time checks for **NIL** are performed.

Fine point: No **NIL** checks are provided in the dereferencing of relative pointers.

Depending upon coding style, these runtime checks can increase the size of the compiled code substantially. The first page of the address space is typically unmapped, so most dereferences of **NIL** generate an *Address Fault*.

p[ause]

This switch is unusual because its meaning is slightly different, depending on whether it is a global or local switch. As a global switch, it means to report (**p**) or not report (**-p**) errors or warnings to the calling Executive. The Executive will typically terminate (pause) if errors or warnings are reported. The global default is to pause. As a local switch, it specifies pausing just after compiling the specified file if that file or *any preceding file* contained errors; moreover, any remaining commands are ignored. The local default is not to pause but to continue with the next input file.

s[ort]

Normally, the Compiler sorts certain items by frequency of use before assigning addresses. This helps to keep the object code compact. If sorting is suppressed (**-s**), the assignments of global frame offsets and entry indices depend only upon order of declaration in the source text. This switch was added in anticipation of tools allowing inexpensive correction and replacement of modules in a configuration. These tools are not yet available.

u[ninitialized variables]

If the **/u** switch is given, the Compiler issues warning messages for uses of apparently uninitialized variables (but not fields of records). The algorithm used to detect suspicious usage is based upon the following assumptions:

- The entire body of a procedure is executed before the bodies of any procedures declared within it.
- Within any procedure, the order of execution is equivalent to the order of appearance of source text (for the purposes of variable initialization).
- The bodies of the contained procedures are executed in order of appearance.

The algorithm works fairly well for detecting certain common errors, but it is obviously not foolproof. *There is no guarantee that all uses of potentially uninitialized variables are reported;* conversely, properly initialized variables are sometimes

flagged when the initialization depends upon the order of execution of subprocedures. (Performance with respect to global variables is improved by putting the initialization code for a module either in the main body or the lexically first procedure.)

w[arnings]

Report (**w**) or don't report (-**w**) certain legal but suspicious constructs that can be detected by the Compiler. Warnings are written to the error log, but are not reported to the calling Executive.

y[ell about runtime calls]

This switch is intended for use by programmers writing such things as bootstrap loaders where the standard Mesa runtime machinery is unavailable. It flags operations, such as certain division, that generate calls to system functions.

19.3 Examples

>Compiler foo

Compile foo using all the default switch settings.

>Compiler foo/-w-j

As above, but suppress warning messages and do not cross-jump.

>Compiler /-p file1 file2 file3

Use this form if you want the Compiler to press on no matter what. If it is part of a command file, the next (Executive) command will be executed whether or not there were errors.

>Compiler file1 file2/p file3

Use this form if you want the Compiler to pause before compiling **file3** if either **file1** or **file2** does not compile successfully. If **file3** depends upon the others (by including them), this can save a lot of wasted time and effort.

>Compiler file1/p ' ; /-p file2 file3

Use this form if you want the Compiler to pause before compiling **file2** if **file1** does not compile successfully. Press on to the next Executive command even if **file2** or **file3** does not compile.

19.4 Error messages

The Compiler writes error and warning messages for **sourcefile.mesa** on either **sourcefile.errlog** or **Compiler.log**, depending on the setting of the /e switch. Each pass detects certain classes of errors. Error messages are logged in (approximate) source order by each pass. Within a single pass, the Compiler does its best to complete its analysis in spite of any errors. Detection of an error by one pass causes all following passes to be skipped. Thus you will sometimes get a new set of error messages after correcting all those

reported by a previous run of the Compiler. The Compiler never writes a bindable or loadable object file if it detects any errors.

The Compiler also logs warning messages. These are advisory only and are intended to draw your attention to suspicious usage. They do not abort compilation or invalidate the object file (but they should be checked).

Here is a trivial and nonsensical program that illustrates the form of the Compiler's error messages.

```
Sample: PROGRAM =
BEGIN
i: INTEGER,
i ← j + TRUE;
END.

i: INTEGER,
      ↑ Syntax Error [46]
Text deleted is: ,
Text inserted is: ;

j is undeclared, at Sample[52]:
i ← j+TRUE;

TRUE has incorrect type, at Sample[52]:
i ← j+TRUE;
```

The first message is generated by the first pass and shows how syntactic and lexical errors are reported. The arrow points to the first symbol that is necessarily invalid (or one symbol before it), and the decimal number is a character index in the source file. Of course, the Compiler cannot know what you intended, and the "real" error might have occurred quite a bit earlier. The Compiler tries to fix these errors as best it can by local deletion and insertion of symbols. These symbols are not written into the source file but are reported to help you interpret subsequent messages. If the Compiler cannot find a way to continue parsing, or if too many of these errors accumulate, it gives up.

Fine point: In order for the arrow to line up under the syntax error, you need to be viewing the file with a fixed-pitch font.

Fine point: If you are viewing the program and its error log in separate windows, you can use the **Position** command on one of the menus of the source window to locate the errors, given the character indices in the error log.

The other error messages report semantic errors. Errors are located by displaying a line of source text (the second line in each message) as well as the character index (a decimal number) and the enclosing procedure or program name (the identifier preceding the number). The text of the error message is intended to be reasonably self-explanatory. Sometimes it refers to an identifier or expression. The Compiler reconstructs these expressions from the parse tree; in later passes, the reconstruction often reflects rearrangement or constant folding so it may not exactly duplicate the source code. As subexpressions, ? indicates an undeclared identifier and indicates either a cutoff

because of depth of nesting or an expression form the Compiler cannot reconstruct from the parse tree.

19.5 Compiler failures

The message reporting a Compiler failure has the following form:

```
FATAL COMPILER ERROR, at id[index]:  
(source text)  
Pass = n, signal = s, message = m
```

Such a message indicates that the Compiler has noticed some internal inconsistency. The Compiler will skip the remainder of the command line if this happens.

19.6 Current limitations

The following limits are built into the current implementation of Mesa and are enforced by the Compiler:

The number of interface items declared in a single **DEFINITIONS** module cannot exceed 128.

Neither the number of procedure bodies nor the number of signal codes defined in a single **PROGRAM** module can exceed 128.

The size of the frame or record required by a procedure or program cannot exceed 4096 words.

Procedure declarations cannot be nested more than five levels deep, counting catch phrases as procedure levels.

The Compiler allocates its internal tables dynamically and tries to adjust their relative sizes to accommodate the program being compiled. When it is unsuccessful, it reports failure with a message of the form:

Storage Overflow in Pass n

Usually, the best thing to do is split your program into two or more smaller modules. If the Pass is 5, you can often get your program compiled by breaking the largest procedure into two or more smaller ones. This is because Pass 5 generates code for the module one procedure at a time, and needs enough table space to hold the code representation of the largest procedure.



Formatter

The Formatter transforms Mesa source files into a standard format. It establishes the horizontal and vertical spacing of the program in a way that reflects its logical structure. Since the Formatter uses the scanner and parser of the compiler to determine structure, only syntactically correct programs may be formatted.

This chapter describes the formatting rules and the operation of the Formatter, including the runtime options and messages.

20.1 Files

Retrieve **Formatter.bcd** from the Release directory.

20.2 User interface

The Formatter runs in the Executive and accepts the same command syntax as the Compiler. The simplest form of command is just the name of a source file to be formatted. If you supply the command **sourcefile** with no period and no extension, the Formatter assumes you mean **sourcefile.mesa**.

The Formatter reports the result of each command in **Formatter.log** with a message having one of the following forms (each * is replaced by an appropriate number; bracketed items appear only when relevant):

file.mesa -- lines: *, time: *

Formatting was successful. The source file has been rewritten.

file.mesa -- aborted, * errors [and * warnings] on file.errlog

Formatting was unsuccessful. The output of the Formatter is undefined if syntax errors exist in the input file. The original file is undisturbed.

File error

The Formatter could not find the specified file.

20.2.1 Command line

The Formatter takes commands of the form

```
[output1 ←] input1[/s] . . . [outputn ←] inputn[/s]
```

where [] indicates an optional part and s is a sequence of switch specifications. Only **inputFile** is mandatory; it names the file containing the source text of the module to be formatted, and its default extension is **.mesa**. Any warning or error messages are written on the file **outputRoot.errlog**, where **outputRoot** is the string obtained by deleting any extension from **outputFile**, if given, otherwise from **inputFile**. If there are no errors or warnings, any existing error log with the same name is deleted at the end of the formatting.

20.2.2 Switches

Switches allow you to modify command input. A switch specification is a letter, identifying the switch, optionally preceded by a '-' or '~' to reverse its sense. The syntax is the same as for the Compiler (chapter 14). The valid switches are:

- e** append errors to **Formatter.log** rather than onto a separate **file.errlog**
- g** don't close print file at end of input file
- h** generate a print file (does *not* force ~t)
- k** generate a two-column landscape print file (does *not* force ~t)
- o** take specified string and include it in the header of the print output of all following files
- p** pause after formatting if there are errors
- t** overwrite input file with plain text formatted version (default)

Each switch has a default setting. The command **sourcefile** is equivalent to **sourcefile ← sourcefile/~e~g~h~i~k~p~rt~v~z** if you use the standard defaults; i.e., the Formatter only generates a plain text file to replace the original source.

You can redefine the default settings by having an entry

```
compilerSwitches: <your defaults>
```

in the **[Executive]** section of the file **User.cm** (**compilerSwitches** because the switch processing code is shared with the compiler).

You can change the default setting of any switch by using a global switch. Switches given with no source file are global. Unless overridden or reset, that default applies to all subsequent commands. (See the multiple program print output example below.)

Some additional information about the options:

- g If a print file is being generated, it is not closed at the end of the current input file. It is expected that another file in the command list will also be generating print file output and a single print file will contain multiple input files. The name of the print file will be that of the first to which print output is being generated. If the type of print file (landscape vs. portrait or print vs. interpress) changes, the first will be closed and another print file will be started. Be careful not to generate a print file larger than will be accepted by your printer.
- p The p (pause) switch has semantics identical to that of the Compiler's p switch.

20.3 Formatting rules

As a general rule, the Formatter changes only the white space in the program. It does not insert or delete any printing characters. On the other hand, it may insert white space where there previously was none.

20.3.1 Spacing

Indentation is done by a combination of tabs and spaces in plain-text mode (assuming that a tab equals eight spaces).

The decision as to where to break lines is made independently of the output mode (print file or plain text).

A logical unit will be placed on a single line if it fits.

A simple carriage return in the input file is treated as a space. The occurrence of consecutive carriage returns (up to six blank lines) are preserved in the output file. Page breaks indicated by **CTRL L**'s in source programs are also preserved. Since all Bravo looks are discarded by the scanner, paragraph leading done with looks is not preserved.

For output files that contain fonts and faces, these additional rules apply:

- Comments are set in italics.
- The names of **PROCEDURES**, **SIGNALS**, **ERRORS**, and **PORTS** (but not user-defined transfer types) are bold where they are defined.
- Reserved words and predeclared identifiers are in a smaller font than other symbols. For portrait listings, Helvetica 10 and 8 are used; for landscape listings, Helvetica 8 and 6 are used.

In general there are no spaces before or after atoms containing only special characters. Exceptions to this rule are as follows:

- A space or carriage return follows (but does not precede) a comma, semicolon, or colon.
- A space precedes a left square bracket when the bracket follows any of the keywords **RECORD**, **MACHINE CODE**, **PROCEDURE**, **RETURNS**, **SIGNAL**, **PORT**, and **PROGRAM**.

- Spaces surround the left-arrow operator.
- The exclamation point (enabling) and equal-greater (chooses) operators are always surrounded by spaces. This is also true for equal signs used in initialization and for asterisks used in place of variant record tags.
- Some arithmetic operators, depending on their precedence, are surrounded by spaces.

20.3.2 Structure

The Formatter determines the indenting structure of the program by the brackets that surround the bodies of compounds. The brackets include {}, (), [], BEGIN-END, DO-ENDLOOP, and FROM-ENDCASE. An attempt is made to maximize the amount of information on a page. For example, consider:

```
Record: TYPE = RECORD [
    field: Type,
    field: Type];

```

```
Record: TYPE = RECORD
[
    field: Type,
    field: Type,
];

```

In both cases, the structure is clear; it is indicated by the indenting, not the placement of the brackets. The Formatter generates the form on the left.

The body of each compound, assuming it does not fit on a single line, is indented one nesting level. The placement of the brackets depends on the bracket and on its prefix and its suffix. For example, a loop statement has the following possible prefixes, brackets, and suffixes:

<u>Prefixes</u>	<u>Brackets</u>	<u>Suffixes</u>
FOR, WHILE	DO	OPEN
UNTIL, (empty)	ENDLOOP	ENABLE

The following paragraphs contain a number of examples. They observe the following rules for the placement of opening and closing brackets:

The opening brackets {}, [, FROM, and DO appear on the same line as their prefixes; BEGIN starts on a new line.

If the remainder of the statement fits on a single line (with its closing bracket), it is placed there, indented one level. Otherwise, all closing brackets except] and } appear on lines by themselves. If } is preceded by a semicolon, then it is also placed on a line by itself.

The statement following a **THEN** or **ELSE** is indented one level, unless it fits on the same line. **THEN** is on the same line as its matching **IF** and **ELSE** is indented the same amount as **IF**.

```
IF bool THEN
  BEGIN
    body
  END
ELSE
  BEGIN
    body
  END
```

```
IF bool THEN statement
ELSE {body}

IF bool THEN {
  statement;
  statement}
```

The labels of a **SELECT** (and its terminating **ENDCASE**) are indented one level, and the statements a second level, unless they fit on the same line with the label.

```
SELECT tag FROM
  case = > statement;
  case = >
  long statement;
ENDCASE
```

Each compound **BEGIN-END**, **DO-ENDLOOP**, or bracket pair is indented one level. When the rules for **IF** and **SELECT** call for indenting a statement, a **BEGIN** is not indented an extra level.

These rules are not exhaustive, but are intended to give the flavor of the Formatter output.

20.4 User.cm

Entries currently implemented are

[Formatter]

CharsPerLevel: n Specifies the number of spaces to be used for each indenting level. Default value is 2

CharsPerLine: n Specifies the number of characters to be used for line breaking. Default value is 82

20.5 Examples

>Formatter foo

Format **foo** using all the default switch settings (standard or established by a global switch).

>Formatter foo/-tk

Formats **foo** into a two-column landscape print file, leaving the original source unchanged.

```
>Formatter /-tkg ProgA ProgB ProgC ProgD
```

Produces a two-column landscape print file **ProgA.interpress** that contains listing of all four programs, each starting on a new page.

```
>Formatter /g~tk "Trinity Release"/o *Defs.mesa
```

Produces a two-column landscape print file that contains listing of all files *.mesa with the heading "Trinity Release".

20.6 Formatter failures

The message reporting a Formatter failure has the following form:

```
FATAL COMPILER ERROR, at id[index]:  
(source text)  
Pass = 1, signal = s, message = m
```

Such a message indicates that the Formatter has noticed some internal inconsistency (the above message is not a typo; the message comes from a module shared with the compiler). The Formatter will skip the remainder of the command line if this happens.

Note: The Formatter uses routines exported by **Print.bcd** to produce print files. If the proper package is not already loaded, the Formatter attempts to load it; if this fails, it complains about the lack of available print software. The file **Fonts.widths** must also be present on the local disk.



MakeBoot

MakeBoot is a program that constructs a boot file suitable for installation on a Pilot logical volume. A boot file is essentially a "virtual execution environment": it consists of a memory image containing a number of object files that have been loaded but not started. The memory image built by MakeBoot is loaded into memory by a simple loader called the *germ*, which transfers control to Pilot initialization code.

The simple view of MakeBoot is that it takes a collection of object files, constructs a memory image, and writes it out as a boot file. In practice, however, MakeBoot requires more information than just the names of the object files; this information is contained in a text file (or files) called the *parameter file*. The parameter file contains two types of information. The first type of information describes sizes of data structures such as the length of the global frame table or the number of processes. The second type of information describes what portions of memory must be resident or initially resident, since they are needed before Pilot's swapping machinery has been set up.

While the loader in MakeBoot is essentially the same as the runtime loader in Pilot, there are some differences. Modules that were bound with code links are always loaded with code links by MakeBoot; you cannot override the link type that was given to the Binder. This has important ramifications. If a configuration in the boot file imports an item that will be supplied at runtime by a dynamically loaded module or configuration, that configuration in the boot file must be bound with **LINKS: FRAME** (which is the default). If this rule is violated, then a dynamically loaded module or configuration will leave dangling pointers in the boot file; thus on a subsequent boot, attempting to (say) call a procedure in such a module when it has not yet been loaded in the new session would cause transfer of control into garbage, leading to unpredictable behavior.

21.1 Files

Retrieve **MakeBoot.bcd** from the Release directory. It requires one or more parameter files that specify various data structure sizes and initial memory configurations.

21.2 User interface

MakeBoot runs in the Executive.

21.2.1 Commands

Commands are of the form:

> **MakeBoot** *command command ... command6*

where each command specifies the creation of one boot file. The commands have the form:

outputfilename ← inputfilename[arguments]/switches

where the *outputfilename* and "←" are optional, and the arguments are a list of "key: arg" pairs separated by commas. *inputfilename* is a bound configuration. Output is written to *rootName.boot* and *rootName.loadmap*, where *rootName* is obtained by removing any extension from either the output file name (if one is given) or the input file name.

The possible arguments are given below. If no key is given for an argument, "parm" is assumed.

parm: parameterFile

ParameterFile names a file that supplies MakeBoot with information about the initial memory configuration and sizes of various data structures. If no extension is given, *.bootmesa* is assumed. These parameter files are released with Pilot. With the exception of the **GFT** and **PROCESSES** entries, ordinary clients will not change any entries in the parameter file. The various parameters are described in the section below. Several parameter files may be specified; the effect is to concatenate them.

nProcesses: number

(Optional) sets the number of processes that can exist. This guarantees that enough space is set aside for *number* processes, but since the table is rounded up to a page boundary, it may be possible to have more than the specified number. A default is normally given in the parameter file.

gftLength: number

(Optional) sets the length of the global frame table. This determines the maximum number of module instances that can exist. The maximum length is 1024.

Note: A module requires one entry in the table for each group of 32 procedures or signals. Thus a module with 60 procedures requires two entries. A default is normally given in the parameter file.

bcd: file

(Optional) names an additional object file to load.

switches: string

(Optional) sets the default boot switches in the boot file.

21.2.2 Switches

MakeBoot's switches are:

- /g Germ: build a germ rather than a boot file.
- /h Hex: print numbers in hexadecimal in the loadmap. The default is octal.
- /d Prints debugging information in the loadmap.

21.2.3 Parameter files

Some parameters require entries that are not numbers. The syntax for these non-numeric entries is given here.

```

list          ::= listItem | list listItem
listItem      ::= module | configPart | CODEPACK [nameList] | FRAMEPACK [nameList]
                  | GLOBALFRAME [configPartList] | CODE [configPartList] | BCD[nameList]
configPartList ::= configPart | configPartList , configPart
configPart    ::= ALL | module | configName [ moduleList ]
moduleList    ::= module | moduleList , module
module        ::= name | name . instance
nameList      ::= ALL | moduleList

```

The specifications **CODE [configPartList]**, **configPart**, and **module** identify unpackaged code segments. The specification **GLOBALFRAME [configPartList]** identifies unpackaged global frames. Unpackaged global frames of a configuration are treated as a unit and are swappable by default. If any of these global frames are made **IN** or **RESIDENT**, all of these frames are made to be so. The specification **CODEPACK [nameList]** identifies packaged code, using names of the code packs in the packaging specifications. The specification **FRAMEPACK [nameList]** identifies packaged global frames, using names of the frame packs in the packaging specifications. (See the chapter on the Packager for more information on packaging specifications.) The specification **BCD [nameList]** identifies the descriptive portion of the input BCDs. Specifications with the keyword **ALL** apply to all items. For example, **GLOBALFRAME[ALL]** identifies all unpackaged global frames, **CODEPACK[ALL]** identifies all code packs, and **BCD[ALL]** identifies the descriptive portions of all the input BCDs. For backward compatibility, **SPACE** is a synonym for **CODEPACK** and **FRAME** is a synonym for **FRAMEPACK**.

Ordinary Parameter File Entries:

GFT: *number*;

allows *number* entries in the global frame table.

PROCESSES: *number*;

allows at least *number* processes.

Special Parameter File Entries:

FRAMEPAGES: *number*;

allows at least *number* pages for the initial local frame heap. The frame heap will contain more pages if the **FRAMEWEIGHT** entries define more space.

FRAMEWEIGHT: *frameSizeIndex*, *weight* (*listEnd*);

makes the frame heap contain at least *weight* frames with index *frameSizeIndex*. This entry can occur for each frame size index. *listEnd* controls how the lists in the frame heap chain to larger sizes; it can be either empty, **INDIRECT**[*index*], or **END**. If the space available for local frames is not exhausted by the requested counts, additional frames of all sizes will be generated in proportion to the weights given.

IN: *list*;

specifies a list of modules, code packs, frame packs, etc., to be initially resident. This can occur multiple times.

PDAPAGES: *number*;

allows *number* pages for the Process Data Area. The number of pages allocated is the larger of *number* and that required to allow the number of processes specified.

RESIDENT: *list*;

specifies a list of modules, code packs, frame packs, etc., to be resident. This can occur multiple times.

The following entries should not be changed without first consulting a member of the Pilot group:

CODEBASE: *number*;

starts allocating code at page *number*.

MDSBASE: *number*;

sets the MDS to be page *number*.

NOTRAP: *modulelist*;

specifies which modules should not be start-trapped.

RESIDENTDESCRIPTOR: *list*;

specifies a list of modules, code packs, frame packs, etc., to have descriptors pinned in Pilot's caches. This can occur multiple times.

STATEVECTORCOUNT: *priority, count*;

allocates *count* state vectors for that priority in the process data area. There can be one entry for each priority level.

STATEVECTORSIZE: *number*;

specifies size of state vectors.

WART: *module*;

specifies which module initially gets control.

21.2.4 Examples

For example,

>MakeBoot TajoDLion[Pilot] will make TajoDLion.boot from TajoDLion.bcd using Pilot.bootmesa as the parameter file.

>MakeBoot Test ← CoPilotDLion[parm: Pilot/h makes Test.boot from CoPilotDLion.bcd using Pilot.bootmesa as the parameter file and produces a hexadecimal loadmap.

>MakeBoot TajoPlusCompiler ← TajoDLion[parm: PilotDLion, bcd: Compiler] writes TajoPlusCompiler.boot, which has both TajoDLion.bcd and Compiler.bcd loaded.

MakeDLionBootFloppyTool

MakeDLionBootFloppyTool runs under Tajo and creates Dandelion-bootable floppies. Bootable floppies are double-density floppies, either single- or double-sided. Your boot file must be a Utility Pilot client; regular Pilot needs to swap its own code, and it cannot swap it off a floppy. Bootable floppies contain a floppy file system.

22.1 Files

Retrieve **MakeDLionBootFloppyTool.bcd** from the Release directory.

22.2 User interface

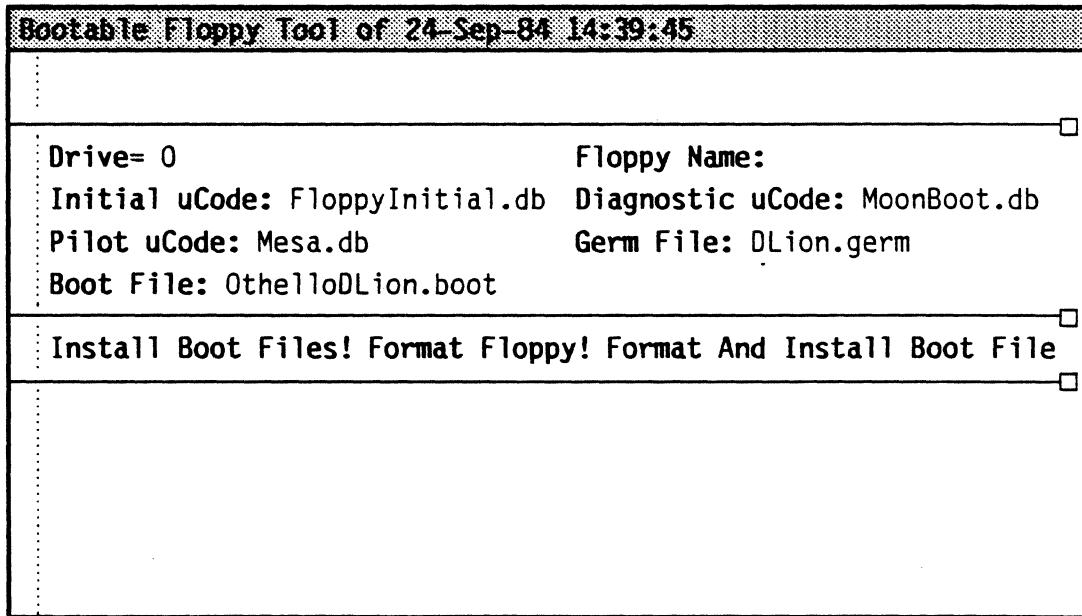


Figure 22.1: MakeDLionBootFloppyTool

22.2.1 Form subwindow

The tool's form subwindow has the following fields. They are presented here with the names of the files usually used:

Drive:	<i>YourDriveNumber</i> (typically 0)
Floppy Name:	<i>FloppyName</i>
Initial uCode:	<i>FloppyInitial.db or TridentFloppyInitial.db</i>
Pilot uCode:	<i>Mesa.db or TridentMesa.db</i>
Germ File:	<i>DLion.germ or TriDLion.germ</i>
Boot File:	<i>yourBootFile.boot</i>
Diagnostic uCode:	<i>Moonboot.db (optional)</i>

22.2.2 Command subwindow

The available commands are:

- | | |
|---------------------------------------|---|
| Install Boot Files! | installs the files specified by the fields of the form subwindow on an already existing floppy file system. |
| Format Floppy! | creates a Pilot floppy file system. |
| Format and Install Boot Files! | formats the floppy and installs the files specified by the fields of the form subwindow. |

In all cases, the process is accompanied by feedback, as it takes a few minutes to write the floppy. If you wish a disk that only has diagnostic microcode, then names for initial microcode, pilot microcode, germ, and boot file are not required.



Packager

The Mesa Packager is a tool that allows you to alter the swapping characteristics of programs. Unpackaged code is swapped in the units of compilation. That is, all the code in a particular module is either all swapped in or all swapped out together. However, efficient use of virtual memory often requires the programmer to be mindful of swapping behavior, lest thrashing occur. The Packager allows the programmer to explicitly group components of modules together into swapping units. For example, a code pack can be defined that includes the code for a several procedures from several different modules; a frame pack can be defined that groups the global frames of a number of modules into a single swapping unit.

In an unpackaged program, all code for a module is swapped as a unit, but some parts of a module are typically "colder" (less frequently referenced) than others; an example is initialization code. A program's performance would be improved if the code for colder procedures were not swapped along with that for warmer procedures. You can split the module to get this improvement, but then logically related procedures and data would no longer be contained in a single source unit.

The Packager gives you fine control over the placement of procedures in code packs. You can, for example, define a code pack that contains just the "cold" procedures from several modules. It is your responsibility, however, to split the code and global frames into reasonable packs, since the Packager simply does what you tell it. It attaches no particular semantics to a pack, except that the pack is swapped as a unit. The order in which you define code packs is significant, as is discussed below (in the section on Packaging description language.)

Conceptually, the Packager loads all modules into a single space and then shuffles the procedures around into appropriate subspaces. The packaged code is then written onto a single file. (If the code is more than 32K words, it must be packaged into multiple code segments, each requiring less than 32K. Code segments are described below along with the packaging description language.)

The Packager also supports the definition of swap units for global frames, called *frame packs*. In an unpackaged program, Makeboot (or the Loader) allocates the global frames for all of a configuration's modules in a single space. Using the Packager, you can define multiple frame packs, each containing the global frames for a set of modules. Makeboot (or

the Loader) will assign these frame packs later to separate spaces that will be swapped independently.

The Packager is a post-processor that is separate from the Compiler and Binder, and no changes to Mesa source files or configuration descriptions are needed in order to do packaging. Its operation resembles that of the Binder.

Fine points: The code rearrangement done by the Packager should not be confused with the Binder's code packing, which was described in the *Mesa Language Manual*. Code packing allows the code for several modules to be packed into a single segment, and is intended to reduce the breakage caused by the allocation of an integral number of pages to each code segment. While packing is still supported by the Binder, the same results can easily be obtained with the Packager.

23.1 Files

Retrieve **Tools > Packager .bcd** from the Release directory.

23.2 User interface

Like the Binder and Compiler, the Packager runs in the Executive and accepts a sequence of commands on the command line. A Packager command usually has one of the forms:

```
>Packager outputBcdFile • packFile[inputBcdFile]/switches  
>Packager packFile[inputBcdFile]/switches
```

(There is also an extractor-like notation for specifying the output files, which is described at the end of this section.)

The default extension for **packFile**, which contains the packaging description, is **.pack**; for **inputBcdFile** and **outputBcdFile** it is **.bcd**. The second form defaults **outputBcdFile** to be the root name of **packFile** with extension **.bcd**.

The switches are a sequence of zero or more letters. Each letter is interpreted as a separate switch designator and can be preceded by a **-** or **~** to reverse its sense. The switches include **/c** (constants shared between code packs), **/p** (pause after processing the command if there were any errors), **/l** (list), and **/m** (map).

The code segment contains multiword constants referenced by the code. The compiler keeps a literal table so that if the same constant is referenced by two different procedures within the same module, they share a single copy of the constant. If the two procedures end up in different code packs, this can lead to undesirable swapping characteristics. If, however, one of the packs is very "hot," and is likely to be swapped in whenever the other is running, then it is reasonable to have only a single copy of the constant. If the switch **/c** is specified, the packager will share multiword constants between code packs; otherwise the constants will be replicated for each pack referencing them. In actual practice, this replication is often "free" since code packs occupy an integral number of pages.

If the switch **/l** is specified, a listing is produced of the procedures that were actually placed in each code pack, as well as the module instances placed in each frame pack. This listing is in the form of a valid packaging description and can be used in place of the

original packaging description. The listing is output to the file with the root name of **packFile** but the extension **.list**.

If the switch **/m** is specified, the Packager produces a map of the code and frame packs on the file with the root name of **packFile** and extension **.map**. For a code pack, the map indicates for each procedure:

- its length in bytes,
- its entry vector index,
- the byte offset of its code from the beginning of the segment,
- its initial byte PC (byte offset of the code from the module's entry vector),
- its module, and
- its name (if a top-level procedure).

Procedures that are not at the top level (i.e., that are nested inside another) are listed below the procedure containing them. The map also includes for each module, the offset and length of its entry vector, and the read-only data shared by its procedures.

In addition to the procedure bodies, the code pack also contains other information. The entry vector (EV) is the mechanism used at runtime to find the initial PC of each procedure in the module. If the module is bound with code links (see Appendix D of the *Mesa Language Manual*) the packager will reserve space ahead of the entry vector to hold the links (LNKS). As the entry vector must lie on a quadword boundary, the size of the links space may not exactly correspond to the number of links reported in the compiler log. The pack also contains multiword constants (<data>) referenced by procedures in the code pack. As a rule of thumb, a constant follows the first procedure in the pack that references it.

For a frame pack, the map indicates for each global frame

- its length in words,
- its word offset if loaded with code links,
- its word offset if loaded with frame links, and
- the module name corresponding to this global frame.

The map also notes for each frame pack its length in pages as well as the number of unused words in the last page. Global frames are aligned on quadword boundaries, so the offset of a given frame is not exactly the offset of the previous frame plus its size.

The Packager writes a summary of the commands on the file **Packager.log**. Any errors are logged on a file with the same root name as the **packFile**, but with the extension **.errlog**.

An extractor-like notation can also be used on the Packager's command line. Commands in this format allow more control over the names of the output files produced by the Packager. One of these commands has the form:

```
>Packager [key1: file1, ..., keyn: filen] ←
    packFile[inputBcdFile]/switches
```

Each *keyi* can be one of *output*, *list*, or *map*. The corresponding *filei* names, respectively, the output object file, the code and frame pack listing file, and the map file; the default extensions are in turn *.bcd*, *.list*, and *.map*. If the keyword *list* or *map* is specified, the Packager will generate the associated output file and it is not necessary to also specify the */l* or */m* switch.

23.3 Information about modules

Any particular module is made of the following:

- *Named procedures*. A module consists of zero or more named procedures.
- *Mainline code*. A module always contains mainline code, which is automatically executed as part of the invocation of the first procedure called in any particular module. Because the mainline code of a module almost always contains only initialization code, the packaging language contains some special constructs for both excluding it from and including it in code packs. (Because the mainline code is implemented as an anonymous procedure, it is often called the main procedure of a module.) The main procedure is named using the keyword **MAIN**.
- *Entry vectors*. The entry vector is a map to the starting location of each procedure in a module, and is referenced in order to call any procedure within that module. The entry vector is *not* referenced during a procedure's **BEGINS**; the entry vector of a procedure is *not* referenced when a procedure calls another procedure (the entry vector of the destination procedure is referenced, and it may be the same as the entry vector of the calling procedure); the entry vector of a procedure is *not* referenced when the procedure returns.
- *Catch code*. Catch code is implementation of the catching of signals either by **ENABLE** or by **!**. Since catch code is usually executed only in exceptional situations, it is placed in a separate unit that may be packaged separately from all procedures in a module.
- *Global frames*. Global frames are storage and overhead required for the execution of any procedure or the catch code within a module. Global frames are swapped in whenever any procedure, main, or catch code of a module is executing. They contain a small amount of information needed by the Mesa environment in order to locate procedures and any variables the programmer has declared having the scope of the entire module. Depending upon coding style, global frames vary in size from a few words to being quite large.
- *Multiword read-only constants*. A module contains zero or more multiword read-only constants that are used during the execution of the procedures within the module. These constants are shared by several procedures whenever possible (that is, whenever they are equal).

Every module has a global frame, entry vector, and mainline procedure. A module can be written that has no procedures; a module has no catch code if it does not use the constructs **ENABLE** or **!;** modules often have no multiword constants.

23.4 Packaging description language

A packaging description consists of a sequence of code segment, frame pack, and merge specifications (merging is used to combine previously defined code segments, and is discussed later).

```
PackagingDesc    ::= DescSeries . | DescSeries ;
DescSeries       ::= DescItem | DescSeries ; DescItem | DescSeries . DescItem
DescItem         ::= CodeSegment | FramePack | Merge
```

23.4.1 Code segments

A code segment contains the code for a number of code packs and must be less than 32K words in length. As noted previously, the effect of the Packager is to combine the code for a set of modules into a single segment and then shuffle the procedures around into swap units according to your code pack descriptions.

If the total amount of code exceeds 32K, then you must define several segments. However, each module must be assigned to only one segment. Although the procedures of a module can be contained in several different code packs of a segment, all such code packs must be defined in the same segment. It is not possible to split a module across segments.

```
CodeSegment      ::= identifier : SEGMENT = SegmentBody
SegmentBody     ::= { CodePackSeries } | BEGIN CodePackSeries END
CodePackSeries   ::= CodePack | CodePackSeries ; CodePack | CodePackSeries ;
```

If you use the /c switch, you should define the code packs in order from the "hottest" (containing the most frequently referenced procedures) to the "coldest," with the hottest code packs defined first. This order determines the placement of multiword read-only constants that are shared by several procedures and are thus not strictly a part of any procedure. In any case, the entry vector for a module must precede any procedures from that module (the EV is an array of unsigned byte offsets of the beginnings of the procedures).

```
CodePack        ::= identifier : CODE PACK = CodePackBody |
                    ComponentDesc | DiscardCodePack -- defined later
CodePackBody    ::= { Excepting ComponentSeries } |
                    BEGIN Excepting ComponentSeries END
Excepting       ::= ... -- defined later
```

```
ComponentSeries ::= ComponentDesc |
ComponentSeries ; ComponentDesc |
ComponentSeries ;
```

Each **ComponentDesc** describes a collection of procedures that are to be included in the code pack. Conceptually, this is just a list of the procedures names, qualified when necessary by the names of containing configurations and modules. However, since long lists of procedure names can be awkward, the packaging language contains several constructs for abbreviating the description. Specifically, you describe each code pack as a list of components (configurations, subconfigurations, or modules), optionally listing the items from the component that are to be included in or excluded from the pack.

```
ComponentDesc      ::= Component |
Component [ ItemList ] |
Component EXCEPT [ ItemList ] |
Component EXCEPT PackList |
Component [ ItemList ] EXCEPT PackList |
Component EXCEPT PackList , [ ItemList ] |
MainOF | -- defined later
CatchOF | -- defined later
EntryOF -- defined later

Component        ::= identifier | Component . identifier

ItemList          ::= Item | ItemList , Item

Item              ::= identifier | MAIN | ENTRY VECTOR | CATCH CODE

PackList         ::= identifier | PackList , identifier
```

Each **ComponentDesc** describes procedures from the configuration or module named by **Component**. In order to uniquely specify a configuration or module, you can qualify its name by the names of enclosing configurations (and you only have to give the qualifying names necessary to uniquely specify it).

Because code is being rearranged, **Component** must refer to a module or configuration prototype, not to an instance. As described in the *Mesa Language Manual*, configurations can include both instances of modules and configurations, and their prototypes (the object files) from which such instances are made. Since different instances of the same prototype in a configuration share the same code, the Packager requires that a **Component** in a code pack name a prototype. However, because each module instance has its own global frame, a **Component** in a frame pack may name an instance.

Some forms of **ComponentDesc** include a list of items, either preceding or following the **EXCEPT** keyword. These must be directly contained in the module or configuration named by its **Component**. If **Component** refers to a module, then each item must name one of the module's procedures; if it names a configuration, the items must be modules or subconfigurations that the configuration directly contains. Most of the different forms of **ComponentDesc** apply to both modules and configurations. The six different forms are interpreted as follows:

Component

All procedures in the module or configuration are included in the code pack, except possibly main procedures, catch code, or entry vectors (see below).

Component [ItemList]

Only the named items of the component are included. If the component is a module, the items must be procedures contained within it (at the outermost level, not nested procedures; nested procedures are included along with the enclosing procedures). If the component is a configuration, the items must be directly contained subconfigurations or modules.

Component EXCEPT [ItemList]

All of the component is included except for the listed items. The items bear the same relationship to the component as in the form above.

Component EXCEPT PackList

The included procedures are those contained in the component that are not included in any of the code packs in the **PackList**. The **PackList** may name only code packs contained in the current segment. This applies to the next two forms as well.

Component [ItemList] EXCEPT PackList

Component must name a configuration. The items must be modules or configurations that it directly contains; their procedures that are not contained in any of the code packs in the **PackList** are included.

Component EXCEPT PackList, [ItemList]

If **Component** names a module, the included procedures are those not named in the **ItemList** and not included in any of the code packs in the **PackList**. If **Component** names a configuration, the included procedures are those not contained in any item and not included in any of the code packs in the **PackList**.

The first three forms of a component description are called *explicit*. The last three are *implicit*, since they define some of a code pack's procedures implicitly in terms of other code packs. Implicit **ComponentDescs** are convenient because they let you abbreviate the specification of procedures. However, you may abbreviate the specification of a component's procedures only once.

Fine point: The restriction on implicit component descriptions may be stated more precisely as follows: in each code pack of a **PackList** in an **EXCEPT** clause, any **ComponentDesc** with a **Component** that contains or is contained in the **Component** of the implicit **ComponentDesc** must be explicit.

There is one more option for defining a **CodePack**. You may use an unnamed **ComponentDesc** when the code pack contains procedures from only a single module or configuration. In this case, the code pack takes its name from that module or configuration. Although the syntax allows it, the **MainOF**, **CatchOF**, and **EntryOF** forms of component descriptions cannot be used to specify an unnamed code pack.

23.4.1.1 Placement of entry vectors, main procedures, catch code

Often the entry vectors, main code, and catch code of modules are treated quite differently from the procedures in the modules. The Packager has special syntax to allow the programmer to place these items more easily.

The **Excepting** clause may appear optionally in a **CodePack** header:

Excepting :: = empty | EXCEPT [ExceptingSeries] ;

ExceptingSeries :: = ExceptingItem | ExceptingItem, ExceptingSeries;

ExceptingItem :: = MAIN | ENTRY VECTOR | CATCH CODE;

This **Excepting** clause lets you exclude from a code pack any mainline code and/or entry vectors and/or catch code contained in the modules of the pack. Since main procedures are executed just once when a module is started, they are often placed in the coldest code pack. Entry vectors are usually included in the hottest code pack. They might be placed together in a separate code pack, or they might be mixed in with code from a logically disjoint pack when the programmer knows that this pack will be the only caller into a particular module. Catch code placement must be carefully weighed by the programmer so that fielding expected signals does not induce unwanted swapping behavior.

You can use the last variants of **ComponentDesc** to include the main procedures, catch code, or entry vectors that were excluded in other code packs of a segment.

MainOF :: = MAIN OF PackList

CatchOF :: = CATCH CODE OF PackList

EntryOF :: = ENTRY VECTOR OF PackList

The main procedures (or catch code or entry vectors) of all of the modules contained in the code packs of the **PackList** are included in the current code pack. The **PackList** must name code packs in the current segment. Each code pack in the list will normally have an **Excepting** clause specified in its header.

23.4.2 Discarded code packs

Discarded code packs allow you to throw away the code for procedures that are not needed. The procedures included in one of these code packs are marked as being unbound, and their code is not copied to the output file.

A discarded code pack is declared much like an ordinary code pack, except for the additional keyword **DISCARD** preceding the usual keywords **CODE PACK**.

DiscardCodePack :: = identifier : DISCARD CODE PACK = CodePackBody

23.4.3 Frame packs

A frame pack contains the global frames for a collection of modules. Because global frames have no finer structure (the storage for each procedure's variables is already allocated separately in local frames), you cannot split a global frame into more than one swap unit.

```
FramePack      ::= identifier : FRAME PACK = FramePackBody |
                    FrameMerge -- defined later
```

```
FramePackBody ::= { ComponentSeries } | BEGIN ComponentSeries END
```

Only the following two **ComponentDesc** variants are allowed in frame pack descriptions. The second form is valid only if the **Component** names a configuration:

```
ComponentDesc ::= Component | Component [ ItemList ]
```

Unlike code packs, a **Component** for a frame pack may name a module or configuration instance. If **Component** refers to a module, that module's frame is included in the swap unit (and only the first form may be used). If it names a configuration, the frame for each module in the configuration is included (in the first form), or the frames of the modules named in **ItemList** are included (in the second form).

Fine point: Future versions of the Packager may support **EXCEPT** clauses for frame packs.

23.4.4 Merging

A **Merge** construct lets you combine existing or previously merged code segments as well as two or more existing or previously merged frame packs. Each code pack of the merged segment consists of the procedures from one or more code packs from the original segments. The original segments (and their code packs) are superseded by the merging.

Merging is useful in the packaging of very large programs that are themselves comprised of large programs with separate packaging descriptions. Merging allows related code packs from different segments to be swapped as a unit and reduces the breakage in code packs and code segments. For example, it may make sense to merge the resident or the initialization code packs of several segments, even though the segments are not otherwise logically related.

```
Merge          ::= identifier : SEGMENT MERGES SegList = SegmentBody
```

```
SegList        ::= identifier | SegList , identifier
```

As before, the segment contains a series of named or unnamed code pack descriptions. However, the specification of these code packs is in terms of previously defined code packs, not in terms of modules and configurations. (Although the syntax allows it, a **CodePackBody** in a merged segment can not contain an **ExceptMain** clause.)

```
CodePack       ::= identifier : CODE PACK = CodePackBody | ComponentDesc
```

In a merged segment, a **ComponentDesc** must name a code pack of a previously defined segment. The name can be qualified by the containing segment when it would otherwise be ambiguous.

ComponentDesc ::= **Component**

The named **CodePack** variant can be used to combine two or more existing code packs, while the unnamed **ComponentDesc** variant is used to copy an existing code pack into the new code segment

As in unmerged code segments, the order in which you specify the code packs of the merge is important. They should be declared in order from "hottest" to "coldest."

Merged code segments, like unmerged code segments, may not be longer than 32K words in length. Thus, it may not be possible to combine the resident parts of all segments of a large system into a single swap unit.

Previously merged or existing frame packs may also be merged into a single swap unit:

```
FrameMerge      ::= identifier : FRAME PACK MERGES FramePackList ;
FramePackList   ::= FramePack | FramePack, FramePackList
```

23.4.5 Rules governing packaging descriptions

For a packaging description to make sense, the following rules must be observed:

- You have to account for every procedure (including main), catch code, entry vector, and global frame. Each procedure must be placed in some code pack. Likewise, each global frame must be placed in some frame pack.
- A procedure can be placed in only one code pack. Likewise, a global frame can be placed in only one frame pack.
- The entry vector as well as all procedures and catch code of a module must appear in a single code segment (since the module's entry vector is required to reference the procedures and entry vector.)
- The entry vector of a module must be placed before any of its other code, including the catch code.
- The code pack identifiers within a code segment must be distinct, but code packs in different segments may have the same name. All frame pack identifiers must be distinct.
- A component of a code pack cannot name a module or configuration instance. However, a component of a frame pack may name an instance.

Fine point: If a module has been table-compiled, its code can be included in a code pack, but only as a unit.

23.4.6 Placement of multiword read-only constants

The Packager replicates multiword constants that are referenced in multiple code packs unless the /c switch is specified on the command line. If /c is given, the order in which code packs are specified is used to make the assignments of multiword read-only constants within a module. The Packager stores a multiword constant in the first code pack that contains a procedure using it. Specifying the "hot" code packs first will thus help to ensure that the additional data needed by a procedure is already in memory.

Fine point: Previous versions of the packager did not replicate constants; they behaved as if the /c switch were always present.

23.4.7 Example

This section presents a simple packaging description. For further examples you might want to look at the packaging description for something real.

The packaging description for **Lex** distributes its procedures into three code packs (**LexicalStringManagement**, **CollectAndDispatchCommands**, and **InitAndSeldomUsed**), depending upon logical function and frequency of use. It also places the global frames for **Lex**'s two modules into separate frame packs, **UtilityFrames** and **DriverFrames**.

```

Lex: SEGMENT =
  BEGIN
    LexicalStringManagement: CODE PACK =
      BEGIN
        Lexicon EXCEPT CollectAndDispatchCommands, [MAIN, CATCH CODE];
        LexiconClient [ENTRY VECTOR];
      END;
    CollectAndDispatchCommands: CODE PACK =
      BEGIN
        Lexicon[PrintLexicon];
        LexiconClient EXCEPT [ENTRY VECTOR, CATCH CODE];
      END;
    InitAndSeldomUsed: CODE PACK =
      BEGIN
        LexiconClient [CATCH CODE];
        Lexicon[MAIN, CATCH CODE];
      END;
  END;

-- Frame packs

UtilityFrames: FRAME PACK = {Lexicon};
DriverFrame: FRAME PACK = {LexiconClient}.

```

LexiconClient is placed in **CollectAndDispatchCommands**, a less frequently used code pack, while its entry vector and the procedures that it calls frequently (most of **Lexicon**'s procedures) are placed in **LexicalStringManagement**, the most frequently used code pack.

The remaining code (mainline code and catch code), which is seldom called, is placed in **InitAndSeldom** used, a code pack that is seldom used.

The global frame of **Lexicon**, which contains the hottest procedures, is placed in the frame pack **UtilityFrames**. The remaining global frame (for **LexiconClient**) is placed in **DriverFrames**.

23.5 Operation

The Packager is run as a post-processor that reads a single object file and a packaging description, and writes a new output object file with a different name. Its operation resembles that of the Binder, except that all symbols for the input object file must be on the disk. The Packager needs these to identify procedures and frame packs, and to locate the code for procedures. The output object file contains the reorganized code of the input object file, but not symbols (i.e., code is copied, symbols are not). The output object file also contains information about the global frame packs for later use by Makeboot and the Pilot Loader.

A packaged object file can be loaded and executed, or bound with other object files using the Binder. However, a packaged object file cannot be further repackaged, since this would require that symbol tables be modified, which would, in turn, cause considerable operational problems. It is possible to combine separate packaging descriptions in a single run with code segment merging, in the sense that code packs from the original descriptions can be merged together into new, larger code packs without modifying the original descriptions.

Although the Packager does not read multiple packaging descriptions, the syntax is designed to allow easy merging of separate descriptions using the Executive's **Copy** command. For example, if **BigApplication** were made up of descriptions for **FirstPiece** and **SecondPiece**, plus a **MergePieces** that specified how to merge the two segments, then the following command would combine the three separate descriptions:

```
>Copy Big.pack ← First.pack Second.pack MergePieces.pack
```

Because the Packager must access the code of every procedure and the symbol table of every module of the system it is packaging, and must also copy the code for each procedure to the output file in random order (in the worst case), it is not very fast. It is roughly an order of magnitude slower than the Binder.



Debugger

This chapter describes the Pilot-based interactive Mesa debugger, CoPilot. CoPilot supports source-level debugging; it allows users to set breakpoints, trace program execution, display the runtime state, and interpret Mesa statements. CoPilot is intended for use by experienced programmers familiar with Mesa.

The runtime and debugging facilities differ in their relationship to the user program. Pilot provides the code necessary for your program to communicate with CoPilot; it resides with the user program. CoPilot, however, resides in a different core image (in addition to a separate logical volume of type `debugger`) that is loaded by the germ when called for; CoPilot operates with a complete *world-swap*. This protects the client and the debugger from each other as well as provides the address space required to implement all of CoPilot's capabilities.

24.1 Files

To run the debugger, use Othello to fetch `CoPilotDLion.boot` onto a logical volume (type `debugger` is recommended) as the boot file for that volume.

24.2 Installing and invoking CoPilot

CoPilot must be installed before a client program can use its facilities. Once fetched, booting the volume installs CoPilot and makes it ready to accept calls from clients. This operation saves the debugger's core image. Unlike normal boot files, CoPilot can be re-entered many times even though it is booted only once. It must be re-installed whenever you begin using a new germ or change the quantity or configuration of memory on the system. To re-install CoPilot, simply re-boot the volume with Othello or the Herald window. See the Othello appendix for further details. While the debugger is installing itself, it examines the (optional) `User.cm` for a [`Debugger`] section.

When CoPilot is installed for the first time, it creates files to hold the client's core-image (`Debuggee.outload`) and its own core-image (`Debugger.outload`). If the memory configuration is changed, CoPilot must be re-installed (re-booted) and the messages `Recreating Debuggee.outload` and `Recreating Debugger.outload` are displayed. CoPilot prevents any attempt to modify or delete these files; Tajo may be used for this purpose.

CoPilot users may have a debugger installed that can be used to catch and diagnose CoPilot failures. This debugger is just another instance of CoPilot installed on a logical volume of type **debuggerDebugger** (this debugger has come to be called CoCoPilot). CoCoPilot must be installed *before* CoPilot is installed. If CoCoPilot is re-installed, CoPilot must also be re-installed. It is recommended that **CoPilot** be put in the **Boot** line of the **User.cm** on the CoCoPilot volume.

Fine point: During the later stages of initialization, Pilot searches for an installed debugger to use. It looks on all volumes of a type one higher than the one on which the boot file resides. For example, if the boot file is on a volume of type **normal**, Pilot looks on volumes of type **debugger**. Occasionally, it is desirable to use an installed debugger other than the one that Pilot would normally choose. In these cases, use Othello's **Set Debugger Pointers** command, which also allows you to have a client and a debugger on volumes of the same type. However, if any other systems are rooted on volumes of the same type as an installed debugger, it is necessary to always boot them (and good practice to boot the debugger itself) with the open-system-volume-only "%" boot switch. Otherwise, running one of the other boot files will delete the temporary files from underneath the installed debugger, leading to a **Disk Label Check** when the debugger is next used. If any volume is booted with the "5" switch, Pilot will enter the teledebugger (MP code 915) rather than look for a debugger.

There are several ways of invoking the debugger. In the Xerox Development Environment for example, **CALLDEBUG** (**SHIFT-ABORT**) simply interrupts your program. In the course of running your program, you may enter the debugger for several other reasons. Your program may generate an uncaught signal, execute a breakpoint/tracepoint that has been placed in your program, require map logging, or make an explicit call to the debugger. CoPilot has different cursors that it displays for each reason it was entered; they are **Unc Sig** (for **Uncaught Signal**), **Call Dbug** (for explicit calls, including **Address Fault** and **WriteProtectFault**), **Brk Pt** (for **BreakPoint**), **Int** (for **Interrupt**), and **Map Log** (for **Processing VM Map**).

The first time CoPilot is invoked for a client marks the start of a *new session*. The debugger takes several special additional actions for a new session, as opposed to when it is simply re-entered. First, it resets the **Debug.log** to be empty and displays the date and time. Next, CoPilot forgets everything it knew about the previous client. Last, CoPilot sets the user password to be empty if the current user name is not the same as the user name in the **User.cm**.

24.2.1 Teledebugging

It is possible to debug clients over the Ethernet. See the following section on low-level facilities for details.

24.3 User interface

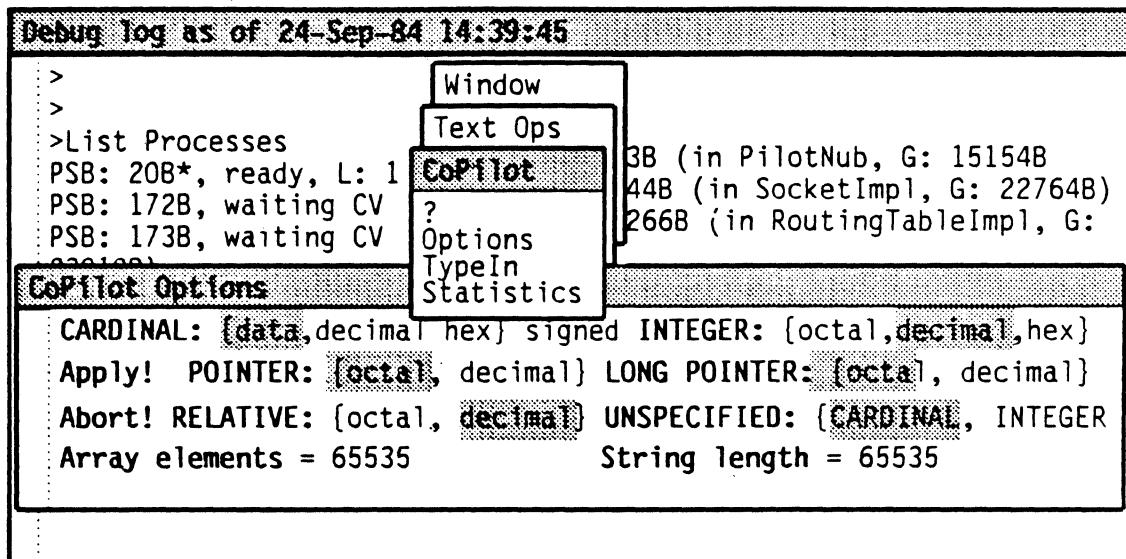


Figure 24.1: CoPilot

When initialized, CoPilot creates two windows: the `Debug.log` window, which becomes a record of the debugging session, and a Herald window that displays CoPilot's version number and date, and various messages from the debugger. These windows may be manipulated by the window manager that comes with your debugger. CoPilot runs in the standard user environment (Tajo).

The user interface to the debugger is controlled by a command processor that invokes a collection of procedures for managing breakpoints, examining user data symbolically, and setting the context in which user symbols are looked up. Data in your program is examined by the debugger's interpreter. The interpreter also allows you to change values of variables in the middle of program execution. See the next section for a complete description of the interpreter.

24.3.1 Talking to the debugger

The debugger accepts commands either from the `Debug.log` window or from selected menu items in a File window. The input conventions of the debugger's command processor are summarized in the next section. The command processor prompt character is `>` (the character is repeated once for each nesting level of the debugger). The standard input editing characters (`BS` to delete a character and `BW` to delete a word) are allowed. Whenever a valid command is recognized, the debugger prompts for the parameters associated with that command (if any are required). Pressing `DELETE` terminates the command; `?` gives a list of valid commands. When a command requires a `[confirm]` (`RETURN`), the debugger enters wait-for-`DELETE` mode if an invalid character is typed.

When receiving commands, the debugger extends each input character to the maximal unique string that it specifies. Whenever an invalid character is typed, a `?` is displayed and you are returned to command level. Pressing `?` at any point during command selection prompts you with the collection of valid characters (in upper case) and their associated maximal strings (in lower case) and returns you to command level. Whenever a valid

command is recognized, you are prompted for parameters. Pressing **DELETE** at any point during command selection or parameter collection returns you to the command processor; pressing **ABORT** at any point during command execution aborts the command.

Current Context

Interpreting symbols (including displaying variables, setting breakpoints, and calling procedures) occurs in the *current context*; it consists of the current frame and its corresponding module, configuration, and process. The symbol lookup algorithm used by the debugger is to search the runtime stack of procedure frames in *Last-In-First-Out* order. First the local frame of the current procedure is examined, next its associated global frame. The search continues by following the return link to the next local frame. This continues until either the symbol is found or the root of the process is encountered.

When you first enter the debugger, the context is set to the frame of whatever process is currently running. Certain commands make it simple to enumerate contexts (**List Processes**, **List Configurations**), to change between contexts (**SET Root configuration**, **SET Module context**), to display the current context (**CURRENT context**), and to examine the current dynamic state (**Display Stack**).

Looking up Symbols

Whenever the debugger needs symbols to display some information, it searches for the original compiler-output object file before looking for symbols where they were last copied by the Binder. Types used, but not declared, within a module are looked up using the same algorithm as in the Compiler. If the interface module containing the original declaration is unavailable, the debugger uses whatever information has been copied into the symbol table of the module using that type.

Leaving the Debugger

In the debugger, you may execute any number of commands to examine (and change) the state of your program. When you are finished, you may decide either to continue execution of your program (**Proceed**), terminate execution of your program (**Quit**), or end the debugging session completely and boot the physical volume (**Kill**). The next subsection contains further details on these commands. It is also possible to boot other logical volumes with the Herald window.

24.3.1.1 Input conventions

String Input

Identifiers are sequences of characters beginning with an upper- or lower-case letter and terminating with a space (**SPACE**) or a carriage return (**RETURN**); *identifiers must be typed with correct capitalization*. The debugger echoes a delimiting character of its own choice to minimize loss of information from the display.

Numeric Input

A numeric parameter is a sequence of characters terminated by **SPACE** or **RETURN**. If the parameter is not a numeric constant, it is processed by CoPilot's interpreter; any expression that evaluates to a number is legal (the target type must be **(LONG) INTEGER**,

CARDINAL, or **UNSPECIFIED**). The default radix is octal for addresses (and input to octal commands) and decimal for everything else (unless otherwise specified with the **CoPilot Options** window). The **D** or **d** suffix forces decimal interpretation; **B** or **b** forces octal. Numbers with a leading zero are considered **LONG**.

Default Values

The debugger saves the last values used as parameters to all of the commands; these values may be recalled by the **COMPLETE** key. The following parameters have default values that may be used or inspected by pressing **COMPLETE**: *octal read address, octal write address, ascii read address, root configuration, configuration, module, procedure, condition, expression, process, address, and frame*. After the default parameter is displayed by the debugger, the standard input editing characters may be used to modify it. Striking the **COMPLETE** key to the command processor uses the last command as the default command (i.e., you receive the prompt for the parameters, if any, for the previously executed command).

24.3.1.2 Output conventions

A "?" in any variable display uniformly means that the value is out of range. An ellipsis ("...") indicates that there are additional fields present in a record that cannot be displayed due to lack of symbol table information. This can happen either in **OVERLaid** records or because a **DEFINITIONS** file is not present on the disk. In display stack mode, variables declared in nested blocks are shown indented according to their nesting level.

The **CoPilot Options** window allows you to change the default format the debugger uses in displaying values of variables. This window is created by selecting the **Options** item in the **CoPilot** menu and operates as a normal Options window (i.e., invoke **Apply!** to effect the changes made, **Abort!** to restore them to the previous options).

The **CARDINAL**, **INTEGER**, **POINTER**, **LONG POINTER**, and **RELATIVE (POINTER)** items are used to set the default output radix for that type. For **CARDINAL** and **INTEGER**, the default representation is signed or unsigned, depending on whether the boolean item **signed** is turned on or off. The **UNSPECIFIED** item is used to set the default type for displaying **UNSPECIFIED** variables. **Array elements** sets the number of **ARRAY** elements displayed to be the given value and **String length** sets the number of **STRING** characters displayed to the given value.

CoPilot uses these default values along with the types of variables to decide on an appropriate output format. Listed below are the built-in types that the debugger distinguishes and the convention used to display instances of each type.

ARRAY

displays elements of an array; e.g., **a = (3){[x: 0, y:0], [x: 1, y: 1], [x: 3, y:3]}**. The parenthesized value to the right of the "=" is the length of the array. Pressing **ABORT** will abort the display of long arrays. The default is to display the entire array; the **Array elements** item of the **Options** window may be used to change this.

ARRAY DESCRIPTOR

displays the descriptor followed by the contents of the array; e.g., **a = DESCRIPTOR[146013B#,3](3){[x: 0, y:0], [x: 1, y: 1], [x: 3, y:3]}**. For a

RELATIVE ARRAY DESCRIPTOR, the word **RELATIVE** is displayed first. Pressing **ABORT** will abort the display of long array descriptors. The **Array elements** item in the **Options** window also controls this.

BOOLEAN

displays **TRUE** or **FALSE**. Since **BOOLEAN** is an enumerated type = {**FALSE**, **TRUE**}, values outside this range are indicated by a ? (probably an uninitialized variable).

CARDINAL

displays an octal number terminated by a "B" as the default. This may also be altered with the **Options** window. Cardinals may be displayed as decimal, octal, or hex; signed or unsigned.

CHARACTER

displays a printing character (c) as 'c. A control character (x) other than **BLANK**, **RUBOUT**, **NUL**, **TAB**, **LF**, **FF**, **CR**, or **ESC** is displayed as ↑x. Values greater than 177B are displayed in octal.

CONDITION

displays a record containing an **UNSPECIFIED** and **timeout**; a **CARDINAL**.

ENUMERATED

displays the identifier constant used in the enumerated type declaration. For example, an instance c of the type **ChannelState: TYPE** = {**disconnected**, **busy**, **available**} is displayed as **c=busy**.

EXPORTED TYPES

displays the name of the type followed by an octal display of the contents if the length of the type is known. For example, an instance of the type **Handle: TYPE [2]** is displayed as **Handle (2) 11234B**.

INTEGER

always displays a decimal number. Uniformly, numeric output is decimal unless terminated by "B" (octal). Integer output may be changed with the **Options** window.

LONG

displays numbers following the same conventions as short numbers; i.e., **LONG CARDINAL** and **LONG UNSPECIFIED** are displayed in octal, **LONG INTEGER** in decimal.

MDSZone

displays a **POINTER**; an **UNCOUNTED ZONE** displays as a **LONG POINTER**.

MONITORLOCK

displays a record containing an **UNSPECIFIED**.

POINTER

displays an octal number, terminated with an "↑"; e.g., **p=107362B↑**. **RELATIVE POINTERS** are decimal and are terminated with "↑R"; e.g., **r=123↑R**. These defaults may be changed for **LONG POINTERS**, **RELATIVE POINTERS**, and **POINTERS** to either octal or decimal with the **Options** window.

PORt

displays two octal numbers; e.g., **p = PORT [0, 172520B]**.

PROCEDURE, SIGNAL, ERROR

displays the name of the procedure (with its local frame) and the name of the program module in which it resides (with its global frame); e.g., **GetMyChar, L: 165064B** (in **CollectParams, G: 166514B**).

PROCESS

displays a **PROCESS** (pointer to a **ProcessStateBlock**); e.g., **p = PROCESS [111B]**.

REAL

displays a floating-point number; e.g., **-1.45**.

RECORD

displays a bracketed list of each field name and its value. For example, an instance **v** of the record **Vector: RECORD [x,y: INTEGER]** is displayed as **v=[x: 9, y: -1]**. Pressing **ABORT** only aborts display of the current field.

SEQUENCE

displays as an array. For example, an instance **s** of the record **Sequence: RECORD [length: UnsignedInt, text: PACKED SEQUENCE maxLength: UnsignedInt OF CHARACTER]** is displayed as **s=[length: 3, text: (3)['a, 'b, 'c]]**.

STRING

displays the name of the string, followed by its current length, its maximum length, and the string body; e.g., **s=(3,10)"foo"**. If the string is **NIL**, **s=NIL** is displayed. Pressing **ABORT** will abort the display of long strings. The default is to display the entire string; the **String length** item in the **Options** window can change this.

UNSPECIFIED

defaults to being displayed as if they were **CARDINALS**; this may be changed with the **Options** window.

Listed below are the conventions used to display context information throughout the debugger:

ProcedureName, L: nnnnnB, PC: nnnB (in ModuleName, G: nnnnnB)

A local context is displayed as the procedure name with its local frame, followed by the module name and its global frame.

ModuleName, G: nnnnnB --global frame

A global context is displayed as the module name and its global frame. If the global frame is followed by * (as nnnnnB*) it is a copy created by the NEW construct. If the global frame has not yet started, it will be followed by a ~.

In response to an expression followed by a ?, the interpreter will show:

```
Octal = Hexadecimal = Unsigned Decimal = Signed Decimal =
Byte,,Byte = Octal Byte,,Octal Byte = CHAR,,CHAR =
Nibble:Nibble,,Nibble:Nibble
```

If any of the values are 0 or out of range, they will not be shown. For LONG values the interpreter will show:

```
Octal = Hexadecimal = Decimal = OctalWord OctalWord =
Byte,,Byte Byte,,Byte
```

For example, in response to 61141B? the debugger displays

61141B = 6261X = 25185 = 98,,97 = 142B,,141B = 'b,,'a = 6:2,,6:1

and for 1234567B? it shows

1234567B = 53977X = 342391 = 34567B 5 = 57,,119 0,,5

24.3.2 Debugger commands

CoPilot provides facilities for managing breakpoints, examining user data symbolically, setting the context in which the user symbols are looked up, and directing program control.

The command tree structure for CoPilot appears at the end of this chapter. Capitalized letters are typed by the user (in either upper or lower case); Commands are extended with lower-case strings by the command processor. Each command (and its parameters) is described below.

24.3.2.1 Breakpoints

The break and trace commands apply to modules that are known within the current context. All breakpoints and tracepoints may be conditional (see **ATtach Condition**, below). An optional command string can also be attached to each breakpoint/tracepoint; it will be executed when the breakpoint/tracepoint is taken (see **ATtach Keystrokes**, below). A tracepoint is a breakpoint that automatically invokes the Display Stack

command processor, displaying the first procedure on the call stack and its parameters (trace entry), variables (trace), or results (trace exit) as appropriate.

You may set breakpoints at the following locations in your program: entry (to a procedure), exit (from a procedure), and at the closest statement boundary preceding a specific text location within a procedure or module body. The debugger can set entry breakpoints on any procedure called from within a module. However, the fact that extra symbols are required to display the parameters or the breakpoint will not be discovered until needed. Breaks on a specific text location can be set only with the **Break** command of the **Debug Ops** (or Symbiote) menu. Note that breakpoints are set in all instances of a module. When the source line of the breakpoint is displayed, the indicator <> appears to the left of the source where the breakpoint has actually been set (e.g., **IF foo THEN <> some statement;**). Before the debugger permits any breakpoints to be set using a FileWindow, the creation date in the source file is checked against the corresponding date recorded by the compiler in the bcd.

Fine point: Since there is only one exit from a procedure, the debugger shows the beginning of the procedure for exit breaks instead of indicating a potentially incorrect **RETURN** statement. Local variables may be invisible if this **RETURN** has a **PC** that is not in the block with their declarations; use source breaks on the **RETURN** statements instead of an exit break.

When a break or trace is encountered during execution, a (possibly nested) instance of the debugger is created and control transfers to the command processor, from which you may access any of the facilities described in this document. The debugger types the name of the procedure containing the breakpoint and the address and **PC** of the currently active frame. If the breakpoint has a condition associated with it, the break is taken only if the condition is satisfied. The multiple proceed counter is reset after being satisfied; e.g., a condition of 5 will actually break on the fifth, tenth, fifteenth, ... times the breakpoint is reached. To continue execution of your client program, use the **Proceed** command; to stop execution of your program, use the **Quit** command.

Fine point: Occasionally a breakpoint will be taken a second time. This is the result of a page fault that occurred as execution of the the client was resumed. It does not indicate that anything is amiss, so simply proceed.

If you compile a module with the cross-jumping switch turned on (the default), be warned that when setting source breakpoints, the actual breakpoint may not end up where you expect (e.g., you may break in the code of an **ELSE** clause when you really want the **THEN** clause if they share some common code). The message **Cross jumped!** will appear before the source of a cross-jumped module is displayed. Entry and exit breakpoints are not affected by cross jumping.

The warning **Eval stack not empty!** will be printed if the debugger is entered via either an interrupt or breakpoint with variables still on the evaluation stack; this indicates that the current value of some variables may not be in main memory, where the interpreter normally looks. Exceptions to this are entry and exit breaks; the debugger has enough information to decode the argument records that are on the stack in this case (if the appropriate symbol tables are available).

Attach (in **Debug Ops** menu of File window)

tells the debugger to ignore the time stamp in the source file when setting breaks. See **ATTach Symbols** in the sub-subsection on Low-level facilities .

ATTach Condition [number, condition]

changes a normal breakpoint into a conditional one. Arguments are a breakpoint number and a condition, which is evaluated in the context of the breakpoint. The breakpoint number is displayed when the break/tracepoint is set, and may also be obtained using the **List Breaks** command.

The three valid formats of a **Condition** are: *variable relation variable*, *variable relation number*, and *number*. Conditions include relations in the set {<, >, =, #, <=, >=}. A *number* (multiple proceeds) means "execute the break *number* times before invoking the debugger." The *variables* are interpreted expressions that are looked up in the context of the breakpoint. A *variable* may not be an expression that is more than one word long, dereferences a pointer (beware of the implicit dereference in record qualification), or indexes an array.

ATTach Keystrokes [number, command]

adds an arbitrary command string to breakpoints/tracepoints; the characters from this string are executed by the debugger when the breakpoint/tracepoint is taken. Arguments are a breakpoint number and a command string terminated with a **RETURN**. A **RETURN** can be embedded in the command string by quoting it with **CTRL-V**.

Break (in Debug Ops menu of File window)

uses the current selection to set a breakpoint. If you select **PROCEDURE** or **PROC**, a breakpoint is set on the entry to the procedure; if you select **RETURN**, a breakpoint is set on the exit of the procedure; otherwise, a breakpoint is set at the closest statement enclosing the selection. **Note:** If the module was compiled with cross jumping, breaks may be set in unpredictable places. Confirmation is given by moving the selection to the place at which the breakpoint is actually set.

For the following code fragments, a breakpoint set on **anyError** will invoke the debugger after the catch frame is entered. If a breakpoint is set on **MFile.Error**, the debugger is invoked for all signals and errors (including things like **DivideCheck**) before any decision is made to catch the signal.

```
BEGIN ENABLE MFile.Error => {anyError ← TRUE; CONTINUE};
  !MFile.Error => {anyError ← TRUE; CONTINUE};
```

If there are multiple instances of a module, the current context must match the source file. In any event, the breakpoint number or any error messages are displayed in the Herald window.

Break All Entries [module/frame]

sets a break on the entry point to each procedure in *module* or *frame* (cf. **Break Entry**); nested procedures and catch code are ignored.

Break All Xits [module/frame]

sets a break on the exit point of each procedure in *module* or *frame* (cf. **Break Xit**).

Break Entry [proc]

inserts a breakpoint at the first instruction in the procedure *proc*. Note: You can place a breakpoint on the entry to the mainline code. For a module to do this, **Break Entry** [*module name*].

Break Exit [proc]

inserts a breakpoint at the *last* instruction of the procedure body for *proc*. This catches all **RETURN** statements in the procedure. Note: You can place a breakpoint on the exit from the mainline code. For a module to do this, **Break Exit** [*module name*].

CLear All Breaks

removes all breakpoints/tracepoints.

CLear All Entries [module/frame]

removes all entry breakpoints/tracepoints in *module* or *frame*.

CLear All Xits [module/frame]

removes all exit breakpoints/tracepoints in *module* or *frame*.

CLear All Traces

removes all breakpoints/tracepoints; it is equivalent to **CLear All Breaks**.

CLear (in Debug Ops menu of File window)

clears the breakpoint or tracepoint at the location specified as above.

CLear Break [number]

removes a breakpoint by number. Pressing **RETURN** in place of a number clears the current breakpoint; i.e., the one that got you into CoPilot.

CLear Condition [number]

changes a conditional breakpoint into a normal one. Pressing **RETURN** in place of a number behaves as in **CLear Break**.

CLear Keystrokes [number]

clears any command string associated with the breakpoint. Pressing **RETURN** in place of a number behaves as in **CLear Break**.

CLear Entry Break [proc]

converse of **Break Entry**.

CLear Entry Trace [proc]

converse of **Trace Entry**; it is equivalent to **CLear Entry Break**.

CLear Xit Break [proc]

converse of **Break Xit**.

CLear Xit Trace [proc]

converse of **Trace Xit**; it is equivalent to **CLear Xit Break**.

Display Break [number]

displays a breakpoint by number. Its type (entry, exit, source), and the procedure and/or module name in which it is found are displayed; for source breakpoints, the source text is also displayed; any attached conditions or keystrokes are also shown. Pressing **RETURN** in place of a number behaves as in **CLear Break**.

List Breaks [confirm]

lists all breakpoints, displaying the same information as **Display Break**.

Trace (in Debug Ops menu of FileWindow)

sets a tracepoint at a location specified, as in **Break** above. Confirmation is given by moving the selection to the place at which the tracepoint is actually set.

Trace All Entries [module/frame]

sets a trace on the entry point to each procedure in *module* or *frame* (cf. **Trace Entry**).

Trace All Xits [module/frame]

sets a trace on the exit point of each procedure in *module* or *frame* (cf. **Trace Xit**).

Trace Entry [proc]

sets a trace on the entry of the procedure *proc*. When an entry tracepoint is encountered, display stack mode is entered and the parameters are displayed (cf. **Break Entry**).

Trace Xit [proc]

sets a trace on the exit of the procedure *proc*. When an exit tracepoint is encountered, display stack mode is entered and the return values are displayed (cf. **Break Xit**).

24.3.2.2 Display runtime state

The scope of variable lookup is limited to the current context (unless otherwise specified below to be the current configuration). What this means is the following: if the current context is a local frame, the debugger examines the local frame of each procedure in the call stack (and its associated global frame) following return links until the root of the process is encountered. If the current context is a module (global) context, just the global

frame is searched. Global frames are searched in the order: declarations, imports, directory. If the variable you wish to examine is not within the current context, use the commands that change contexts.

CoPilot displays a global frame followed by a * if the frame is a copy created by the NEW construct; it is followed by a ~ if it is not started.

AScii Read [address, n]

displays *n* (decimal) characters as a string starting at *address* (octal).

AScii Display [address, count]

interprets *address* as **POINTER TO PACKED ARRAY OF CHARACTER** and displays *count* characters.

Display Configuration

displays the name of the current configuration followed by the module name, corresponding global frame address, and instance name (if one exists) of each module in the current configuration.

Display Frame [address]

displays the contents of a frame, where *address* is its octal address (useful if you have several instances of the same module or examining a specific local frame); display stack subcommand mode is entered.

Display GlobalFrameTable

displays the module name and corresponding global frame address, pc, codebase, and gfi of all entries in the global frame table. If a frame has been unNEWed, it will be followed by the word "deleted."

Display Module [module]

displays the contents of a global frame, where *module* is the name of a program in the current configuration.

Display Process [process]

displays interesting things about *process*. This command shows you the *process*, the frame associated with *process*, and the state of the *process*. A *process* can be:

ready (ready to run and has a state vector)

waiting SV (ready to run but needs a state vector)

waiting ML (waiting on a monitor)

waiting CV (waiting on a condition variable)

frame fault, fsi: nn (needs a frame whose size index is *nn*)

page fault, address: nnnnnB (waiting for page whose address is nnnnnB; this is an address fault if location nnnnnB isn't mapped)

write fault, address: nnnnnB (waiting to write into location nnnnnB, which is write-protected)

faulted (unknown fault has occurred)

A * marks the current process. A process can be on one and only one queue (associated with a condition, monitor, ReadyList, etc.). Then you are prompted with > and you enter process subcommand mode. A response of **N** displays the next process; **S** displays the source text and loads and positions the source file in the Source window; **L** just displays the source text; **R** displays the root frame of the process; **P** displays the priority of the process; space (**SPACE**) enters the interpreter; -- delimits a comment; and **Q** or **DELETE** terminates the display and returns you to the command processor. **Note:** Either a variable of type **PROCESS** (returned as the result of a **FORK**) or an octal **PROCESS** is acceptable as input to this command (*process* is an interpreted expression).

Display Queue [id]

displays all the processes waiting on the queue associated with *id*. If *id* is simply an octal number, you are asked whether it is a condition variable (e.g, **Condition? [Y or N]**). For each process, you enter process subcommand mode. The semantics of the subcommands remain the same as in **Display Process**, with the exception of **N**, which in this case follows the link in the process. This command accepts either a condition variable, a monitor lock, a monitored record, a monitored program, or an octal pointer.

Display ReadyList

displays all the processes waiting on the queue associated with the **ReadyList**; i.e., the list of processes ready to run. For each process, you enter process subcommand mode; the semantics of the subcommands are the same as in **Display Queue**.

Display Stack

displays the procedure call stack of the current process. At each frame, the corresponding procedure name and frame address are displayed. You are prompted with >. A response of:

V displays all the frame's variables.

G displays the global variables of the module containing the current frame.

P displays the input parameters.

R displays the return values (**anon**) appears before those that are not *named* in the parameter lists.

N moves to the next frame.

J, *n*(10) jumps down the stack *n* (decimal) levels (if *n* is greater than the number of levels it can advance, the debugger tells you how far it was able to go).

S displays the source text and loads and positions the source file in a source window.
(It also sets the context for setting breakpoints in that window.)

L just displays the source text.

SPACE enters the interpreter.

-- delimits a comment.

Q or **DELETE** terminates the display and returns you to the command processor.

When the current context is a global frame, the **Display Stack** subcommands **G**, **J**, and **N** are disabled. When the debugger cannot find the symbol table for a frame on the call stack, only the **J**, **N**, **Q**, **--** and **SPACE** subcommands are allowed. For a complete description of the output format, see the section on Unrecognized structures.

Find variable [id]

displays the contents and module location of the variable named *id*, searching through only the **GlobalFrames** of all the modules in the current configuration.

Statistics (in CoPilot menu of Debug.log window)

writes statistics about CoPilot's internal caches into the debug window. It is not normally used by clients.

24.3.2.3 Current context

The current context is used to determine the domain for symbol lookup. There are commands to display the current context, to display all the configurations and processes, to restore the starting context, and to change contexts.

Every time the debugger is entered, the current context is automatically set to (1) the process that caused the debugger to be called; (2) some significant frame in the calling process, not necessarily the innermost frame (top of the call stack) of the process (for example, an uncaught signal sets the frame in which the signal was raised); and (3) the module and configuration of the local frame set in (2).

CURRENT CONTEXT

displays the name and corresponding global frame address (and instance name if one exists) of the current module, the name of the current configuration, and the **PROCESS** for the current process.

List Configurations

lists the name of each configuration that is loaded, beginning with the last configuration loaded. If you wish to see more information about a particular configuration, use the **Display Configuration** command.

List Processes

lists all processes by **PROCESS** and frame. If you wish to see more information about a particular process, use the **Display Process** command.

ReSet context

restores the context that this instance of the debugger set upon entry (see the introduction to this section). **Note:** The local frame set by this command is not necessarily the same as that set by the **Set Process Context** command.

SEt Configuration [config]

sets the current configuration to be **config**, where **config** is nested within the root configuration that is current. This command is useful for "jumping" further into the nested block structure of a configuration.

SEt Module context [module/frame]

changes the context to the program module whose name is **module** (within the current configuration). If there is more than one instance of **module**, the debugger lists the frame address of each instance and does *not* change the context. Using a **frame** address has the same effect as **SEt Octal context**.

SEt Octal context [address]

changes the current context to the frame whose address is **address**. This is useful when there are several instances of the same module or in setting the current context to a specific local frame.

SEt Process context [process]

sets the current process context to be **process** and sets the corresponding frame context to be the innermost frame associated with that **process**. Upon entering the debugger, the process context is set to the currently running process. Note that either a variable of type **PROCESS** (returned as the result of a **FORK**) or an octal **PROCESS** is acceptable as input to this command. **Note:** If the process is the same as that in which the debugger was entered, the local frame set by this command is not necessarily the same as that frame initially set by the debugger, the one that would be set by the **Reset Context** command.

SEt Root configuration [config]

sets the current configuration to be **config**, where **config** is at the outermost level (of its configuration). This command is sufficient for simple configurations of only one level. It is also useful in getting you to the outermost level of nested configurations, from which you may move "in" using **SEt Configuration**.

24.3.2.4 Program control

Certain commands allow you to determine the flow of control between the debugger and your program.

Kill session [confirm]

ends your debugging session, swaps back to the client world, and executes **TemporaryBooting.BootButton**.

Proceed [confirm]

continues execution of the program (i.e., proceeds from a breakpoint, resumes from an uncaught signal).

Quit [confirm]

raises the signal **ABORTED** in the process that entered the debugger. If the process was already processing an uncaught **ABORTED** signal (perhaps from a previous **Quit** command), this command passes the signal **UNWIND** to each frame of the process and then simulates a **RETURN** with no results by the root frame of the process, causing the process to be deleted. If this process is supposed to return any results, the parent process will get a stack error when it attempts to **JOIN** the process.

Start [address] [Confirm]

starts execution of the module whose frame is **address**. If the module has already been started, a **RESTART** will be done. Unlike the **START** statement in the Mesa language, no parameters may be passed.

Userscreen [confirm]

swaps to the user world for a look at the screen. Control is returned to CoPilot automatically after 20 seconds or by typing the **ABORT** key earlier; it does not return until the **ABORT** key is let up.

24.3.2.5 Low-level facilities

Additional commands allow you to examine (and modify) what is going on in the underlying system.

Fine point: When a space is first mapped as a data space, Pilot arranges things so that an attempt to read it by CoPilot before it is swapped in will show data left in backing store from a previous mapping, rather than the expected zeros.

ATtach Symbols [globalframe, filename]

attaches the **globalframe** to **filename**. **ATtach Symbols** is useful for allowing you to bring in additional symbols for debugging purposes when you do not have the correct object file. The default extension for **filename** is **.bcd**.

Warning: This command overrides version checking of symbol tables and should be used with caution; it may cause CoPilot to display incorrect values.

Note: Only compiler output object files for program modules can be attached; neither interfaces nor symbols files may be attached.

Display Eval-stack

displays the contents of the Mesa evaluation stack (in octal), which is useful for low-level debugging or for displaying the (un-named) return values of a procedure that has been broken at its exit point. This command is most useful at octal breakpoints because the eval stack is empty between most statements.

Octal Clear break [globalframe, bytepc]

is the converse of **Octal Set break** (these octal commands are low-level debugging aids for system maintainers who must diagnose the higher-level debugging aids and system).

Octal Read [address, n]

displays the *n* (decimal) locations starting at **address**. An **address** in the first 65K is interpreted as an absolute (virtual) address if and only if it has a leading zero; it is treated as MDS-relative otherwise.

Octal Set break [globalframe, bytepc]

sets a breakpoint at the byte offset **bytepc** in the code segment of the frame **globalframe**.

Octal Write [address, rhs]

stores **rhs** (octal) into the location **address**; the default for **rhs** is the current contents of **address**. **address** is treated the same as in **Octal Read**.

ReMote debuggee [host] [confirm]

converts CoPilot into a teledebugger. **host** is the name or net address of the client. (A net address has the form *netNumber#hostNumber#* where both numbers are octal, no "B" appended.) An empty **host** means to quit teledebugging. Pressing **ABORT** while waiting for the client will also abort teledebugging. Ending teledebugging in either of these ways causes CoPilot to start a new session without a client; the message **Invalid Swap Reason: Context Invalid** will be displayed in the new log, and the existing log is reset. CoPilot reverts to a local debugger for its next session.

After communications have been established, CoPilot starts a new session, losing all information about the previous debuggee. Immediately after receiving the **ReMote Debuggee** command and whenever CoPilot is waiting for the remote machine (e.g., for a breakpoint), it displays: **Waiting for client....** This is followed by the message **Client responds** when communications are re-established. Teledebugging may be terminated by the **ABORT** key; this is the only way to abort teledebugging while the **Waiting for client...** message is displayed. At other times, teledebugging may be aborted by issuing the **ReMote Debuggee** command with no host. If CoPilot is booted with the **W** switch, it immediately begins teledebugging, instead of completing the normal installation process. After communications have been established, the Debugger moves maplog and other information into its own memory and flushes it from the client; thus that client may not be subsequently be debugged by any other debugger until it is re-booted.

If a **Domain** and **Organization** have been specified in your user profile (either through Domain and Organization items in your **[System] User.cm** section, or with the Profile

Tool) they will be used to qualify any unqualified or partially qualified host names. Otherwise you will have to supply fully qualified host names for any remote clients you wish to debug.

For example, if the [System] section of your `User.cm` contained

```
Domain: OSBU North
Organization: Xerox
```

you could specify the ReMote debuggee `Thisbe:OSBU North:Xerox` as ReMote debuggee `Thisbe`.

Worry on [confirm]

sets breakpoints such that no local frames will be allocated when a breakpoint is taken. This is typically only of interest when debugging the operating system itself. As a side-effect, all conditional breakpoints will be temporarily made unconditional. After taking a worry mode breakpoint, all of the debugger commands are allowed, with the exception of `Start`, `Quit`, and calling procedures with the interpreter. Note: the Perf tools set worry mode breakpoints.

Warning: In the current version of Pilot, `Worry` should be turned on only when all breakpoints are in code that does not generate page faults.

Worry off [confirm]

turns off worry mode (this is the default state upon starting the debugger).

-- [comment]

inserts a comment into the debugger's log file. Input is ignored after the dashes until `RETURN` is typed.

24.3.3 The Debugger interpreter

CoPilot contains an interpreter that handles a subset of the Mesa language; it is useful for common operations such as assignments, dereferencing, procedure calls, indexing, field access, addressing, displaying variables and `TYPES`, and simple type conversion. It is a powerful extension to the debugger command language, as it allows you to more closely specify variables while debugging, thus giving you more complete information with fewer keystrokes.

Only a specific subset of the Mesa language is acceptable to the interpreter (see the end of this chapter for details on the grammar). Several specialized notations (abbreviations) have been introduced in the interpreter grammar; these are valid only for debugging purposes and are not part of the Mesa language. The interpreter operates much like the Compiler: strict target typing is performed on assignments and procedure calls.

24.3.3.1 Statement syntax

Typing `SPACE` to the command processor enables interpreter mode; the limited command processors of `Display Stack` and `Display Process` also permit a space. At this point

the debugger is ready to interpret any expression that is valid in the (debugger) grammar. The ? interpreter command may be invoked by either the ? item in the CoPilot menu, or the CLIENT1 key at any time.

Multiple statements are separated by semicolons; the last statement on a line should be followed by RETURN. If the statement is a simple expression (not an assignment), the result is displayed after evaluation.

For example, to perform an assignment and print the result in one command, you would type:

```
foo ← exp; foo
```

24.3.3.2 Loopholes

A more concise LOOPHOLE notation has been introduced to make it easy to display arbitrary data in any format. The character % may be used instead of LOOPHOLE[*exp*, *type*], with the expression on the left of the %, and the *type* on the right. However, % is not a valid LeftSide; all *type* expressions involving % must be enclosed in parentheses.

The following expressions are equivalent to the interpreter:

```
foo % (short red Foo) and LOOPHOLE[foo, short red Foo]  
(p % (LONG POINTER TO Object))↑ and LOOPHOLE[p, LONG POINTER TO Object]↑
```

The first pair will loophole the type of the variable *foo* to be a *short red Foo* and display its value. The second pair will loophole *p* to be a *LONG POINTER TO Object* and dereference it. *foo* % is a shorthand notation for *foo* % UNSPECIFIED.

A number may be loopholed into PROCEDURE, SIGNAL, or an ERROR. If it is valid, the debugger will display the procedure (or signal's) name, module and global frame. If a signal/error is the same as the uncaught signal that trapped to the debugger, CoPilot also displays the parameters.

24.3.3.3 Subscripting

There are two types of interval notation acceptable to the interpreter; the closed, open, and half -pen interval notation accepted by the Compiler and a shorthand version that uses !. The notation [*a* . . *b*] means start at index *a* and end at index *b*. The notation [*a* ! *b*] means start at index *a* and end at index (*a+b-1*).

The following expressions all display the contents of MDS-relative memory locations 1104B through 1107B:

```
MEMORY[1104 . . 1107]  
MEMORY[1104 . . 1108)  
MEMORY(1103 . . 1107]  
MEMORY(1103 . . 1108)  
MEMORY[1104 ! 4]
```

Note that the interval notation is only valid for display purposes and therefore is not allowed as a **LeftSide** or inside other expressions.

24.3.3.4 Explicit qualification vs qualification in the current context

To improve the performance of the interpreter, the \$ notation has been introduced to distinguish between qualification in the current context and explicit qualification. The character \$ indicates that the name on the left is a module name or frame in which to look up the identifier or **TYPE** on the right. If a module cannot be found, it uses the name as a file (usually a definitions file).

For example, **FSP\$TheHeap** means look in the module **FSP** to find the value of the variable **TheHeap**. In dealing with variant records, be sure to specify the variant part of the record before the record name itself (e.g., **foo % (short red FooDefs\$FOO)**, not **foo % (FooDefs\$short red Foo)**).

24.3.3.5 Type expressions

The notation **@type** may be used as shorthand to construct a **POINTER TO type**. This notation is used for constructing types in **LOOPHOLES** (ie., **@foo** will give you the type **POINTER TO foo**). There is no special shorthand to construct **LONG POINTER TO type**; however, **LONG @type** is legal.

24.3.3.6 Radix conversion

The notation **expression?** prints the value of the expression in several formats, including octal, decimal, and hex. Radix conversion between octal and decimal can be forced using the loophole construct; for example, **exp%(CARDINAL)** will force octal output and **exp%(INTEGER)** will force decimal. Output radix may also be controlled by the **CoPilot Options** window discussed in the **Output conventions** sub-subsection previously mentioned.

24.3.3.7 Arithmetic expressions

Target typing is applied to arithmetic expressions. In complex expressions, atoms that change the target type must occur first. For example:

```
(POINTER + offset)↑ -- correct
(offset + POINTER)↑ -- error message
LONG[400B] * 400B -- 200000B
400B * LONG[400B] -- overflow
```

24.3.3.8 Procedure calls

It is often useful to call procedures from a breakpoint or after getting an uncaught signal; this is generally done in the interpreter with the same syntax as in Mesa. CoPilot is able to invoke any procedure that is imported into the current module context; complications arise when you wish to call a procedure that is not imported. However, the \$ notation may be used to solve most of them.

CoPilot can only call procedures in modules for which it has complete symbols; this can be somewhat confusing since the debugger "knows" a little about the procedures imported into a module it has symbols for. To determine whether CoPilot has symbols for a procedure and where it is implemented (a more useful feature), simply type the procedure name to the interpreter. For example, typing either **Process.SetPriority** or **SetPriority** to the interpreter (while inside a module that imports it) will cause the debugger to display the following

```
SetPriority = PROCEDURE [5461B] (in module Processes, G: 11644B)
```

when symbols for **Processes** are not available. Reinterpreting **SetPriority** after retrieving the object file for **Processes** gives the following result:

```
SetPriority = PROCEDURE SetPriority (in module Processes, G: 11644B).
```

The notation **Process.SetPriority** means the same to the interpreter as to the Mesa compiler; **SetPriority** is a procedure imported through the **Process** interface.

Since **SetPriority** is imported in this example, you could, for example, *call* it (nicknamed *interpret call* for historical reasons) by typing **SetPriority[1]**. To call **Process.Abort**, which is not imported, the notation **Processes\$Abort[processId]** or **nnnnnB\$Abort[processId]** (**nnnnnB** is the global frame of **Processes**) works. If you are lacking a variable of type **PROCESS**, **Processes\$Abort[20B%]** works; it loopholes the process ID number 20B into an **UNSPECIFIED**. (The trailing **%** notation is a very easy method for constructing pointers; e.g., **123456B%** is easier to type in a procedure call than **LOOPHOLE[123456B, POINTER]**.)

24.3.3.9 Sample expressions

Here are some sample expressions that combine several of the rules into useful combinations:

If you were interested in seeing which procedure is associated with the third keyword of the menu belonging to a particular window called **myWindow**, you would type:

```
> myWindow.menu.array[3].proc
```

which might produce the following output:

```
CreateWindow (PROCEDURE in WEWindows, G: 120134B).
```

The basic arithmetic operations are provided by the interpreter (with the same precedence rules as followed by the Mesa compiler).

```
> 3+4 MOD 2 ; (3+4) MOD 2
```

would produce the following output:

```
3  
1.
```

A typical sequence of expressions one might use to initialize a record containing a pointer to an array of Foos and display some of them would be:

```
> rec.array ← FSP$AllocateHeapNode[n*SIZE[ FooDefs$Foo ]];
> InitArray[rec.array]; rec.array[first..last].
```

The following command would display `rec` in octal:

```
>Octal Read: @rec, n: SIZE[Rec]
```

To find out what type a `HeapImpl Handle` pointed to:

```
> HeapImpl$Handle
Handle: PRIVATE TYPE = LONG POINTER TO Data;
> HeapImpl$data
Data: PRIVATE TYPE = RECORD
```

or to find out what parameters a `SchemaDefs.PvPrint` took:

```
> SchemaDefs.PvPrint
Pvprint: PUBLIC TYPE = PROCEDURE[lschema: Lschema, posn: Posnarea,
oispfh: OISPFH]
```

24.4 Signal and error messages

The following messages are generated by the debugger. Wherever possible, there is also an explanation of what might have caused the problem and what you can do about it.

24.4.1 Entering the Debugger

The following messages are feedback from CoPilot informing you why the debugger was entered.

***** Processing VM Map *****

Pilot maintains a log of virtual page to file page mappings so that CoPilot can read and write the client's virtual memory without regard to whether the data is currently in real memory. In order to keep mapping operations simple, Pilot logs changes serially. When the log fills up, CoPilot is invoked to empty it. CoPilot will issue the `Processing VM Map` message and return to the client after a few seconds without requiring or allowing any user intervention.

***** Interrupt *****

Appears at the top of the `Debug.log` window after you have entered the debugger via interrupt mode (`SHIFT-ABORT(CALLDEBUG)` has been held down).

***** uncaught SIGNAL SoS (in MayDay)**

The user program has raised a `SIGNAL (ERROR)` which no one dynamically nested above the `SIGNAL` invocation was prepared to catch. The debugger prints the name of the `SIGNAL`, lists its parameters (if any), creates a new instance of the debugger, and gives

control to the command processor. At this point you may, for example, display the stack to see who raised the uncaught **SIGNAL**.

If the semantics of the situation permit, you may proceed execution at the point of the **SIGNAL**'s invocation by issuing a **Proceed** command. Programs often allow themselves to be aborted by CoPilot's **Quit** command; it simply raises the aborted **ERROR** on the client side. If no client catches this error, you end up in the dynamically enclosing instance of the debugger. If the **SIGNAL** actually was an **ERROR** and you elect to **Proceed**, you get a **ResumeError**.

Note: If CoPilot does not have access to the required symbol tables, the information is printed in octal. For standard Mesa software, listings which decode these numbers are available.

The remaining error messages in this section are not fatal, but you should be suspicious of the state of the client.world when they are given.

CoPilot inloaded twice! Click to boot.

CoPilot was not exited cleanly in the previous session. The most common ways to leave CoPilot cleanly are with the **Boot from** menu in the Herald window, or the **Quit** or **Kill** commands. Pressing any mouse button will re-install CoPilot; any debugging of the new client is impossible.

breakpoint not found!

You have swapped to the debugger when the breakpoint information (frame, pc, etc.) cannot be found (check the code for your program).

Eval stack not empty!

The warning is printed if the debugger is entered via either an interrupt or breakpoint with variables still on the evaluation stack; this indicates that the current value of some variables may not be in main memory, where the interpreter normally looks. Exceptions to this are entry and exit breaks; the debugger has enough information to decode the argument records that are on the stack in this case (if the appropriate symbol tables are available).

***** Invalid Swap Reason - Context Invalid *****

CoPilot has been entered with a damaged (core-clobber) or missing client (teledebugging ended). No debugging is possible in this state; attempts to do so receive warning messages. However, other cascade features continue to work normally.

Eval-stack is wrong

The evaluation stack had an incorrect number of arguments on it at the time a **Runtime.CallDebugger** was made. In this event, CoPilot works normally, but any attempt to return to the client will probably cause a stack error.

*****Invalid Load State*****

CoPilot has been entered without the client's load state available. The load state is used by the debugger to translate octal information (e.g., module names) into English for the user; without the load state only octal debugging features are available.

24.4.2 Symbol lookup**xxx is compiled for an incompatible version of Mesa!**

A wrong version of the Compiler was used; e.g., this is an old Mesa object file.

xxx cannot be acquired with read access!

The file named **xxx** exists, but cannot be read.

xxx is read protected!

The file **xxx** has been left read-protected by the File Tool or some other subsystem. Refetching the file will remove the error.

xxx not found!

The variable or file named **xxx** cannot be found.

!File: xxx

The file named **xxx** cannot be found.

nnnnnB not started!

The global frame **nnnnnB** has not yet been started. Any variables looked up will be uninitialized.

xxx not bound!

The imported variable **xxx** is not exported by anyone.

!xxx: --compressed symbols--

The symbol file is compressed.

xxx has incorrect version!

The symbol file has an incorrect version stamp.

!Tree for xxx not in symbol table

A multiword constant in your code wasn't copied into the symbol table. Look in the source file to find the value.

xxx is missing some pages [base, pages+extra]

The **bcd** or **symbols** file **xxx** is not as long as CoPilot expected it to be. **base** is the page that CoPilot believes the symbols start on. **pages** is the length of the symbol table and **extra** is the length of the fine-grain table. Try refetching **xxx** to solve this problem.

Use Interface.importedVariable, not Interface\$importedVariable

The debugger cannot find imported variables from an interface file (the "\$" notation). The "." notation will tell it to use the interface record (if found) available in the current context.

24.4.3 Unrecognized structures

```
!Can't find links from frame: nnnnnB
!Invalid global frame
xxx not a frame!
xxx has a NULL returnlink!
xxx has a clobbered accesslink!
xxx is a clobbered frame!
xxx is an invalid PROCESS !
xxx is an invalid global frame!
xxx is an invalid image file!
xxx is not a valid frame!
```

The structure in question appears to be clobbered (invalid in some way).

24.4.4 Command execution errors

... aborted

Execution of the current command has been aborted (**ABORT** has been typed).

Can't use <module> of <time> instead of version created <time>

This message is printed if the creation date in the source, object, or symbols file on your disk is different than the corresponding date recorded by the Compiler or Binder.

!Resetting symbol table

This warning is displayed before the debugger's scratch symbol table overflows. The debugger's performance decreases somewhat until the symbol table is refilled.

!Number

An invalid number has been typed.

xxx is a definitions file!

You have tried to set a break in a definitions file.

xxx not a REAL!

xxx is not a valid representation of a real number.

xxx not implemented!

Feature xxx is not implemented.

!Invalid Address [nnnnB]

During the execution of a command, CoPilot attempted to read or write location nnnnB, which was not mapped. I/O pages and pages belonging to the germ appear unmapped to CoPilot.

!Write protected [nnnnB]

During the execution of a command, CoPilot attempted to write location nnnnB, which was write-protected.

! unknown file problem! Your directory probably needs scavenging.

Something is wrong with your directory.

!Command not allowed

Execution of the current command is not allowed, since the state of the user core image appears to be invalid.

!MDS exhausted [n]

The debugger has run out of memory.

24.4.5 Breakpoints

When using the menu break commands, each of the following errors will cause the screen to flash after posting a message in the Herald window.

Multiple instances; Use Display Stack, Source to load window.

You have tried to set a break when multiple instances of the module exist; explicitly setting the context for the source window will permit the break to be set.

Can't dereference or access array to test condition!

You have specified a condition that requires dereferencing or an array indexing to test; the runtime is unable to evaluate conditions that complex.

too many conditional breaks!

You have tried to set more conditional breaks than the system allows.

invalid relation!

You have specified an illegal relation expression for a condition.

user break block not found!

You have tried to free a conditional breakpoint when the conditional breakpoint information cannot be found (probably a core clobber).

variable is larger than a word!

You have tried to set a condition that uses a multiword value.

rhs on stack not allowed!

You have tried to set a condition where the right side of the relational expression is on the stack. Only the left side can be on the eval stack. This can only happen on entry and exit breakpoints.

Can't check condition not in MDS!

You tried to use a long address as part of a condition.

conditions not checked in Worry mode!

You have attached a condition while in worry mode. This is a warning only.

no exchangable code found!

The debugger has tried several consecutive instructions and has not found an opcode on which a breakpoint is allowed. The code has probably been clobbered.

no breaks have been set!

You did a **L**ist **B**reaks when there weren't any.

symboltable missing!

The debugger is trying to manipulate a breakpoint for which there is no **symboltable** and it is not prepared to handle the situation.

not allowed in INLINE!

You have tried to set a breakpoint in an **INLINE** procedure.

already set!

You have already set a breakpoint there.

does not return!

An attempt was made to set an exit breakpoint on a procedure in which the return statement is not in the correct location (check the code for your program). This occurs

most often in procedures that end with **ERROR** or a loop that does not terminate; a code clobber is also possible.

! Patch table full

The maximum number of breakpoints (50) allowed by Pilot has been reached.

??

Unknown error.

24.4.6 Displaying the stack

No previous frame!

The end of the call stack has been reached.

No symbol table for nnnnnnB

The symbol table file corresponding to the frame **nnnnnnB** is missing; any attempt to symbolically reference variables in this module will fail. (In general, this message is a warning.)

Cross jumped!

The bcd was compiled with the cross-jumping switch turned on. The source line displayed may not be what you expect.

Pc not in any procedure!

The debugger was unable to find a procedure or mainline code that matched the current pc. This is probably due to a clobber.

24.4.7 Interpreter

! x is an invalid character

The character **x** typed to the interpreter is illegal.

! Syntax error at [n]

There was a syntax error at location **n** in the expression given the interpreter.

! Parse error at [n]

There was a error at location **n** parsing the expression given the interpreter.

The following errors may have the offending identifier preceding the message:

can't call a SIGNAL!
can't call an ERROR!
can't call an INLINE!

You tried to call a **SIGNAL**, **ERROR**, or **INLINE PROCEDURE**.

can't lengthen!

The interpreter needed to lengthen a part of an expression while trying to evaluate it.

can't make a constructor!

Use field by field assignments. You gave the interpreter an expression using [] that looks like a constructor.

double word array index!

The index for an array must be a single word.

has an invalid address!

The expression to the right of the @ is not word-aligned.

is an invalid number!

This is probably a type mismatch.

is an invalid pointer!

This is probably a type mismatch.

invalid subrange!

This is probably a type mismatch.

pointer fault!

You tried to dereference **NIL**.

xxx is a constant array. Look at source code for value.

An operation on a constant array is too complicated to perform. The operation can be done by hand, however, by looking at the constant value in the source.

xxx is not an array!

You have tried to use **xxx** as an array.

is not a valid control link!

The procedure or signal in your expression has an illegal value.

is not a relative pointer!

In the expression **base[rel]**, **rel** wasn't a RELATIVE POINTER.

is not a type!

The identifier used in a type expression was not a type.

is not a unique field selector!

The field selector occurs more than once in the computed or overlaid variant.

is not a valid field selector!

The identifier given for a field selector is not in the record. You may lack the symbols for the record declaration on your disk.

overflow!

Overflow occurred while doing arithmetic. Perhaps you need a **LONG** in the expression.

relations not implemented!

a = b is not allowed.

size mismatch!

You tried to assign or loophole two things of different sizes. Loopholing pointers is a useful trick for records of different sizes.

too many arguments for stack!

You can only call procedures that take 11 or fewer words of arguments.

has incorrect type!

Type mismatch.

unknown variant!

The interpreter found a garbage tag field.

Won't dump that much memory!

You tried to print more than 64K with the **MEMORY** construct.

not permitted in worry mode!

You can't call procedures in worry mode.

is the wrong base!

In the expression **base[rel]**, the type of **base** is not what **rel** expects.

has the wrong number of arguments!

The arguments to a procedure call are wrong.

used incorrectly with []!

You probably tried to use [] as a type constructor.

illegal indexing operation

You tried to index something that wasn't an array or sequence.

xxx\$ is ambiguous; use frame \$!

There is either more than one instance of **xxx** instantiated, or the code for **xxx** is packed with another module.

BUG: !NotAnArray

This is a bug in CoPilot. It means that the interpreter didn't recognize an error condition. You should submit an AR if this happens and you have a repeatable test case.

BUG: !NotHere

This is the same as **BUG: !NotAnArray**, but a different internal error.

24.5 User.cm

The **User.cm** is used by CoPilot only during installation.

[Debugger]

Boot: volume

Tells CoPilot what volume to boot (and with what switches) when the installation process is finished; CoPilot does a physical volume boot if this line is absent.

DebugLog:

Description of the box for the **Debug.log** window.

Example:

[Debugger]

Boot: Tajo/%2

DebugLog: [x: 0, y: 0, w: 512, h: 412]

[System]

User: Smith

Registry: PA

Domain: OSBU North

```
Organization: Xerox
FileWindow: [x: 512, y: 0, w: 512, h: 512]
```

24.6 CoPilot interpreter grammar

StatementList	::=	Statement StatementList; StatementList; Statement
Statement	::=	LeftSide Interval LeftSide ← Expression MEMORY Interval Expression Expression ?
LeftSide	::=	identifier (Expression) LeftSide Qualifier identifier \$ identifier number \$ identifier MEMORY [Expression] LOOPHOLE [Expression] LOOPHOLE [Expression , TypeExpression]
Qualifier	::=	.identifier [ExpressionList]
Interval	::=	[Bounds] [Bounds) (Bounds] (Bounds) [Expression ! Expression]
Bounds	::=	Expression .. Expression
Expression	::=	Sum
Sum	::=	Product Sum AddOp Product
AddOp	::=	+ -
Product	::=	Factor Product MultOp Factor
MultOp	::=	* / MOD
Factor	::=	Primary Primary
Primary	::=	Literal LeftSide @ LeftSide BuiltinCall Primary % Primary % (TypeExpression)
Literal	::=	number character string
BuiltinCall	::=	NIL NIL [TypeExpression] PrefixOp [ExpressionList] TypeOp [TypeExpression]
PrefixOp	::=	ABS BASE LENGTH LONG MAX MIN
ExpressionList	::=	empty Expression ExpressionList, Expression

TypeOp	:: = SIZE
TypeExpression	:: = identifier TypelIdentifier TypeConstructor
TypelIdentifier	:: = BOOLEAN INTEGER CARDINAL WORD REAL CHARACTER STRING UNSPECIFIED PROC PROCEDURE SIGNAL ERROR identifier identifier identifier TypelIdentifier identifier . identifier identifier \$ identifier
TypeConstructor	:: = LONG TypeExpression @ TypeExpression POINTER TO TypeExpression

24.7 CoPilot summary

ASci

Read [address, count]
Display [address, count]

ATtach

Condition [number, condition]
Keystrokes [number, command]
Symbols [globalframe, filename]

Break

All
Entries [module/frame]
Xits [module/frame]
Entry [procedure]
Xit [procedure]

Clear

All
Breaks
Entries [module/frame]
Traces
Xits [module/frame]
Break [number]
Condition [number]
Entry
Break [procedure]
Trace [procedure]
Keystrokes [number]
Xit
Break [procedure]
Trace [procedure]

CUrrent context

Display
Break [number]
Configuration
Eval-stack
Frame [address] (g,j,l,n,p,q,r,s,v)

GlobalFrameTable
Module [*module*]
Process [*process*] (*l,n,p,q,r,s*)
Queue [*identifier*] (*l,n,p,q,r,s*)

Display
ReadyList (*l,n,p,q,r,s*)
Stack (*g,j,l,n,p,q,r,s,v*)

Find variable [*identifier*]

Kill session [*confirm*]

List
Breaks [*confirm*]
Configurations
Processes

Octal
Clear break [*globalframe, bytepc*]
Read [*address, number*]
Set break [*globalframe, bytepc*]
Write [*address, value*]

Proceed [*confirm*]

Quit [*confirm*]

ReSet context [*confirm*]

ReMote debugger [*host*] [*confirm*]

SEt
Configuration [*config*]
Module context [*module/frame*]
Octal context [*address*]
Process context [*process*]
Root configuration [*config*]

STart [*address*] [*confirm*]

Trace
All
Entries [*module/frame*]
Xits [*module/frame*]
Entry [*procedure*]
Stack
Xit [*procedure*]

Userscreen [*confirm*]

Worry
off [*confirm*]
on [*confirm*]



DebugHeap

The DebugHeap Tool allows you to interrogate and analyze Pilot node storage usage and find storage leaks. It understands the structure of Pilot heaps and zones. See the *Pilot Programmer's Manual* for a complete definition of heaps and zones.

Heaps are used to allocate small objects. They can be thought of as retail storage allocators, while the space machinery can be thought of as a wholesale storage allocator. Heaps allocate nodes from segments, which are multi-page blocks of memory allocated from the space machinery. Heaps can allocate either variable-length objects or fixed-length objects. Heaps that allocate variable-length objects use zones to keep track of allocation within a segment but allocate rather large objects directly from the space machinery.

Pilot heaps optionally allow owner checking. When owner checking is enabled, an extra word is allocated with each node; this word contains the global frame address of the module that requested the allocation. Other heaps may allocate additional information for debugging purposes. DebugHeaps allow you to specify how many such additional "client words" were allocated with each object and use them to filter which nodes are displayed.

25.1 Files

Retrieve **DebugHeap.bcd** from Release directory.

25.2 User interface

The DebugHeap tool interacts through a form subwindow, a file subwindow, and a menu:

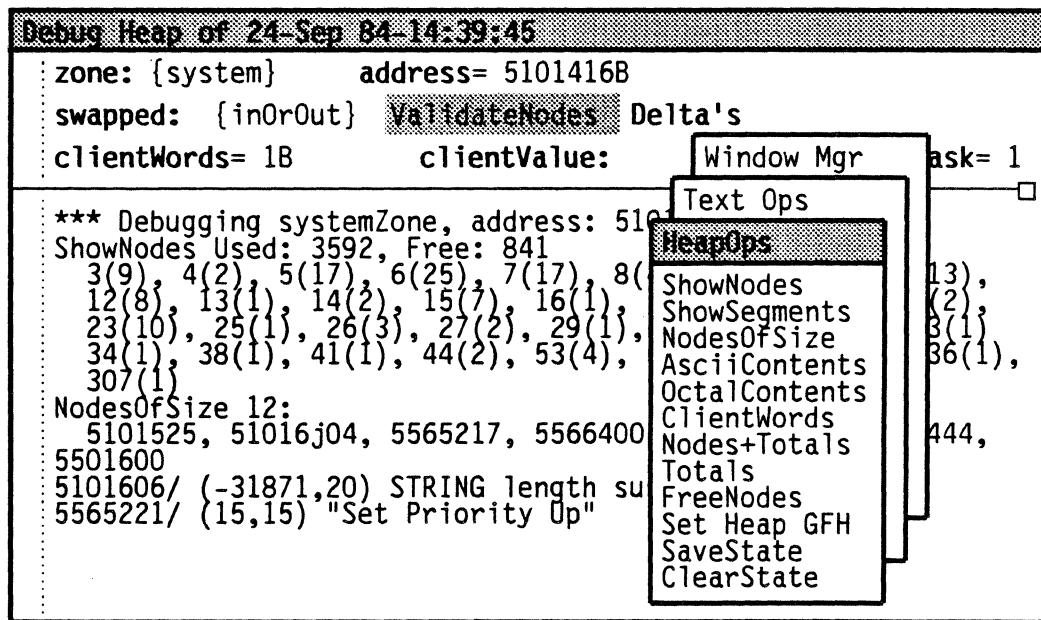


Figure 25.1: DebugHeap tool window

25.2.1 Form subwindow

The fields in the DebugHeap Tool form subwindow are as follows:

zone: is an enumerated item that specifies whether to look at one of the Pilot built-in heaps or a private heap or zone. The zone options are as follows:

systemMDS processes the built-in MDS heap.

system processes the built-in heap.

zone processes a private zone specified by **address**.

heap processes a private heap specified by **address**.

heapMDS processes a private MDS heap specified by **address**.

address= is a long number used to specify the address of the heap or zone of interest.

swapped: is an enumerated item that specifies whether to restrict DebugHeap to examining nodes that are swapped in, swapped out, or either.

validateNodes is a Boolean telling DebugHeap to check that values supplied as node addresses are really nodes. This Boolean is also used by the string printing routines to check for invalid or suspicious strings.

delta's	is a Boolean used to indicate processing of the heap or zone relative to the saved state (see the SaveState and ClearState menu commands below).
clientWords=	indicates the number of words in each node that are being used for debugging purposes (e.g., one word is used for normal Pilot owner checking).
clientValue:	is a string form item used to specify a filtering value for processing nodes. If the heap has Pilot owner checking, specifying a global frame will cause DebugHeap to display only those nodes that were allocated by the module. Multiple values can be supplied, separated by commas and/or spaces, and a range may be specified by two values separated by "..".
mask	is a number (usually specified in octal). If clientWords=1 and any client values are specified in the clientValue field, the value of mask (if any) is bit-anded with the client words in each node before comparing with the specified client values.

25.2.2 DebugHeap menu

The DebugHeap menu is attached to the DebugHeap Tool window. The commands are listed below:

ShowNodes	tabulates and displays the current state of the selected heap or zone. The number of free and used words in the entire heap or zone are displayed, as are the size and number of all used nodes.
ShowSegments	displays all segments that make up the selected heap or zone, and notes their sizes.
NodesOfSize	displays the address of nodes of the specified size within the selected heap or zone. The current selection is used to indicate the size. The heap manager's overhead (currently one word) is included in the size.
AsciiContents	displays the contents of the specified node as an Ascii string. The current selection is used to indicate the the node address. The Boolean validateNodes indicates whether to check that the address is really a node and to perform a check of valid strings. Multiple nodes may be printed by selecting multiple node addresses separated by spaces and/or commas. (e.g., the output of NodesOfSize is valid input to this command).
OctalContents	displays the contents of the specified node as <i>n</i> octal words. The current selection is used to indicate the the node address. The Boolean validateNodes indicates whether to check that the address is really a node. Multiple nodes may be printed by selecting multiple node addresses separated by spaces and/or commas. (e.g., the output of NodesOfSize is valid input to this command).

ClientWords	displays the contents of the client-words' portion of the specified node in octal. The current selection is used to indicate the the node address.
Nodes&Totals	displays the node address, length, and module for each node in use in the current heap or zone. If the clientValue field is empty, free nodes are also displayed; otherwise only nodes whose client words match clientValue are displayed. The totals by module are displayed following the display of all nodes. This command only works if clientWords=1 .
Totals	acts like Nodes&Totals , but displays only the totals by module.
FreeNodes	displays the address and size of each free node in the current heap or zone.
SetHeap GFH	manually sets the global frame for the built-in Pilot heaps. DebugHeap always attempts to find this value automatically. This command allows you to override the default.
SaveState	processes the current zone and saves the size and addresses of all allocated nodes. Setting the Boolean delta's tells DebugHeap to display only the differences between the saved state and the current zone.
ClearState	takes all of the state saved as a result of the last SaveState and discards it.

25.3 Example

To find a suspected leak:

1. Boot the client with the **heapOwnerChecking** switch (see **PilotSwitches** interface in the *XDE User's Guide* for the current value).
2. Get the client to a stable state (e.g., deactivate all tools in Tajo); then go to the debugger.
3. Run **DebugHeap** in the **SimpleExec**.
4. Set the **zone:** and possibly the **address:** fields so that you are investigating the particular zone of interest. You will either be interested in the system zone or a private heap. To examine a private heap, for example, select the **heap** parameter in the **zone:** field and put the value of your **UNCOUNTED ZONE** variable in the **address:** field.
5. Do a **SaveState** and proceed to the client.
6. Repeat the suspicious action that might have resulted in a space leak; then try to get the client back to the state that you had originally (e.g., deactivate tools in Tajo).

7. Interrupt to the debugger and turn **Deltas** on. While **Deltas** is on, most commands show the difference between the new state and the saved state.

If you invoke **Totals**, anything that shows up is suspicious (see **Totals**). **Totals** will tell you what the modules were that allocated the suspicious nodes.

8. Now that you have a list of modules that are suspect, put the global frame handles of the modules in the **clientValue:** field.
9. Invoke **Nodes&Totals**. Investigate each node or a list of nodes using the **OctalContents** or **AsciiContents** commands. The size of the node is also a good hint as to what was allocated. Subtract one (two, if you booted with the **heapOwnerChecking** switch) from the size of the node and try to figure out where in the module you allocated such a node.

Repeat the above steps for every heap and zone where you suspect a leak.



IncludeChecker

The IncludeChecker is a program that examines a collection of local or remote text and object files for consistency and produces an output listing that gives a compile, bind, and package order for the files in the collection. For each object file, a list of all the object files that it includes and a list of the object files that include it is also produced. Any inconsistencies (described below) are flagged in this listing by an asterisk. As an option, the IncludeChecker will also generate a compile, bind, and package command in `Line.cm` that is its best guess as to the way to make the files consistent.

The IncludeChecker determines that an inconsistency exists among the input files if either:

1. An object file includes another object file with a version that is different from any version of the included file that was found. This might happen, for example, if the included file had been recompiled.
2. A text file is newer than the corresponding object file. This could happen if the text had been edited, or if the text had been retrieved from a remote file server. The IncludeChecker compares the creation date of the text file against the creation date recorded in the corresponding object file.

When determining consistency, the IncludeChecker tries to deal gracefully with files found in multiple locations and versions. It attempts to match these files with the corresponding object and text files (possibly on other directories). It also tries to match included files against versions of those files that it has found.

26.1 Files

Retrieve `IncludeChecker.bcd` from the Release directory.

26.2 User interface

The IncludeChecker runs either as a tool or in the Executive. It lists file names in the compilation order, and the consistent compilation command, by inclusion depth, with the deepest files included first. Within that constraint, definitions modules are printed before

program modules. In general, then, the lowest-level definition modules appear first, while the highest-level program modules appear last.

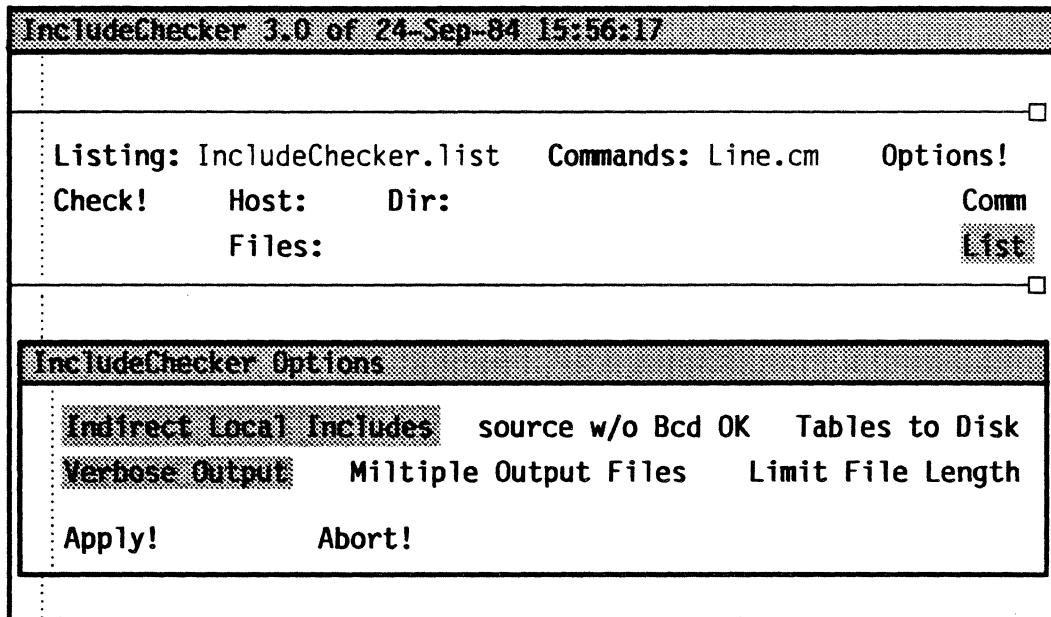


Figure 26.1: IncludeChecker tool window

The Includes list indicates the host and directory for both text and object files. It also notes, when multiple copies of a file are found, the different versions and their locations. If an object file was derived from a version of the text that was never found, there will be one entry for the object file and one entry for each version of the text that was found (since in general, these can be in different locations). Obtaining this list (with the /i OperatingSwitch, which is the default) is strongly recommended because it can explain, for example, why the IncludeChecker wanted to recompile some file. This means that the /s OperatingSwitch should not be used.

Note: It is also a good idea to inspect **Line.cm** before executing it, since the IncludeChecker's idea of what should be recompiled and rebound may not be the same as yours. Because the compiler does not give enough information to completely construct the packaging command, the packaging command is incomplete and must be edited by hand.

26.2.1 Tool interface

The IncludeChecker communicates through a message subwindow, a form subwindow, and a file subwindow. The fields in the form subwindow are as follows:

- | | |
|---------------|--|
| Check! | starts the IncludeChecker. |
| Host: | is the name of the host to be used for remote files. |
| Dir: | is the default remote directory. |
| Files: | are the files to be checked by the IncludeChecker. |

Listing:	is the name of the outputfile the IncludeChecker generates that shows the dependencies of the files. The outputfile requires a substantial amount of disk space. The default extension is .list .
Commands:	is the file where the IncludeChecker writes the rebuild commands. The default extension is .cm .
Options!	brings up a separate Options window.
Command:	causes a command file to be written to the file named by the Commands: field.
Pause	causes a /p to be appended to the compile command in rebuild command.
List	prints the includes and included-by relationships in the Listing: file. Default = TRUE .
Order	prints compilation order in the Listing: file. Default = TRUE .

The following switches are in the **Options!** window:

Indirect Local Includes	causes analysis of both directly and indirectly included files. Thus only the top-level bcd need be specified in the Files: item. Default = TRUE .
Source w/o Bcd OK	If there is a text file without the corresponding bcd, no error will be raised. Default = FALSE .
Tables To Disk	causes the IncludeChecker's internal data structure to be written to outputfile.data . This option is intended for future use. It is not needed by standard users of Mesa 11.0. Default = FALSE .
Verbose Output	gives complete file list. Default = TRUE .
Multiple Output Files	writes output to outputfile.includes and outputfile.includedBy . Default = FALSE .
Limit File Length	limits file lengths to 100,000 bytes. Successive file names are outputfile.list2 , outputfile.list3 , etc. Default = FALSE .
Apply!	invokes options.
Abort!	resets to previous options.

26.2.2 Command line

The syntax for the command line is:

```

CommandLine ::= IncludeChecker [<OperationParameters>]
                  [<FileList>]

<OperationParameters> ::= <OutputFile>/<OperatingSwitches>
                         [<CommandList>]

<OperatingSwitches> ::= a | c | i | l | m | n | o | p | s | v | x
                         (See the section on Operating switches)

<CommandList> ::= {<Command>/c <Name>}^

<Command> ::= open | dir | commandfile

<FileList> ::= <FileName1 FileName2...>^

```

The <OperationParameters> and <FileList> components of the **CommandLine** are optional. In <CommandList>, the /c switch indicates to the **IncludeChecker** that the token before the /c is a command (e.g., **open**, **dir**, **commandfile**), not a **FileName**.

The **OutputFile** is the name of the file written. If no extension is given, .list is assumed. If no **OutputFile** is given at all, **IncludeChecker.list** is assumed. <FileList> is the list of file names specifying the text and .bcd files to be checked. It is not necessary to give an extension, since the **IncludeChecker** will look for any .mesa, .bcd, .config or .pack file with the specified name. (Consequently, don't specify both **Foo.bcd** and **Foo.mesa** on the command line, since **Foo** would be checked twice.)

In general, a **FileName** can be fully qualified by giving a host and directory; e.g., [**server**]<**Int**>**Pilot**>**Public**>**Heap.mesa**. It is possible to intermix remote and local files on the command line since the host name **ME** is interpreted to mean the machine running the **IncludeChecker**, so that [**ME**]**Space.bcd** refers to a file on the local disk. The initial setting for the global host name is **ME** and the global directory name is empty.

26.2.3 Operating switches

Each operating switch can be preceded by a - or ~ to reverse its meaning. The switches are:

- a Check all directly and indirectly included files on the local disk (the default).
- c "Consistency command": write a compile and bind command in **Line.cm** (-c is the default). In addition, list as comments any object files and text files not found that are needed for the compilation or binding.
- i Print both the includes and included-by relationships in the output file (the default).
- l Limit output file size to 100,000 bytes per output file. Successive file names are **outputfile.list2**, **outputfile.list3**, etc.

- m** Use multiple output files (-m is default). The compilation order is written on **source.outputfile**. The includes and included-by relations are written onto **outputfile.includes** and **outputfile.includedBy**, respectively.
- n** Don't list text files for compilation or rebinding that have no object file on the disk (-n is the default).
- o** Print a compilation order in the output file (the default); -o suppresses this listing.
- p** Place a /p after every change of inclusion depth (see below) in the consistency command (-p is the default). This will cause the Compiler or Binder to stop if errors are found while processing the files of that depth.
- s** Same as /c-i-o. This is used when only a consistent compilation command is needed. This switch is not recommended, since the includes/included-by list (produced by /i) is very helpful in determining why the IncludeChecker asked that particular files be recompiled or rebound (-s is the default).
- v** Verbose listing. This switch will produce feedback about all files checked even if errors are detected. /-v will produce feedback only on files that generate errors. (v is the default.)
- x** Just activate the tool and don't run in the Executive.

26.3 Examples

To check files on the local disk, just list them, e.g.:

```
>IncludeChecker Lex.list/cio LexiconDefs Lexicon LexiconClient
```

inspects the text and object files for the modules **LexiconDefs**, **Lexicon**, and **LexiconClient** for consistency. It also checks that these files are consistent with their included object files. **Lex.list** is the output file.

If you have a list of the text files for a program in a file, say, **ListOfFile.cm**, you can check these files with a command line of:

```
>IncludeChecker MyStuff.list/cio @ListOfFiles.cm@
```

MyStuff.list is the output file. Note: The Executive replaces **@File@** with the contents of **File** (see the Executive chapter).

To check all files on the current search path, use the following command line:

```
>IncludeChecker AllFiles.list/c
```

processes all **.bcd**, **.mesa**, **.config**, and **.pack** files on the current search path. **AllFiles.list** is the output file.

Remote files are checked by using a command line syntax much like that for FTP (see the FTP chapter). The **open** and **dir** commands specify a remote host and directory. The **/c**

switch associated with **open** and **dir** indicate to the IncludeChecker that the previous token is a command. The **/c** operating switch associated with the output file, **MyProgram.list**, instructs the IncludeChecker to write a compile and bind command in **Line.cm** (see the Operating switches section).

```
>IncludeChecker MyProgram.list/c open/c server dir/c  
WorkingDir>MyProgram @Source.MyProgram@ ...
```

To check all files on the remote directory [**server**]<**WholeDir**>, use the following command line:

```
>IncludeChecker WholeDir.list/c open/c server dir/c WholeDir
```

To run the IncludeChecker on a local directory named Temp and create a rebuild command:

```
>IncludeChecker AllOfTemp.list/c dir/c Temp
```

Note that giving the IncludeChecker an explicit local directory to check is somewhat faster than setting the search path to that local directory and using the command line:

```
>IncludeChecker AllOfTemp.list/c *.mesa
```

Specifying an explicit local directory avoids the Executive expansion of ***.mesa**, the parsing of a potentially very long command line, and the lookups for each **FileName F** (**F.mesa**, **F.bcd**, **F.config**, **F.pack**). Instead, the entire directory is enumerated; no unnecessary probes are done to determine if files exist.

To bring up the tool only, type either of the following commands to the Executive:

```
>IncludeChecker/x
```

```
>Run IncludeChecker.bcd
```

The output file by default is written on **IncludeChecker.list** and the command file is **Line.cm**. To direct the output file to **MyFile.list** and the command file to **MyCommand.cm** in the first example, type:

```
>IncludeChecker MyFile/c dir/c Temp commandFile/c MyCommand
```

26.4 User.cm

The following is a list of the `User.cm` fields used by the IncludeChecker:

[IncludeChecker]

CommandNameFromRoot:	Boolean item that, if <code>TRUE</code> , will cause the IncludeChecker to use <code><root>.cm</code> instead of <code>Line.cm</code> as the name of the <code>compile</code> , <code>bind</code> , and <code>package</code> command produced by running the IncludeChecker with <code>/c</code> . <code><root></code> is the output file name minus any extension.
DefaultSwitches	Operating switches to be used by the IncludeChecker. (See the Operating switches section.)



Lister

The Lister produces various listings of information in object files, such as dates of the definitions files used by an object file and a cross-reference listing of procedure calls within the object file.

27.1 Files

Retrieve **Lister.bcd** from the Release directory.

27.2 User interface

The Lister runs in the Executive. Commands look like procedure calls with constant (string, numeric, character, boolean) arguments. Arguments are type-checked by the command interpreter. To run the Lister, type to the Executive:

```
>Lister <command1[arg1, arg2, ...]> <switches> <command2[arg1, ...]> <switches>
```

You actually type the square brackets, as in a Mesa procedure call. For parameters of string type, quote marks are optional; the scanner will take any characters up to the next comma or right bracket if the first character is not a quote. The optional local switches are a sequence of zero or more letters preceded by a slash (/). Each letter is interpreted as a separate switch designator, and each may optionally be preceded by - or ~ to invert the sense of the switch. The switches that apply to each command are documented in the description of the command.

Almost all of the Lister commands read one or more object files and extract information from them. The files can be the output of either the Compiler, the Binder, or the Packager, although some commands require one or the other specifically. In the case of a single file, the parameter is the name of the file; if no extension is given, .bcd is assumed. Some commands take a list of files. In this case, the parameter specifies a file (such as **object.defs**) that contains a list of object files separated by blanks.

The commands are divided into two sections below: those of general use, and those used internally by the Mesa implementors. Quote marks are shown for command parameters that are of string type; it is usually not necessary to type them to the Lister.

27.2.1 Commands useful to general Mesa users

Compress["*FileList*"]

FileList is the name of a file that contains a list of compiler output object files. The **USING** lists of the directory statement are generated for each module in the list; they are then sorted to show for each interface, and for each item in the interface, which modules reference that item. The same caveat about implicitly included symbols applies as for the **Using** command. The output is written to *FileList.ul*.

Help[], Help["*CommandName*"]

Help[] will list the set of Lister commands and the command syntax for each. This can also be done by calling the Lister with no command, or by calling the Lister with a command it does not recognize. **Help["*Commandname*"]** will print the syntax for a particular command.

Implementors["*FileList*"]

FileList is the name of a file that contains a list of compiler output object files (interfaces and program modules). This command creates a file, *FileList.ilm*, showing where the various interface items are implemented for each interface exported by any program in the list. If the list also includes the object file for a particular interface, the interface items not implemented by any program are also shown. In order to run this command, you need not only the object files in the list, but also the object files for the interfaces exported by the programs therein. Missing object files are reported and the command attempts to forge on.

Interface["*FileName*"]

Given the object file for an interface (**DEFINITIONS** file), this command produces a list of the interface items and numbers (on *FileName.i1*). These numbers are the ones reported by the Binder for unbindable items in the absence of the proper symbols.

Stamps["*FileName*"]

FileName is a Compiler, Binder, or Packager output object file. This command generates a file, *Filename.bl*, that shows the version stamps of any modules bound in the file, and of all imports and exports of the top-level configuration in the file.

UnboundExports["*FileName*"]

FileName is a Compiler, Binder, or Packager output object file. This command examines all of the exported interfaces and generates a file, *FileName.x1*, which lists the items in those interfaces that are not exported by this module or configuration.

Using["*FileName*"]

FileName is a Compiler output object file. This command generates a directory statement with its included identifier lists (on *FileName.ul*). Since there is not enough information in the symbol table to tell reliably which symbols were implicitly included, the **USING** clauses may contain a superset of those items actually needed.

```
UsingList["FileList"]
```

FileList is the name of a file that contains a list of Compiler output object files. This command creates a ".ul" file for each file named in the list.

```
Version["FileName"]
```

FileName is a Compiler, Binder, or Packager output object file. This command shows, on **SimpleExec.log**, the object, source, and creator version stamps of the file.

```
Xref["FileList"]
```

FileList is the name of a file that contains a list of Compiler output object files. This command creates one or more files, *filename1.xref*, *filename2.xref*, etc. that contain a sorted list of all public declarations in the collection of modules and interfaces. A few dummy lines are inserted to make this file a Mesa program syntactically. You should run it through the Formatter (see the Formatter chapter) to make it more readable. If the /p switch is specified, the output file will also show the private declarations.

```
XrefFileSize[ByteCount]
```

This command tells the **Xref** command to limit the size of the output files to *ByteCount*.

```
XrefByCaller["FileList"]
```

FileList is the name of a file that contains a list of Compiler output object files. This command creates a single file, *FileList.xlr*, that shows for each procedure of each module in the list, what other procedures it calls. It does this by scanning the code for the modules. It does an imperfect job in that it cannot tell who is being called via a procedure variable. However, if there are any procedure variables called, it makes an entry for "*" in the list of called procedures. You can check these procedures by hand. It does not report calls to procedures nested within the given procedure.

```
XrefByCallee["FileList"]
```

This is similar to **XRefByCaller**, except that the results are shown sorted by callee, and the output file is named *FileList.xle*. Thus, the entry for "*" is the set of procedures in the list of modules that contain calls to procedure variables.

27.2.2 Commands useful to wizards

```
Bcd["FileName"]
```

FileName is a Compiler, Binder, or Packager output object file. This command produces a listing of the internal tables of the binary configuration description (on *Filename.b1*).

```
BcdLinks["FileName"]
```

This is the same as the **Bcd** command, except that the control links of imported and exported items are included.

BcdSegment["*FileName*", *Base*, *Pages*, *Links*]

This is the most general form of the **Bcd** command, which allows you to specify the location of the configuration description by file name, starting page number, number of pages, and whether you want the links (specify **TRUE** or **FALSE**).

Code["*FileName*"]

FileName is a Compiler output object file. This command produces a listing of the object code (on *filename.c1*). If the source file is available on your disk, the source for each statement is listed just before the object code.

Switches:

/d give all numbers in decimal.

/h give all numbers in hexadecimal.

/o give all numbers in octal (default).

Warning: This command produces a large amount of output.

Warning: If the module is subsequently packaged, the code offsets will change (although the sequence of operations will be the same). If you are making listings for low-level octal debugging, be sure to make new listings of code for packaged modules using the **CodeInConfig** command.

CodeInConfig["*Config*", "Module"]

This command produces a listing of the object code of a module that has subsequently been packaged. The listing reflects the new code offsets produced by the Packager. *Config* should be the bcd produced by the packager, or one including it. *Module* is a module within the packaged configuration. This command may also be applied to unpackaged configurations; in this case it produces the same output as the **Code** command. If the module is in a configuration that was bound with symbol copying, the symbols file must be available on the local file system.

Switches:

/d give all numbers in decimal.

/h give all numbers in hexadecimal.

/o give all numbers in octal (default).

OctalCode["*FileName*"]

This is the same as the **Code** command, except that opcodes are given in octal as well as by name.

Switches:

- /d give all numbers in decimal.
- /h give all numbers in hexadecimal.
- /o give all numbers in octal (default).

Warning: This command produces a very large amount of output.

OctalCodeInConfig["Config", "Module"]

This command is the combination of the **CodeInConfig** and **OctalCode** commands.

Switches:

- /d give all numbers in decimal.
- /h give all numbers in hexadecimal.
- /o give all numbers in octal (default).

Symbols["FileName"]

Given a Compiler output object file, this command lists the internal symbol table (on *FileName.s1*).

SymbolSegment["FileName", Base, Pages]

This is a more general form of the **Symbols** command, which allows complete specification of the location of the symbols (e.g., in a *.symbols* file).

There are several other commands that are either self-documenting or uninteresting to all but the most hardcore Compiler debuggers.



Performance tools

This chapter documents four tools that aid in the study of the behavior of Mesa programs: the CountPackage, PerfPackage, Spy, and Ben.

The CountPackage is based on trapping control transfers (**XFERS**). An **XFER** is the general control transfer mechanism in Mesa. The following are all **XFERS**: procedure call, return from a procedure, traps, and process switches. The CountPackage counts the number of control transfers (**XFERS**) to a module and records the time spent executing in a module. It can also be used to gather information on the flow of control between groups of modules.

The PerfPackage allows you to identify places in your programs and then collect timing and frequency statistics of program execution between these places.

Spy can measure the amount of time spent executing in a module, certain procedures, or even source statements within a procedure; it can optionally charge the caller for this time. The Spy operates by waking up periodically and sampling the PC. Spy is probably the simplest tool to use; it is especially useful for top-down analysis of a program (i.e., the Spy can be used to identify the hottest modules, then the hottest procedures within those modules, and so forth). It also has less effect on the execution of the client than the CountPackage or PerfPackage. However, the Spy is not as useful as the PerfPackage for studying very short or infrequent actions. The PerfPackage is best for studying the precise time spent in a module by various paths.

Ben is a package that is used to produce a list of the backing-store transfers that occur during some interval of client activity. The output report also contains other information, such as what caused the transfer to occur. This package is useful in determining why code and data is in the working set for a user action, and may be used to debug code packaging specifications.

All four tools come in two pieces: a client part and a tool that runs in CoPilot. The client part must always be loaded and started before any measurements can be made. For the CountPackage and PerfPackage the client part is **RuntimePerf.bcd**; for the Spy it is **SpyNub.bcd**; for Ben it is **Ben.bcd**. The tools for the CountPackage and PerfPackage are bound together in **CPPerf.bcd**; the Spy tool is contained in **Spy.bcd**; the data reduction program for Ben is contained in **ReduceBen.bcd**.

28.1 Control Transfer counter tool

The CountPackage is implemented as a set of commands that can be executed from CoPilot, a routine that intercepts all **XFERs** and collects statistics about them, and a routine that intercepts conditional breakpoints for turning the **XFER** monitoring on and off. Existing CoPilot commands are used to specify where **XFER** monitoring is enabled, and additional commands are provided for controlling the counting of **XFERs** and outputting the results.

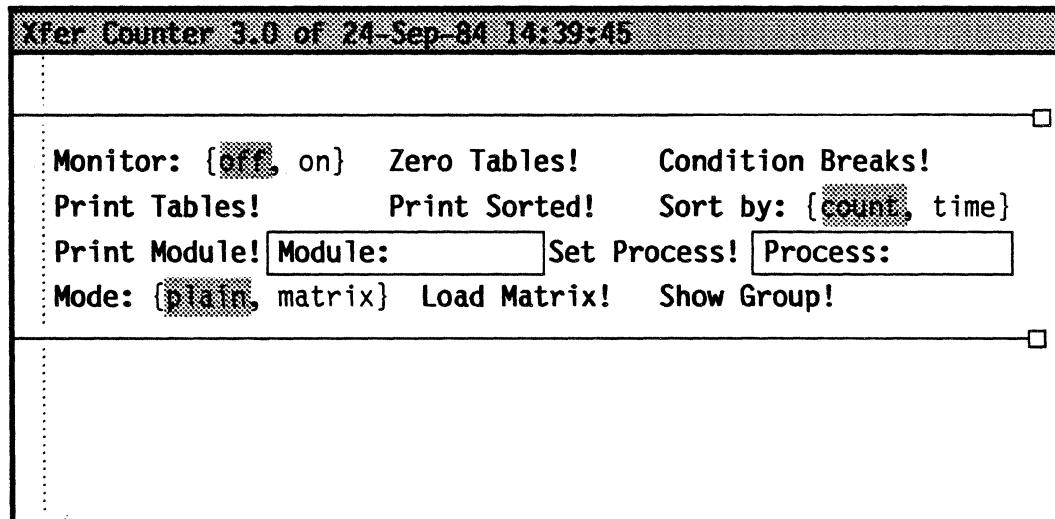
This tool is intended to provide a global view of the behavior of a system. With it, you can identify modules that warrant closer study with other tools such as the PerfPackage and Spy.

28.1.1 Files

Retrieve **RuntimePerf.bcd** onto the client volume. Retrieve **RuntimePerf.symbols** and **CPPerf.bcd** onto the debugger volume.

28.1.2 User interface

Interaction with the CountPackage is done through its window. There are three subwindows: the message subwindow, the form subwindow, and the log subwindow. Error messages and warnings are displayed in the message subwindow. Commands are invoked in the form subwindow. All output is displayed in the log subwindow and written on **Count.log**.



28.1 Control Transfer cCounter tool

Available commands are:

Monitor: {off, on}

turns off/on the tool's breakpoint handler. All conditional breakpoints will affect the state of **XFER** monitoring when the monitor is on and will behave as normal conditional breakpoints when it is off.

Zero Tables!	zeroes out all counts and times.
Condition Breaks!	makes all non-conditional breakpoints conditional by adding the condition "1" to them.
Print Tables!	displays all the statistics for each module in order of increasing global frame table index (gfi) for plain mode. In matrix mode, it displays the statistics for each nonzero element of the matrix. The output format of times is sec.msec:usec . This command may be aborted by typing ABORT .
Print Sorted!	displays all the statistics for each module in order of decreasing time or decreasing number of XFERS , depending on the value of Sort by . This command may be aborted by typing ABORT . This is not allowed in matrix mode.
Sort by: {count, time}	when set to count , the Print Sorted command displays table entries in order of decreasing number of XFERS ; otherwise it displays them in order of decreasing time.
Print Module!	displays the statistics for the module specified by Module . This is not allowed in matrix mode.
Module:	specifies the module to the Print Module command. It is either the module's global frame table index (gfi), its global frame address (g), or its module name (if the current configuration contains the desired module).
Set Process!	specifies that only those XFERS executed by the specified process are to be counted. The default case is to track all processes.
Process:	used by the Set Process command. It contains an octal ProcessHandle as obtained from the CoPilot's List Processes command. If Process is empty when Set Process is invoked, all processes are tracked.
Mode: {plain, matrix}	when set to plain (default), the Xfer Counter records transfers between modules. When set to matrix , the Counter records transfers from one group of modules to another.
Load Matrix!	reads the file to collect group information treating the current selection as a file name.
Show Group!	using the current selection as a group number, prints the names of the modules belonging to that group. This command may be aborted by typing ABORT .

28.1.3 Operation

There are two modes of operation: plain and matrix. Plain mode (the default) simply records the time spent in a module and the number of **XFERS** to that module. Matrix mode is used to gather information on the flow of control between groups of modules. Each module is a member of one of as many as 16 groups. A 16-by-16 matrix of counts and times is maintained by the Xfer Counter. The rows of the matrix are the groups of the source of the **XFER**, the **from** group. The columns of the matrix are the groups of the destination of the **XFER**, the **to** group.

In plain mode when **XFER** monitoring is enabled and an **XFER** occurs, the trap handler calculates the time since the last **XFER** and adds that to the cumulative time for the current module. It then calculates which module is the destination of the **XFER** and makes that the current module, incrementing its count. In matrix mode when **XFER** monitoring is enabled and a **XFER** occurs, the trap handler updates the appropriate element of the matrix. In both modes, the **XFER** handler then completes the **XFER**, and the client program continues.

The state of **XFER** monitoring can be controlled by two methods. The first is by setting a conditional break to be handled by the tool's breakpoint handler. The second is by calling the procedures **XferCountDefs.StartCounting** and **XferCountDefs.StopCounting**.

When the break handler intercepts a breakpoint, it checks to see if the breakpoint is conditional. If not, the break handler just proceeds to CoPilot. If it is, the state of **XFER** monitoring is changed and program execution is resumed. A condition of 0 turns on **XFER** monitoring; a condition of 1 toggles the state of **XFER** monitoring; a condition of 2 turns off **XFER** monitoring. Any other condition has no effect.

The procedures **XferCountDefs.StartCounting** and **XferCountDefs.StopCounting** provide an alternative method of enabling **XFER** monitoring. These procedures may be called from statements in the client program, or they may be called from the debugger's interpreter. If they are to be called from the CoPilot interpreter, you should set module context to **PilotCounter** and interpret call **StartCounting** and **StopCounting**.

Since multiple processes may interact with each other, there is the concept of the tracked process. If the tracked process is not **NIL**, only those **XFERS** that are encountered during execution of the tracked process are counted; all others are simply resumed. If the tracked process is **NIL**, then all processes are tracked.

The group information for matrix mode is entered into the Counter by reading an edited version of the output from the debugger's **Display GlobalFrameTable** command. Appending the group number to the line for a module will assign the module to that group. If no group number is specified, the module is assigned to the group of the previous line. Modules not assigned to any group are members of group 0. For example:

```
BcdOperations      G:400B 1    -- group 1
PilotLoadState    G:430B 2    -- group 2 the Loader proper
PilotLoaderSupport G:404B
PilotLoaderCore   G:444B
STLeafImpl        G:17554B 3 -- group 3 Pilot
SpaceImplB        G:17524B
SpaceImplA        G:17504B
STreeImpl         G:17324B
```

ProjectionImpl	G:17370B
STreeImpl	G:17124B
HierarchyImpl	G:17150B
VolFileMapImpl	G:20060B
FileImpl	G:17020B
CachedSpaceImpl	G:14644B
CachedRegionImplB	G:14400B
CachedRegionImplA	G:14314B
FileCacheImpl	G:13204B
ZoneImpl	G:14304B
UtilitiesImpl	G:14300B
HeapImpl	G:20334B
Processes	G:14120B

The significant part of each line in this matrix specification is the part that begins with "G:". This must be followed by a number, the actual `global frame handle` number. To assign that module to a group, the `global frame handle` must be followed by a space and the group number it is to go into. The rest of the line is ignored.

28.1.4 Limitations

Execution speed: `XFER` monitoring slows down the execution of a program considerably, since extra processing is done on every `XFER`. As a result, interrupt processes that are triggered by real-time events (e.g., the keyboard process) will run relatively more frequently.

Idle loop accounting: When no process is running, the Mesa emulator runs in its idle loop waiting for a process to become ready. This idle time is charged to the process that was last running.

Time base: The time base is a 32 bit counter, where the basic unit of time is a `System.Pulse` whose resolution varies between 1 and 1000 microseconds. The counter typically turns over about once an hour; no individual time greater than an hour is meaningful. Total times are 32-bit numbers and will overflow after 340 minutes.

Overhead calculation: Due to implementation restrictions and timer granularity, some of the overhead of processing an `XFER` trap is incorrectly assigned to the client program instead of the `CountTool`. As a result, times must be interpreted as only a relative measure of the time spent in a module.

Counter sizes: Counts are 32-bit numbers. The maximum total count is 4,294,967,295 `XFERS`.

Memory requirements: The `CountTool` requires 16 pages of the client's resident memory.

Worry mode: The `CountTool` operates in worry mode; see the chapter on CoPilot for more information about worry mode.

28.1.5 Getting started

The steps required for using the Count Tool are outlined in the following steps.

1. Retrieve **RuntimePerf.bcd** onto the client volume. Retrieve **RuntimePerf.symbols** and **CPPerf.bcd** onto the debugger volume.
2. Run **CPPerf** in CoPilot.
3. Start your program with **RuntimePerf** included. This can be done by running **RuntimePerf** in the Tajo executive.
4. Enter CoPilot and set conditional breakpoints to enable monitoring as desired.
5. Turn the break handler on by setting the **Monitor** parameter to **on**.
6. Proceed with program execution.
7. Return to CoPilot via an interrupt or an unconditional breakpoint.
8. Display results with the **Print** commands.

28.1.6 Sample session

The following annotated listing of **Debug.log** and **Count.log** should give a fair idea of the use of the count tool. It counts the **XFERS** executed when loading a module.

```

3-Feb-82 11:57
*** interrupt ***
-- set breakpoints to count XFERS involved with loading
>SET Root configuration: Tajo
>SET Module context: PilotLoaderCore
>Break Entry procedure: New Breakpoint #1.
>Break Xit procedure: New Breakpoint #2.
>ATTach Condition #: 1, condition: 0
>ATTach Condition #: 2, condition: 2
-- condition 1 turns XFER counting on; condition 2 turns it off
>LIST Breaks
1 -- Break at entry to New (in PilotLoaderCore, G: 444B). Condition:
0
2 -- Break at exit from New (in PilotLoaderCore, G: 444B).
Condition: 2
>Proceed [Confirm]
*** interrupt ***
-- look at the XFER count results
>--Test.map      -- file containing group information
-- set mode to matrix and load group information using Load Matrix
command
>Proceed [Confirm]
*** interrupt ***
-- look at the matrix

```

From Count.log:

Xfer Counter 8.0 of 2-Feb-82 17:32
 3-Feb-82 12:10

Track process: 100B -- ignore processes not involved in loading

-- Output of Print Tables command with mode = plain

Total Xfers 5,150
 Total Time 600:638

Frame	Module	#Xfers	%Xfers	Time	%Time
13750B	FrameImpl	20	.38	805	.13
14120B	Processes	45	.87	3:539	.58
20334B	HeapImpl	64	1.24	4:115	.68
14300B	UtilitiesImpl	39	.75	9:900	1.64
14304B	ZoneImpl	22	.42	5:266	.87
13204B	FileCacheImpl	28	.54	2:734	.45
13440B	SubVolumeImpl	2	.03	633	.10
14314B	CachedRegionImplA	172	3.33	60:754	10.11
14400B	CachedRegionImplB	100	1.94	8:259	1.37
14644B	CachedSpaceImpl	87	1.68	17:584	2.92
16460B	MStoreImpl	1	.01	115	.01
16570B	PageFaultImpl	20	.38	1:496	.24
17020B	FileImpl	32	.62	2:331	.38
20060B	VolFileMapImpl	39	.75	3:482	.57
20164B	VolumeImpl	20	.38	1:323	.22
17150B	HierarchyImpl	95	1.84	5:698	.94
17124B	STreeImpl	120	2.33	20:433	3.40
17370B	ProjectionImpl	73	1.41	5:266	.87
17324B	STreeImpl	276	5.35	55:861	9.30
17310B	MapLogImpl	5	.09	805	.13
17504B	SpaceImplA	189	3.66	11:051	1.83
17524B	SpaceImplB	34	.66	2:273	.37
17554B	STLeafImpl	72	1.39	6:792	1.13
12570B	DiskChannelImpl	16	.31	1:064	.17
444B	PilotLoaderCore	2,483	48.21	168:017	27.97
404B	PilotLoaderSupport	176	3.41	10:418	1.73
430B	PilotLoadState	52	1.00	127:955	21.30

-- Output of Print Sorted command with Sorted by = count

Total Xfers 5,150
 Total Time 600:638

Frame	Module	#Xfers	%Xfers	Time	%Time
444B	PilotLoaderCore	2,483	48.21	168:017	27.97
400B	BcdOperations	868	6.85	62:654	10.43
17324B	STreeImpl	276	5.35	55:861	9.30
17504B	SpaceImplA	189	3.66	11:051	1.83
4 04B	PilotLoaderSupport	176	3.41	10:418	1.73
14314B	CachedRegionImplA	172	3.33	60:754	10.11

17124B	STreeImpl	120	2.33	20:433	3.40
14400B	CachedRegionImplB	100	1.94	8:259	1.37
17150B	HierarchyImpl	95	1.84	5:698	.94
14644B	CachedSpaceImpl	87	1.68	17:584	2.92
17370B	ProjectionImpl	73	1.41	5:266	.87
17554B	STLeafImpl	72	1.39	6:792	1.13
20334B	HeapImpl	64	1.24	4:115	.68
4 30B	PilotLoadState	52	1.00	127:955	21.30
14120B	Processes	45	.87	3:539	.58
14300B	UtilitiesImpl	39	.75	9:900	1.64
20060B	VolFileMapImpl	39	.75	3:482	.57
17524B	SpaceImplB	34	.66	2:273	.37
17020B	FileImpl	32	.62	2:331	.38
13204B	FileCacheImpl	28	.54	2:734	.45
14304B	ZoneImpl	22	.42	5:266	.87
13750B	FrameImpl	20	.38	805	.13
20164B	VolumeImpl	20	.38	1:323	.22
16570B	PageFaultImpl	20	.38	1:496	.24
12570B	DiskChannelImpl	16	.31	1:064	.17
17310B	MapLogImpl	5	.09	805	.13
13440B	SubVolumeImpl	2	.03	633	.10
16460B	MStoreImpl	1	.01	115	.01
Ignored Xfers	973 -- Xfers not in the tracked process				
Ignored Time	86:829 -- time spent outside tracked process				

Tables zeroed

Matrix loaded

Track process: 100B

-- Output of Print Tables command with mode = matrix

Total Xfers	4,919			
Total Time	523:623			
From -> To	#Xfers	%Xfers	Time	%Time
-----	-----	-----	-----	-----
1 -> 2	854	17.36	70:482	13.46
2 -> 1	861	17.50	62:596	11.95
2 -> 2	1,759	35.75	194:121	37.07
2 -> 3	30	.60	2:244	.42

Ignored Xfers 973

Ignored Time 86:829

28.2 Performance Measurement Tool

The Performance Measurement Tool (PerfPackage) uses CoPilot's breakpoint mechanism to collect timing and frequency statistics of program execution between breakpoints. The client part of the PerfPackage, `RuntimePerf.bcd`, contains a routine that intercepts all conditional breakpoints and collects statistics about them. Existing CoPilot commands are used to specify what points are to be monitored, and the tool provides commands for controlling the measurements and outputting the results.

28.2.1 Files

Retrieve `RuntimePerf.bcd` onto the client volume. Retrieve `RuntimePerf.symbols` and `CPPerf.bcd` onto the debugger volume from the Release directory.

28.2.2 Concepts

A *node* is defined to be a point in a program where a breakpoint can be set by CoPilot. In fact, nodes are implemented via conditional breakpoints, so that while monitoring is turned on, the *functioning of all conditional breakpoints is different*. In particular, conditional breakpoints cause performance data to be gathered rather than a breakpoint to be taken. The number of times a node is encountered is tallied by the Perf Package.

A *leg* is defined by a pair of nodes, one called the *from* node and the other the *to* node. A leg is the code executed between these nodes. Interesting items measured about a leg include the number of times this leg was executed and the time required to execute the leg.

Facilities are also provided for associating a *histogram* with any node or leg, thereby providing more detailed distribution information about the entry than is provided by counts, sums, and averages.

Since *processor time* or *task time* is not available, the measure of computing is simply the *elapsed time* between the time the *from* node is executed and the time the *to* node is executed.

28.2.3 Definition of terms

<i>Node Table</i>	A <i>node table</i> is a table maintained by the measurement module that contains information about each node. A node for each conditional breakpoint is entered into this table by the Collect nodes command or by the measurement module when it encounters a conditional breakpoint that is not already in the table. The node table has 20 entries.
<i>NodeID</i>	A <i>NodeID</i> is the name of a node in the node table, used in commands to identify a particular node. This is the same as the breakpoint number assigned by CoPilot.
<i>Leg Table</i>	A <i>leg table</i> is a table maintained by the measurement module containing various information about each leg. Legs are entered into this table by the command Add Legs or by the measurement module when it encounters a new leg and automatic insertion is enabled. The leg table has 41 entries, one of which is reserved.
<i>LegID</i>	A <i>LegID</i> is the name of a leg in the leg table. The LegID for a particular leg does not change during a measurement session and is used in commands to identify a particular leg.
<i>Histogram</i>	A <i>histogram</i> is an optional table that may be associated with either a node or leg that records the distribution of a variable associated with the node or leg by incrementing counters in a number of <i>buckets</i> . The distribution may be either <i>linear</i> or <i>logarithmic</i> . In a linear

distribution, a *base* may be specified which will be used as the offset for the first bucket. In a logarithmic distribution, the buckets are indexed by the number of leading binary zeros in the *value*. A *scale* is used to adjust the value for an optimal fit into the number of buckets. There is a storage pool of 256 words that is shared among all histograms to hold buckets and histogram information.

Node Histogram A *node histogram* is a histogram associated with a node. The histogram variable of the node is the first variable in the conditional expression attached to the breakpoint that defines the node. The value is treated as a 32-bit unsigned quantity. For a simple node histogram, the value is adjusted by subtracting the base (if any) and dividing by the scale factor; the resulting quotient is recorded. A logarithmic node histogram has a maximum of 32 buckets because the value is a 32-bit quantity.

Leg Histogram A *leg histogram* is a histogram associated with a leg. The histogram variable of the leg is the 32-bit leg time in units of pulses. The value is adjusted by shifting the value to the right by the scale. A logarithmic leg histogram has a maximum of 32 buckets because the value is a 32-bit quantity.

28.2.4 User interface

Interaction with the PerfPackage is done through its window. There are four subwindows: the message subwindow, the common commands subwindow, the specific commands subwindow, and the file subwindow. The commands available in the specific commands subwindow depend on whether you are using the PerfPackage's histogram facilities. They are either the *Mode Commands* or the *Histogram Commands*. You may change the commands available in this subwindow by using the **Commands** pop-up menu.

Common Commands

Monitor: {off, on} turns off/on performance monitoring. All conditional breakpoints will be monitored when the monitor is on, and will behave as normal conditional breakpoints when it is off.

Condition Breaks! makes all non-conditional breakpoints into conditional breakpoints by adding the condition "1" to them.

Collect Nodes! enters all currently existing conditional breakpoints as nodes in the node table.

Add Leg! adds the leg specified by **From Node** and **To Node** to the leg table. If a designated leg entry is already in the leg table, the leg is not affected.

From Node: contains the **NodeID** of the **from** node for the **Add Leg** command. The character "*" may be used as a wild card meaning "all nodes."

To Node:	contains the NodeID of the to node for the Add Leg command. The character "*" may be used as a wild card meaning "all nodes."
Delete Leg!	deletes the specified leg from the leg table.
Leg:	contains the LegID used by the Delete Leg command.
Print Tables!	displays all the summary statistics gathered so far and the complete contents of the node table and the leg table. This command may be aborted by pressing ABORT .
Print Nodes!	displays the contents of the node table. A NodeID followed by an asterisk has a histogram associated with it. This command may be aborted by pressing ABORT .
Print Legs!	displays the contents of the leg table. A LegID followed by an asterisk has a histogram associated with it. This command may be aborted by pressing ABORT .
Zero Tables!	zeroes out all counts and sums from the tables (including the total time spent measuring) but leaves all other information in the tables unchanged. This command is useful for preserving the measurement environment while zeroing out the counts and sums collected so far.
Reinitialize Tables!	completely reinitializes all tables and counters. The node table, the leg table, and all histograms are cleared.

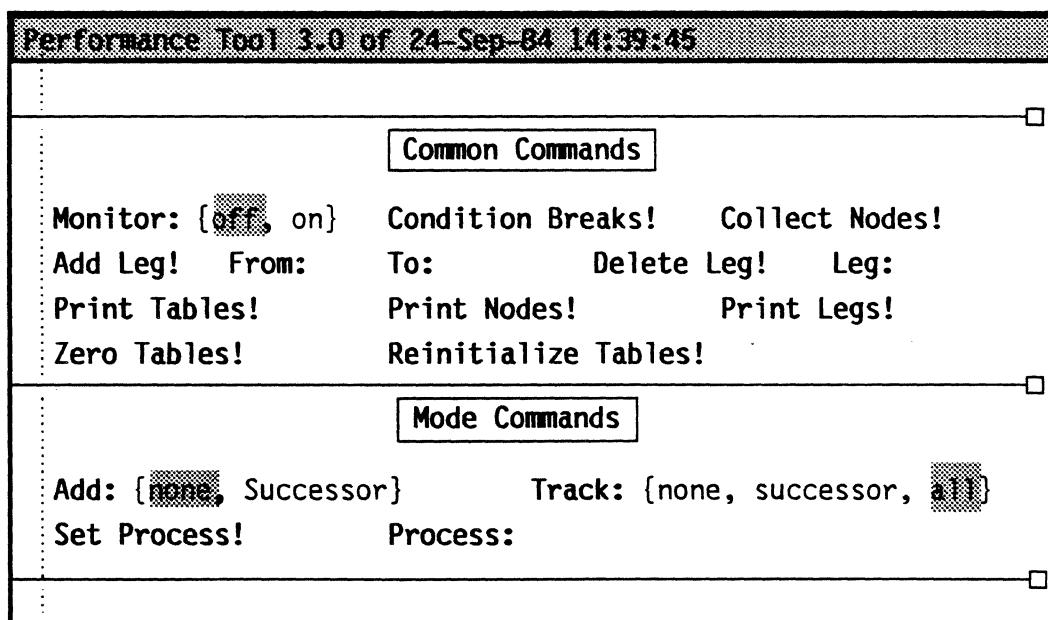


Figure 28.2: PerfPackage window with mode commands

Mode Commands

Add: {**none**, **successor**} if set to **none**, prevents the PerfPackage from adding legs that are not in the table as it encounters pairs of nodes during the execution of the client program that have not been specified as legs already. This is the default mode for automatically adding legs. If set to **successor**, the PerfPackage adds legs that are not in the table. These legs may be deleted if there is no room in the leg table when legs are added by the **Add Legs** command.

Track: {**none**, **successor**, **all**} if set to **none**, the PerfPackage disables tracking of legs. If set to **successor**, the PerfPackage tracks only the leg defined by the last node encountered and the current node. If set to **all**, the PerfPackage tracks all legs in the table. This is the default mode for tracking legs.

Set Process! tells the PerfPackage to track only those legs that are executed by the process specified by **Process**. Nodes encountered by other processes will not be recorded. An octal **ProcessHandle** as obtained from CoPilot's **List Processes** command is acceptable as input to this command. The default case is to track all processes.

Process: used by the **Set Process** command. It contains an octal **ProcessHandle** as obtained from CoPilot's **List Processes** command. If **Process** is empty, all processes are tracked.

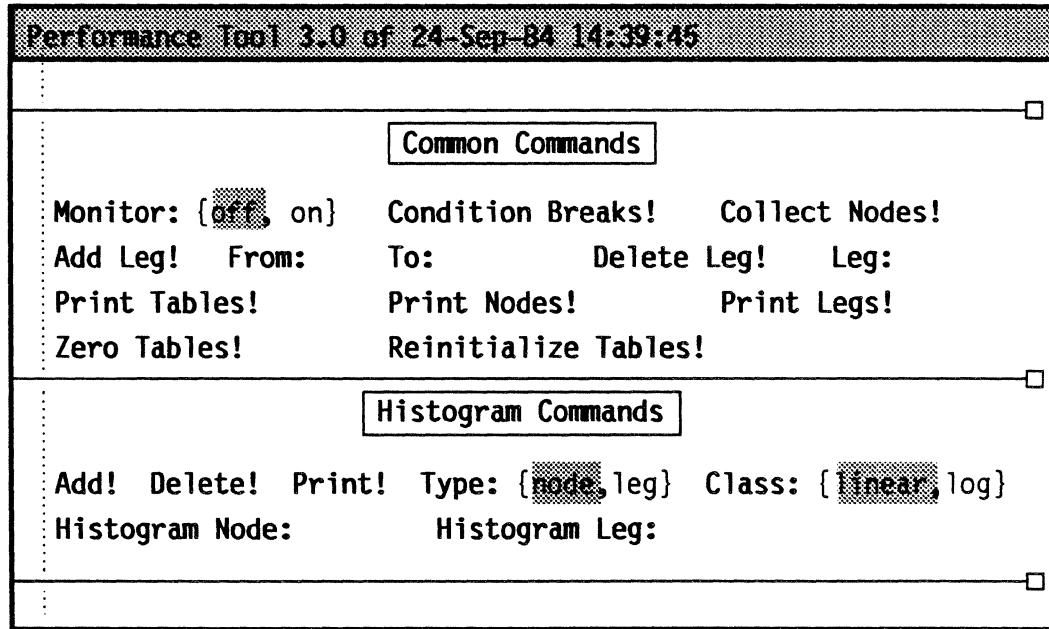


Figure 28.3: PerfPackage window with histogram commands

Histogram Commands

Add!	adds a histogram and associates it with either Histogram Node or Histogram Leg , depending on the value of Type . The command gets its parameters from the Class , Buckets , Scale , and Base fields.
Delete!	deletes the histogram associated with the specified node or leg.
Print!	displays the histogram associated with the specified node or leg. This command may be aborted by typing ABORT .
Type: {node, leg}	if set to node , the above histogram commands operate on the histogram associated with the node specified by Histogram Node . If set to leg , the above commands operate on the histogram associated with the leg specified by Histogram Leg .
Class: {linear, log}	used to specify the kind of distribution of the histogram to the Add command.
Histogram Node:	contains a NodeID for specifying a node to the Add , Delete , and Print commands.
Histogram Leg:	contains a LegID for specifying a leg to the Add , Delete , and Print commands.
Buckets:	used to specify the number of buckets to the Add command.
Scale:	used to specify the scale of the histogram to the Add command. Note that since scaling of a leg histogram is done by shifting instead of dividing, the scale is entered as a power of two.
Base:	used to specify to the Add command the base of the distribution of values for linear histograms.

28.2.5 Operation

When the break handler intercepts a breakpoint, it checks to see if the breakpoint is conditional. If so, it finds the node corresponding to the breakpoint, increments its counters, and processes its histogram if one exists. If tracking of legs is enabled, the leg table is searched for the legs of which this node is a part. Otherwise, the breakpoint is resumed.

In the simple case, a leg is tracked as follows: The break handler intercepts a conditional breakpoint that is the *from* node of the leg **from**, and some time later it intercepts a conditional breakpoint that is the *to* node of the leg **to**. At this point, the leg's time is recorded, its count is incremented, and its histogram (if any) is processed.

This simple model of tracking a leg is complicated by recursion, signals, and multiple processes. With recursion, **from** may be encountered several times before **to** is encountered. With signals, a process may be unwound after it encounters **from** but before it encounters **to**. With multiple processes, one process may encounter **from** and then another immediately encounter **to**.

To deal with these complications, there is a *leg owner*. A leg owner is the process that last encountered **from**. When **to** is encountered and the current process is its owner, then the leg is recorded and the leg owner is cleared. If the current process is not the owner, the leg is ignored. As a result of ignoring legs, **from** and **to** may be counted more times than the leg between them is counted.

To deal with the complication of multiple processes, there is the concept of the *tracked process*. If the tracked process is not **NIL**, then only those conditional breakpoints that are encountered by the tracked process are treated as nodes. All others are simply resumed as if they did not exist. If the tracked process is **NIL**, then all processes are tracked.

Normally, when a node is encountered, all legs of which it is a part are tracked. Alternatively, only the leg defined by the last node encountered and the current node is tracked.

28.2.6 Limitations

Time base: The time base is a 26-bit counter, where the basic unit of time is a **System.Pulse** whose resolution varies between 1 and 1000 microseconds. The counter typically turns over about once an hour; no individual time greater than an hour is meaningful. Total times are 32-bit numbers and will overflow after 340 minutes.

Overhead calculation: Due to implementation restrictions and timer granularity, some of the overhead of processing a breakpoint is incorrectly assigned to the client program instead of the PerfTool. As a result, leg times will be about 10 microseconds high for each node that was encountered while processing that leg. Elapsed time is similarly affected. This effect is particularly noticeable with short legs. Comparing relative times of different legs may give better information about program performance.

Counter sizes: In a long measurement session, the node, leg, or histogram counters may overflow. Node and leg counters are 22 bits, while histogram counters are 16 bits. If a node or leg counter overflows, a "*" follows the count when the field is listed.

Recursive procedure calls, UNWINDs, multiple processes: These interfere with the simple start-to-end concept of a leg. With recursion and multiple processes, the start node of a leg may be tripped several times before the end node is tripped. With unwinding, the start node of a leg may be tripped and the end node never reached. If any of these cause a leg to be ignored, the referenced field in the Leg Table has a "~" following it when the table is listed.

Breakpoints taken twice: Nodes are implemented as conditional breakpoints. If for some reason the broken instruction is interrupted (e.g., it takes a page fault), the breakpoint is taken again, and that node will get an extra count. This can cause node counts to be greater than leg counts for corresponding legs, and is another cause of "~" appearing in the Leg Table.

Table sizes: The node table contains 20 entries. (Note that the PerfPackage automatically extends the number of conditional breakpoints that can be set in the debugger from 5 to 20.) The leg table currently has 40 entries. Note that this number is small when compared to the 20*20 possible legs. For this reason, there are a number of commands that give you control over exactly what legs are in the table.

Memory requirements: The Perf Tool requires seven pages of the client's resident memory; three for PerfPackage's code and four for PerfTool's frames. This may affect the performance of systems that use a lot of memory.

Worry mode: The PerfPackage operates in worry mode; see the Debugger chapter for more information about worry mode.

28.2.7 Getting started

The steps required for using the measurement tool are outlined below.

1. Retrieve **RuntimePerf.bcd** onto the client volume. Retrieve **RuntimePerf.symbols** and **CPPerf.bcd** onto the debugger volume from the Release directory.
2. Run **CPPerf** in CoPilot.
3. Start your program with **RuntimePerf** included.
4. Enter CoPilot and set breakpoints as desired; then condition them with the **Condition Breaks** command.
5. Turn measurements on by setting the **Monitor** parameter to **on**.
6. Collect nodes and manipulate the leg table as desired.
7. Proceed with program execution.
8. Return to CoPilot via an interrupt or an unconditional breakpoint.
9. Display results with the **Print** commands.

28.2.8 Sample session

The following annotated listing of **Debug.log** and **Perf.log** should give a fair idea of the use of the measurement tool. It monitors the time required for the swapper to allocate real memory pages.

```
10-Feb-82 12:42
*** interrupt ***
Performance Tool 8.0 of 2-Feb-82 17:32
10-Feb-82 12:46
>SET Root configuration: Tajo
>SET Module context: PilotLoaderCore
-- set breakpoints to time the ProcessLinks procedure inside the
Loader
```

```

>Break Entry procedure: ProcessLinks Breakpoint #1.
>Break Xit procedure: ProcessLinks Breakpoint #2.
-- Condition breaks wth the PerfTool; turn on PerfTool
>Break Xit procedure: New Breakpoint #3.
>List Breaks
1 -- Break at entry to ProcessLinks (in PilotLoaderCore, G: 444B).
Condition: 1
2 -- Break at exit from ProcessLinks (in PilotLoaderCore, G: 444B).
Condition: 1
3 -- Break at exit from New (in PilotLoaderCore, G: 444B).
>Proceed [Confirm]
Break #3 at exit from New, L: 4470B, PC: 1237B (in PilotLoaderCore,
G: 444B)

```

From **Perf.log**:

```

Performance Tool 8.0 of 2-Feb-82 17:32
10-Feb-82 12:46

```

```

Collecting nodes 1 2 done
Leg from 1 to 2 added

```

```

-- Proceed from CoPilot to collect information
-- unconditional break returned control to CoPilot after loading

```

Total Elapsed Time of Measurements =	205:517
Elapsed Time less PerfMonitor Overhead =	204:366
Total Overhead of PerfMonitor Breaks =	1:151
Total number of Perf Breaks handled =	4
Average Overhead per Perf Break =	287
% of Total Time spent in PerfMonitor =	.56

- - - - - N O D E T A B L E C O N T E N T S - - - - -

Node Id	Global Frame	Program Counter	Number of References	Config Name	Module Name
1	444	3032	2	Tajo	PilotLoaderCore
2	444	3115	2	Tajo	PilotLoaderCore

- - - - - L E G T A B L E C O N T E N T S - - - - -						
Leg	From	To	# of Times	Total Time	Longest Time	
Id	Node	Node	Referenced	sec.msec:usec	sec.msec:usec	sec.msec:usec
0	1	-> 2	2	53:502	27:834	
Shortest Time		Average Time	% of Time			
		sec.msec:usec	sec.msec:usec	sec.msec:usec		
		25:668		26:751	26.17	

28.3 Spy

Spy is a performance measurement tool for determining where a program spends its time. The SpyNub is the client part; Spy is the tool executing in CoPilot that interprets the data recorded by the SpyNub. The SpyNub works by waking up on every display vertical field and incrementing a count in a bucket for the current PC. Spy's default mode is to collect information on a module level only; i.e., it has one bucket for every module. In addition, it can be instructed to create buckets for procedures or all the statements within a procedure. Spy also allows control over which processes to watch. The major advantages of Spy over the CountPackage and PerfPackage are that it is easy to use and has little impact on the client. However, because Spy samples on the vertical retrace, it is a poor choice to study actions of short duration; the PerfPackage is recommended for that use.

28.3.1 Files

Retrieve **SpyNub.bcd** onto the client volume and **Spy.bcd** onto the debugger volume.

28.3.2 User interface

Interaction with the Spy is done through its window.

Available commands are listed below:

Spy: {on, off}

Spy must be turned on to start spying. The interface **SpyClient** contains the procedures **StartCounting** and **StopCounting** if you want to do this from a program.

DisplayData!

causes the Spy to display its tables; **ABORT** aborts this display.

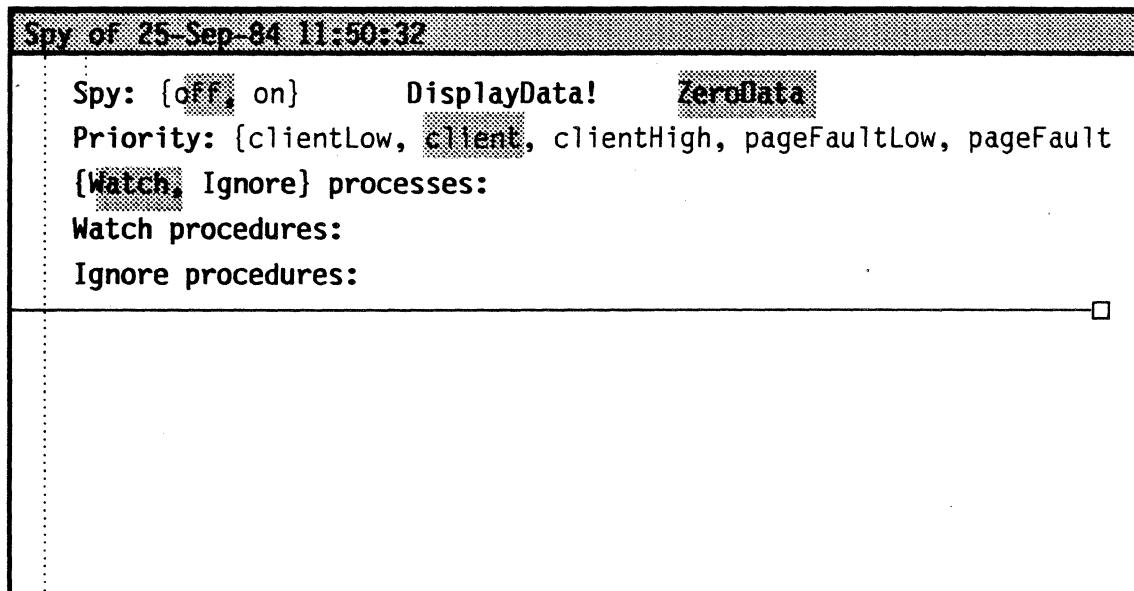


Figure 28.4: Spy tool window

ZeroData

is a Boolean that determines, in part, whether the buckets will be zeroed when execution of the client proceeds. If anything is changed in the **Priority**, **Processes**, or **Procedures** specifications, the buckets will be zeroed regardless of the setting of **ZeroData**. If, when you proceed, none of these specifications has changed, the buckets will be zeroed only if **ZeroData** is TRUE. Thus, if you happen to hit a breakpoint or press **CALL DEBUG** to enter CoPilot while the Spy is on, you can proceed without disturbing the counts just by setting **ZeroData** to FALSE.

Priority:

{clientLow, client, clientHigh,
pageFaultLow, pageFaultHigh, IOLow, All}

specifies the priority of the processes to Spy on: **clientLow** is Process.priorityBackground; **client** is Process.priorityNormal; **clientHigh** is Process.priorityForeground.

{Watch, Ignore} processes: If a list of processes is specified, ({**Watch, Ignore**} **processes:** P1, P2, ..., etc.), only those processes will be watched (ignored); all others will be ignored (watched). If no list appears, the default is that all processes of the indicated priority will be watched (or ignored, but this isn't very useful). Processes are specified in the same way you would to CoPilot, with the additional feature that you may write P1..P2 to specify all processes in the inclusive range P1 to P2. The default radix is octal.

Watch procedures:**Ignore procedures:****Watch procedures:** M1; M2: p1, p2/s; etc.;**Ignore procedures:** M3: p4; M4; etc. means:
"watch all procedures in module M1, watch only
procedures p1 and p2 in module M2, but watch p2 at
the individual statement level; watch all procedures
in M3 except p4, and ignore M4 entirely". /s means to
make a source level accounting. If the module being
watched was compiled with the j switch, use of the /s
option in Spy may produce invalid information. Note
that it's an error to mention the same module name
more than once in these lines, and that the /s option
is useless on the **Ignore** line. There is an accelerator
in the form of a pop-up menu for setting watched and
ignored procedures.

28.3.3 Operation

The most common way to use the Spy is to simply turn it on and perform some client operation. After doing a **DisplayData** to see where the client is spending time, it is a simple matter to use procedure level or source level Spying to track the problem down further. If no hot spots are immediately apparent, the Spy can be instructed to ignore some set of modules that provide a function (e.g., swapping). When an ignored module is found, Spy will continue up the call stack until it finds a valid module that will be charged instead. This has the effect of charging the caller of that function for the service rather than charging the procedure or module itself. When a hot spot does appear, you know who is using that function excessively.

Before a Proceed is done by CoPilot, Spy zeroes its tables and interprets the contents of the fields of processes and procedures to watch. If the number of buckets needed by the SpyNub to handle the data is greater than the amount already allocated, the Spy calls to the client world (after printing the message *Allocating extra buckets*) to allocate more before letting the Proceed finish.

The Spy looks up module names within the configuration currently set in CoPilot. If the module is not found, the Spy enumerates the global frame table, which can be slow. Because of this, a global frame handle may be used instead of a module name, which is much faster.

28.3.4 Getting started

The steps required for using Spy are:

1. Retrieve **Spy.bcd** onto the debugger volume and **SpyNub.bcd** onto the client volume.
2. Run Spy in CoPilot.
3. Start your program with SpyNub included.
4. Enter CoPilot and turn on Spy.

5. Proceed with program execution.
6. Return to CoPilot via an interrupt or an unconditional breakpoint.
7. Display results with the **DisplayData** commands.
8. Repeat steps 5-7 with modules ignored or watching procedures to find hot spots.

28.3.5 Error messages

SpyNub not found!

You forgot to load the SpyNub.

SpyNub not started!

SpyNub is loaded, but it hasn't been started.

xxx is ambiguous!

There is more than one instance of **xxx**.

xxx is crossjumped!

xxx was compiled with the **j** switch. Beware of source level data.

Symbol table for module containing xxx is missing!

Adequate symbols for the procedure **xxx** are not available. You should fetch the correct object or symbols files.

No symbols for xxx!

No symbols have been found for **xxx**.

xxx is an invalid global frame!

Invalid global frame specified in **Watch** or **Ignore Procedures** section.

xxx is not a module!

xxx is neither a module name nor a valid global frame address.

xxx is not a number!

Invalid number.

xxx begins an illegal process range!

Invalid process range.

/... is illegal after xxx!

Invalid use of switch.

modulename is mentioned more than once!

A module name may appear only once in the **Watch** or **Ignore** list.

28.3.6 Limitations

Sampling technique: Because Spy does its sampling based on the vertical retrace, no process with a priority lower than background can be watched. In addition, processes that do a `UserTerminal.WaitForScanLine` will look as if they are taking more time than they actually do.

Counter sizes: Counts are 32-bit numbers. The maximum total count is 4,294,967,295.

Memory requirements: The SpyNub requires 12 pages of the client's resident memory: three for its code, eight for module buckets, and one spare for extra buckets. One extra page is allocated for about every additional 50 buckets. This may affect the performance of systems that use a lot of memory.

Frame faults: Note that if a procedure call causes a frame fault (e.g., the procedure called has a large local frame), the time that Pilot takes to allocate the frame is charged to the caller, not to the called procedure.

28.4 Ben

Backing-store transfer tracing, of which page faults are a special case, is accomplished with two programs. The data is generated by the program `Ben.bcd`, which runs in the environment to be monitored. The other program, `ReduceBen.bcd`, is used to process the raw data generated by Ben, and produces a human-readable text file as output. It runs in CoPilot. These programs are described below.

28.4.1 Files

Retrieve `Ben.bcd` onto the client volume. Retrieve `Ben.symbols` and `ReduceBen.bcd` onto the debugger volume.

28.4.1.1 Collecting the data

To collect the data, load and start `Ben.bcd` in the environment to be investigated.

To start tracing transfers, get to CoPilot and tell Ben to begin tracing. Proceed as follows:

```
>SET Module context: BenImpl
> StartTracing[] --(note the leading space)
```

or

```
> BenImpl$StartTracing[] --(note the leading space)
```

You must have **Ben.symbols** on your debugger volume to do this.

When the **StartTracing** operation completes, you will be back in CoPilot. Proceed back to the client world.

```
>
>Proceed [Confirm]
```

Now perform the sequence of user operations that you wish to monitor. When done, get back to CoPilot, and finish the tracing by doing

```
>SEt Module context: BenImpl
> StopTracing[] --(note the leading space)
```

or

```
> BenImpl$StartTracing[] --(note the leading space)

logFileLength -- (printed by CoPilot)
>
```

Backing-store transfer data will have been recorded in a file in the root directory of the client system volume. When **StopTracing** returns to CoPilot, it reports the number of disk pages used by the trace log file.

When tracing is started, Ben creates the log file to hold the trace data. Tracing terminates either when the log file fills up or the user instructs Ben to stop. It is possible to adjust the maximum amount of data to be captured by setting a variable in **BenImpl**. The variable **nBuffers** (default value: 10) times the variable **bufferPages** determines the maximum size of the log file. Adjust **nBuffers** if you need a larger log file. This variable must be set before **StartTracing** is called. The requested size of the log file will be trimmed as necessary to fit on the client volume.

28.4.1.2 Reducing the data

The data reduction program **ReduceBen.bcd** runs in CoPilot in the Executive. The simplest way to use it is to collect the data with Ben and then immediately analyze it.

If transfer data is to be analyzed at a later time, ReduceBen requires that the volume that CoPilot is currently debugging have the same load state as when the tracing data was generated. This means it must have the same boot file and loaded configs as were present during the test, and that all loaded configs must be currently loaded in the same order that they were during the test.

The debugger volume should have all of the symbols for the client environment that might be referred to in the data file. If they are not, ReduceBen will report the symbols needed.

To analyze the transfer data, give the Executive the command:

>ReduceBen clientVolume/v6

ReduceBen will read the log file from the client volume and produce a file with the default name **Swapping.log** on the debugger volume containing the output report.

The full form of the command, with all of the default names explicitly specified, is

>ReduceBen /sd Swapping.log/o Swapping.data/i Star/v6

Filename/o specifies the name of the output file name. *Filename/i* specifies the name of the input file name. This makes sense only if you have used some utility program to copy the log file from the client root directory into a file on the debugger volume. If an input filename is not specified, the log file in the root directory of the specified volume is used.

The global switch **s** tells ReduceBen to print the source line of the program that caused the transfer, if the source file is available.

The global switch **d** sets the Debug mode. The dictionary contents are displayed in the Executive along with the output file contents.

ReduceBen registers a help command with the Executive. Typing "Help ReduceBen" will produce a short explanation of the command line format.

28.4.1.3 Report format

The output is a sequence of text lines, two or three per transfer. The format of the first line is

dt: number; Page: octal-number; location

where a *location* is either

File: file.file - type: type

or

swap-unit-type: name

or

volume root page: type

or

unknown backing store: data

The meaning of each of these fields is as follows:

dT: *number* is the number of microseconds that elapsed since the last backing-store transfer. This is real time, and will be slightly distorted because Ben is running.

Page: *octal-number* is the virtual page number of the transferred page.

location is an attempt to determine what the transferred page represents. It may be swapped from the disk or from some other backing store. In the former case, the page may represent either a specific file or otherwise. If otherwise, the rest of the line is reported as if the transferred page were backed by a file, thus the line **File:**. If it was a specific file, the rest of the line is reported as swap-unit-type. In the case of non-disk backing store, the volume root page and file type may be found, or Ben may not even be able to get that much information, thereby resulting in an unknown.

File: *file.file* occurs if it could be found in Pilot's caches. The file ID is reported as seven octal numbers. If the ID could not be found, **NIL** is inserted in the line.

type: *type* is either **file** if the page is backed by a file, or **data** if it is backed by the default backing file.

swap-unit-type can be one of four values: **Pack** - indicates that the page is a packaged swap unit; **Frame** - the page is in a swappable frame; **Module** - the page is in an unpackaged module; **?** - the type of page is unknown but it points to code or frames.

name is the name of the module, code pack, or frame, or "**anonymous**", if it cannot be determined.

volume root page is the physical volume page number of the transferred page for a non-disk backing store.

type is an octal number indicating the file type for whatever kind of non-disk backing store the page is on.

unknown backing store indicates the transferred page is not backed by the disk but by some other unknown source.

data consists of seven octal numbers giving the transfer data from Pilot's backing store. You would need to interpret this data according to the backing store used.

The second line for each item gives information about where the program was executing when the transfer occurred. It has the format

Called from module: *module-name*; Proc: *proc-name*; Type: *proc-type*

where

Called from module: *module-name* indicates the module that was executing.

Proc: *proc-name* indicates the name of the procedure or number of the catch phrase that was executing.

Type: *proc-type* is the type of procedure: **normal** - a normal procedure; **MAIN** - mainline code in the module; **nested** - a procedure nested in another; **catch** - a catch phrase in the module.

If the name for either a module or a procedure cannot be found, an annotation will be made in the output and the field left blank. This usually occurs because the (correct) symbols could not be found.

If you have specified the global switch **s**, a third line may appear for each item. This will be the source line corresponding to the place in the program that caused the transfer. This line will be output in the same format that CoPilot uses for showing source locations within a program. If there were no symbols for the module or if the source file was not found, this third line will not appear in the output.

The output file can become quite large. In a test case of 2000 transfers, a 500-page output file was generated. In Gacha 8, 10 disk pages roughly correspond to a printed page.

28.4.1.4 Error recovery

ReduceBen must sort all of the configurations by page number. To do this, it creates a data file whose initial size is 500 pages. If the data won't fit in the file, the file is dumped, its size is increased by 100 pages, and the sorting is attempted again. This will continue until either there is no space left on the debugger volume or the sort completes. The sorted information is called the dictionary. When the sorting starts, the message **Building dictionary** is displayed. If the sort restarts, the message **Dictionary space exhausted at number words and Trying again** is displayed. When the dictionary is built, the message is **dictionary built**.

28.4.1.5 Messages

The following is the alphabetized list of the output written by ReduceBen to the Executive window. Most of the messages describe the state of the computation; some are error messages.

Building dictionary . . .

The swap units in the boot file are being sorted by virtual page number.

Data file and client do not match!

CoPilot has discovered a disparity between the client being debugged and the input data file.

... dictionary built

The dictionary of correspondences between virtual page number and swap unit name has been built.

Dictionary space exhausted at *number* words. Trying again ...

The space for the dictionary was not large enough. The space is made larger and another attempt is made. *number* indicates the old size of the space in pages, not words, as the message indicates.

Empty input file

The input file contained no data. No data is written into the output file.

End of input file

The end of the input file has been encountered.

**** File has wrong version number**

The input file was written by a version of Ben that is incompatible with the current version of ReduceBen. Two lines follow that show what the two version numbers were. ReduceBen will terminate after cleaning up.

!Input file not found in root directory

The specified input file does not exist on the designated volume.

!Input not available: *filename* !Output file not available: *filename*

ReduceBen encountered problems acquiring the specified file.

!Input file too long: *file-name*

The input file is too large to be processed.

Insufficient space on volume

There was no more space to construct the dictionary, or write the output file on the volume. Program execution terminates.

No symbols for *module-name*

CoPilot couldn't find the symbols for the designated module.

Number of input items: *number*

Indicates the number of input items read.

Reading input data . . .

Indicates the program has finished initialization and is starting to read the input file.

!Volume not found: *volume-name*

The specified or assumed volume does not exist.

28.4.1.6 Cleaning up

After you have analyzed the log data, you can delete the log file from the client volume by doing

```
>SET Module context: BenImpl  
> DeleteLogFile[] --(note the leading space)
```

or

```
> BenImpl$DeleteLogFile[] --(note the leading space)  
>
```




Statistics

The Statistics tool gathers statistics about Mesa source and object files, such as number of characters, frame size, etc., and writes them to a file.

29.1 Files

Retrieve **Statistics.bcd** from the **Tools >** subdirectory of the Release directory.

29.2 User interface

Statistics runs in the Executive. Its command line format is

```
>Statistics filename1/switches ... filenamen/switches
```

Output from Statistics is sent to **Statistics.stats** by default, but can be directed to another file with the **/o** switch.

29.2.1 Switches

Statistics recognizes the following switches:

- b** produce bcd statistics, that is, code bytes, frame size, ngfi, nlinks, code pages, and symbol pages (default).
- c** command: use *filename_i* not as the name of a file, but as a sequence of switches (e.g., **s/c** prints a subtotal of all statistics gathered up to this point).
- h** print heading (default).
- m** produce source statistics, that is, chars and lines (default).
- o** direct output to **rootname.stats**, where **rootname** is the specified file name (*filename_i*) with any extension removed.
- s** print subtotal.

t print total.

x "Management" statistics (i.e., chars, lines, code bytes, and frame sizes).

29.3 Types of statistics

Statistics generates the following information:

chars	the number of characters in the source file.
lines	the number of lines in the source file.
code bytes	the number of bytes of code in the object file.
frame size	the size of the global frame of the module (in words).
ngfi	the number of global frame table slots needed by the module (one slot for every 32 procedures or signals).
nlinks	the number of items imported into the module.
code pages	the number of pages of code in the object file (one page is 256 words).
symbol pages	the number of pages of symbol table in the object file (one page is 256 words).

29.4 Example

The following command line will generate the output shown below:

```
Statistics CPSyms Actions ComData s/c CPSwap DIHot DIMath t/c
```

```
Mesa Statistics Package 11.1 of 3-Oct-84 17:13
Statistics as of 4-Oct-84 14:18
```

	chars	lines	code bytes	frame size	ngfi	nlinks	code pages	symbol pages
CPSyms	1731	62						7
Actions	1025	37						6
ComData	1242	57	10	43	1	0	1	8
SUBTOTAL	3998	156	10	43	1	0	1	21
CPSwap	8343	236	1084	16	1	49	3	26
DIHot	24023	779	3234	11	2	81	7	51
DIMath	16339	543	2202	15	2	26	5	27
TOTAL	52703	1714	6530	85	6	156	16	125

Note: Sometimes the program puts an asterisk after the number of code pages for a module. This means that the number of code bytes is very close to a page boundary, and the number of links is such that binding with code links will cause the code to "spill over" into another page.

Mesa Services

Mesa Services help users communicate with remote machines and other users. They comprise the Mail tools, the MFileServer, and the Network executive tools.

IV.1 Mail tools

The mail tools include the MailTool itself, for sending and receiving mail messages; the Mail File Scavenger, for repairing damaged mail files; and Maintain, a tool for maintaining mail distribution lists.

IV.2 MFileServer

The MFileServer allows a workstation to serve as a file server for other workstations. Using MFileServer, any file on a workstation can be retrieved to any other machine. If a host with needed files on it is running MFileServer, other hosts can use the File Tool or FTP to retrieve whatever they need.

IV.3 Network executive tools

The Network executive tools are Chat, Remote Executive, NSTerminal, and TTYTajo. Chat provides TTY emulation as well as interactive communication between workstations. A user can chat with other users running Chat elsewhere on the network. Remote Executive allows remote users to connect to a machine and type commands to it as though they were typing commands to their local Executive. NSTerminal allows users to connect to remote computers using RS232 ports and modems.

TTYTajo is a version of the Tajo environment that runs on a teletype-style terminal without windows for tools.



Mail tools

The **MailTool** is the NS-protocol-based mail reading and sending interface to the mail system. The MailTool allows you to retrieve, read, send, forward, save, move, delete, and answer mail.

If your mail file becomes damaged, you may be able to save it by running the **MailFileScavenger**. The MailFileScavenger can restore the internal structure of your mail file to a consistent state. It copies the damaged mail file into a scratch file as it operates; therefore, you must have enough free disk pages available for this scratch file in addition to the number of disk pages that your damaged mail file already occupies. The MailFileScavenger will warn you if there is not enough room.

Maintain is the NS-based interface to the Clearinghouse database. Using Maintain, you can inspect and modify information in the database about message system users and distribution lists.

30.1 Mail Tool

30.1.1 Files

Retrieve **MailTool.bcd** from the Release directory.

30.1.2 User interface

The MailTool has its own window consisting of a message subwindow, two text subwindows and a form subwindow, as shown in figure 30.1. Information and error messages are posted in the message subwindow. The table of contents for the currently active mail file is displayed in the text subwindow directly below the message subwindow. The form subwindow lists commands for manipulating your mail. The lower text subwindow displays individual mail messages. The name stripe of this window indicates whether there is new mail for the user.

30.1.2.1 Text subwindow–Table of contents

An index of all messages in this mail file, called the Table of contents (or TOC), appears in the upper text subwindow of the MailTool window. Each entry contains header information, which includes the message number, the date it was sent, the name of the sender, and the subject of the message.

The user can have more than one mail file to facilitate the organization of his messages. The *current* mail file is the one whose TOC is displayed and the one to which new messages will be retrieved. Its name is displayed in the **File:** field described below. When the MailTool starts up, it reads the mail file specified by the **User.cm** or **Active.nsMail** if none is specified. The user can change the current mail file by chording and selecting from the **File:** field.

The currently displayed message is indicated by a » character after the date column. Deleted messages are indicated by having a line through their entries in the TOC. Unexamined messages are indicated by the character * in the entry. Messages that are not entirely readable by the MailTool, such as Star documents, are left on the Mail service so that the user may read them using another mail tool (such as Star mail). In this case, the message is marked with an "a" in the TOC to show that an unreadable "attachment" to this message is still on the server.

If a one character selection is made for the first character in a TOC line, then the next character typed will become the "flag" character for that entry. This flag has no semantic meaning to the MailTool, but may be used for whatever purpose the user wishes. For instance, you might mark all those messages you need to answer with the character "A", or you might mark those that are urgent with the character "U".

30.1.2.2 Form subwindow

By making a text selection that spans a number of lines in the Table of Contents, it is possible to select a range of messages. Those messages are said to be the *current* messages. The MailTool uses the current messages as an argument for most commands. If there is no selection in the TOC, the current message is the displayed message.

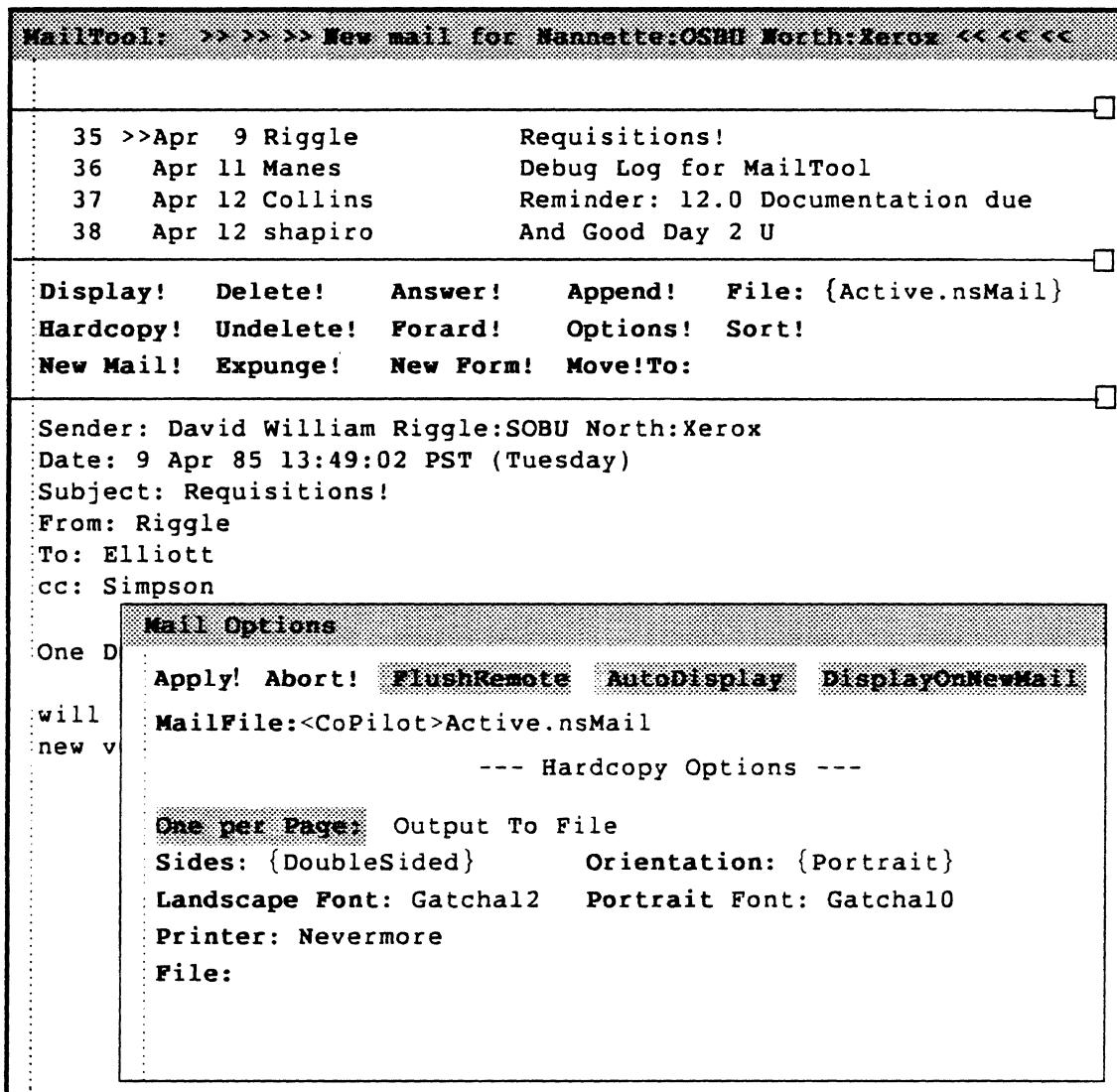


Figure 30.1: The MailTool

Display! displays the first of the current messages if there is a selection in the TOC; otherwise, it displays the next message. If the last command to the MailTool was a **New Mail!**, then the next message is the first message retrieved. If not, the next message is the first undeleted message following the displayed message.

Hardcopy! formats the current messages for printing and either spools them to a printer or writes them into a local file. **Print** will be loaded as needed.

New Mail! retrieves new mail (if any exists) from the user's mailbox to a local mail file. If the **DisplayOnNewMail** option has been set in the User.cm, the first of the retrieved messages will automatically be displayed upon retrieval. Messages that are not entirely readable by the MailTool such as Star Documents, are left on the Mail Service so that the user may read them with another mail tool

(such as Star mail). The readable parts of the message (for example, the header information and MailNote) are copied to the local mail file. If the **Flush Remote** option is set, the message is marked on the server so that it will no longer show up as new mail. It is also marked in the TOC with an "a" to indicate that an unreadable "attachment" to this message is still on the server.

Delete!

marks the current messages for deletion, indicating this by drawing a line through their entries in the TOC. Messages are not removed from the message file immediately, but only when expunged (see **Expunge!** below), after which there is no way to restore them. If a message has an attachment, deleting has no immediate effect on the attachment; the local part in the mail file is marked for deletion, but the attachment remains on the server intact. Before deleted messages are expunged, they may be restored by the **Undelete!** command. Messages without attachments are permanently deleted whenever you either deactivate the MailTool, change the current mail file, or invoke **Expunge!**. An **Expunge!** of a message with an attachment will first delete the attachment from the mailbox. If this is successful, the message will then be expunged from the mail file. Deactivating the MailTool or changing mail files does not affect messages with attachments: they remain in the mail file, marked as deleted.

Undelete!

restores the current messages marked for deletion.

Expunge!

permanently removes messages marked for delete from the mail file and destroys attachments for those messages. Any messages with attachments that are marked for deletion will be deleted from the user's mailbox. Once a message has been expunged, it cannot be restored. *The logged in identity of the user must be the same at expunge time as at retrieve time.* Attachments have associated with them the name of the logged in user at retrieve time. If this identity is different at expunge time, the MailTool will not allow the message to be removed. For instance, if two people retrieve mail to the same mail file, neither of them will be able to expunge the other's messages which have attachments because their own logged in identities do not match the identity stamped on the other's attachments. If you get two copies of a message with an attachment, do not delete one of the copies and expunge before you retrieve the attachment. Expunging will delete your only copy of the attachment.

Append!

inserts the current selection at the end of the mail file and creates a TOC entry for it. The result looks as if the new message were retrieved using "**New Mail!**". This can be used to extract a forwarded message so that it may be answered with the "**Answer!**" command, or to insert a comment into the mail file at an arbitrary location by setting the "**Date:**" field of the comment

appropriately, and following the "**Append!**" command with the "**Sort!**" command.

Forward! produces a SendTool form containing a message body that is a copy of the current message and header fields that can be filled in by hitting the "**Next**" key

New Form! produces a blank SendTool form with header fields that can be filled in by hitting the "**Next**" key.

File: {**Active.nsMail**, ...} is an enumerated item which indicates the current mail file (i.e. the file where new messages will be retrieved to and whose TOC will be displayed). You may choose a different message file to be current by selecting from the menu under this item. Only .nsMail files will be shown, and if there are duplicates in the search path, only the first will be found. The default mail file can be set from the User.cm or from the Options window.

Options! activates the Options window.

Sort! sorts the mail file by the date and time each message was sent.

Move! moves the current messages to the mail file named in the **To:** item. This feature allows you to better organize your messages for easy reference. The extension .nsMail will be assumed if there is no period in the name.

Warning: Any selection in the TOC will be cleared if you edit the **To:** field; you must fill in that field before selecting the messages to be moved. If you are merely moving a displayed message, this problem does not occur.

Warning: You cannot move messages with attachments from one mail file to another unless you confirm the delete of those attachments. Messages with attachments are intended to be read by some other mail reading tool (such as Star mail). If you want the message bodies after you have read them with another tool, remail the mail notes to yourself. You will then be able to move them to another mail file.

To: contains the name of the mail file that is the destination for **Move!** The extension is defaulted to .nsMail. You can also fill in this field by pressing **MENU** and selecting a name from the currently existing .nsMail.

ExpandPvtDLs: (expand private distribution lists) is a Boolean that is currently unimplemented. It will eventually enumerate the members of private mailing lists in the message header so that the message may be answered more easily.

30.1.2.3 Options window

The Options window contains the following items. For most options default initial values may be specified in the MailTool section in **User.cm**.

Apply! causes the fields in the Options window to take effect and closes the Options window.

Abort! closes the Options window without making any changes.

Flush Remote is a Boolean that allows you to retain a copy of your new mail on your mail server. Normally, when you get your new mail, it is completely removed from the mail server, with no copy left. Sometimes you wish to keep a copy on the server, such as when you are reading your mail while using someone else's workstation. To keep a copy on the mail server, turn off the **Flush Remote** Boolean. This must be done before you invoke **New Mail!** Flush Remote defaults to **TRUE**.

AutoDisplay is a Boolean that, if **TRUE**, causes the next message to be displayed when the current message is deleted or moved. The default is **FALSE**.

DisplayOnNewMail is a Boolean that, if **TRUE**, causes the first retrieved message to be displayed after a **New Mail!** command completes. The default is **FALSE**.

Mail File: names the mail file you wish to work with. This file becomes the current mail file when you invoke **Apply!** The extension is defaulted to **.nsMail**. You can also fill in this field by pressing **MENU** and choosing the name from the currently existing **mail** files. If you invoke **Apply!** when the **Mail File** field is blank, the value defaults to **Active.nsMail**.

-- Hardcopy Options --

One Per Page is a Boolean that, if **TRUE**, will cause each message to start on a separate page. The default is **TRUE**.

Output To File is a Boolean that, if **TRUE**, will cause the output from **Hardcopy!** to be written to a file instead of being spooled to a printer. The default is **FALSE**.

Sides: {PrinterDefault, SingleSided, DoubleSided} is an enumerated item that tells the printer whether to do two-sided printing or not. If the printer does not support two-sided printing, this option is ignored. The default is **SingleSided**.

Orientation: {Portrait, Landscape} is an numerated item that specifies the orientation of the output. **Landscape** output is two columns per page; **Portrait** is one. Default is **Portrait**.

Landscape Font: **Portrait Font:** are two fields to indicate which fonts to use when messages are printed. The default font when printing in **Portrait** orientation is Gacha6; for **Portrait**, Gacha8.

Printer: is a tag specifying the name of the interpress printer where hardcopy will be sent.

30.1.3 The MailTool via the Executive window

The MailTool.~ command can change the current mail file, start a retrieval of new mail or change the state of the MailTool window. The general form is:

>MailTool.~ filename/switches

filename should identify an existing message file. Legal switches are:

- a activate MailTool.
- n retrieves new mail
- i inactivates MailTool (causes an expunge).
- t makes MailTool tiny.

30.1.4 Send Tool

The Send Tool is used to send messages. A blank mail form is created by either invoking **New Form!**, **Answer!**, or **Forward!** in the MailTool window or invoking **Another!** in an open Send Tool window. The Send Tool has a message subwindow, a form subwindow, and a text subwindow. **SendTool.bcd** is also available independently from the Tools subdirectory of the Release directory.

30.1.4.1 Form subwindow

Five items are always available in the form subwindow. A sixth, **Deliver!**, appears after the message has been edited.

- | | |
|-----------------|--|
| Another! | creates another instance of the Send Tool. |
| Destroy! | destroys this instance of the Send Tool. If the form has been edited but not sent, this command requires confirmation. If there are no instances of a SendTool on the inactive list, this command will merely deactivate the current instance. |
| Reset! | leaves the SendTool window open but clears it of inserted text. If the form has been edited but not sent, this command requires confirmation |
| Put! | writes the contents of the SendTool window to the file named in the File: field. |

Get!	replaces the contents of the SendTool window with the contents of the file named in the File: field. If the form has been edited but not sent, this command will require confirmation.
File:	is a string item used to hold the name of the file used in the Put! and Get! commands.
Invalid OK	is a Boolean that allows you to send a message containing invalid recipients. The default is FALSE .
If Need Reply-To	is an enumerated item that allows you to control what happens if the message should have a Reply-To field, but does not. A message should have a Reply-To field <i>if it includes a public distribution list</i> in the To: or cc: fields in order to limit those who automatically receive answers to the message. When you press MENU over If Need Reply-To , the following choices will appear:
don't send	prevents the message from being sent and puts in the message subwindow the line Add Field to message header: Reply-To: value .
add to form	adds the necessary field to the message and sends it.
send anyway	allows the message to be sent, even if the Reply-to: field is needed.
	This field is defaulted to don't send unless NeedReplyTo: value is specified in the User.cm (where value is Don'tSend , SendAnyway , or AddToForm).
Deliver!	sends the mail to the recipients indicated in the To: and cc: lines of the message. This command is available only when the body of the message has been edited. After the delivery has taken place, the Deliver! command is replaced by the message Delivered . To send a message, you must be logged in.
SendAs:	is an enumerated item that provides three ways of sending a message: MailNote, Text, or MailNote with an attachment. A MailNote is the simplest way to exchange mail between environments. A Text message requires the user in another environment to convert the message before reading it, and its delivery incurs a little more overhead than the MailNote, but it does allow for long messages (a MailNote is currently limited to 8000 characters). A MailNote with an attachment includes material not entirely readable (such as Star documents) by the MailTool. Presently, there are no facilities provided by the MailTool either to convert XDE files to other formats or to forward attachments.

30.1.4.2 Text subwindow

The text subwindow contains the text of the message, including a header part and a message body part. The header part includes **Subject:**, **To:**, **Reply-To:**, and **cc:** fields that are used by the message system to direct the message when it is sent.

30.1.4.3 Subject: field

The topic of your message goes in the **Subject:** field. The topic should express the content of your message so that interested people will take the time to read the message, but uninterested people can delete it without reading it. For example, if your message contains ideas for improving the MailTool, the topic might be "Suggestions: improving MailTool," *not* "Suggestions."

30.1.4.4 To: field

The **To:** specifies who is to receive your message. A recipient entry has three parts (name, domain, and organization) separated by colons. It may be the name of an individual or an NS-based distribution list (for example, **Secretaries:OSBU North:Xerox**). Only those groups and entities with mailboxes are valid recipients. A domain is simply a device for grouping related names and most messages are sent within a single domain. The MailTool allows you to omit the domain name for recipients who are in your same domain. For example, someone in the domain for the Palo Alto area, say **Someone:OSBU North:Xerox**, could send a message with the following acceptable message header:

```
Subject: Demonstration of recipient naming
To: Person1, Person2
cc: Person3, FarAwayPerson1:OSBU South
```

The MailTool assumes that names lacking domains are in the sender's domain, which in this case is **OSBU North**. Since **FarAwayPerson1:OSBU South** explicitly includes the domain, **OSBU South** is used by the MailTool. In this case, the message will go to **Personnel:OSBU North:Xerox**, **Person2:OSBU North:Xerox**, **Person3:OSBU North:Xerox** and **FarAwayPerson1:OSBU South:Xerox**.

Public Distribution Lists:

NS-based public distribution lists are groups in the Clearinghouse consisting of mailbox names. No special delimiter is needed to tell the MailTool you're mailing to a distribution list rather than an individual. Using such a name as the recipient of a message causes the message to go to all the individuals included in the group. For example, the line

```
To: Secretaries:OSBU North:Xerox
```

will cause the message to be delivered to all the Xerox secretaries in Palo Alto.

The public distribution lists for each domain are stored in the Clearinghouse. They are typically maintained by the individuals who "own" the lists. You can have yourself added to appropriate lists by contacting the owner (in the case of closed distribution lists) or by using the Maintain program. While you are able to use any public distribution list from any domain in delivering any message, *you should think very carefully about your choice of*

message and list so as not to bother recipients. Check with experienced users to find out which lists should be used for which kinds of messages.

Private Distribution Lists:

A private distribution list is a file which resides on your local work station and contains legal (in the Clearinghouse sense of the word) recipient names separated by carriage returns <CR>. Private distribution lists may be indicated in the **To:** field by suffixing the name of the file with an asterisk (*). The basic form is:

Filename.extension*

If you fail to include the extension in the filename, the MailTool will assume a *.dl* extension and look for the corresponding file. It is also possible to use files stored on remote file servers as private distribution lists. The syntax for naming them is:

[host]<directory>subdirectory>..>filename.extension*

Remotely stored private distribution lists are appropriate if a small group of people want to share the use of the list.

30.1.4.4.1 Reply-To: field

The **Reply-To:** field works in conjunction with the **Answer!** command. **Answer!** initializes a message form so as to reply to the message selected in the Table of Contents. If the message being answered contains a **Reply-To:** field in its header, then only those recipients in the **Reply-To:** field will be included in the **To:** field constructed by **Answer!** The **Reply-To:** field thus limits those who automatically receive answers to messages. A recipient of such a message can change the recipient fields constructed by **Answer!**.

30.1.4.4.2 cc: field

The **cc:** field identifies others who are to receive your message. Names should be separated from each other by commas. When you send your message, these people will automatically receive it along with the person(s) specified in the **To:** field.

30.1.4.4.3 Message body

The message body (the actual content of the message) follows the header. There must be an empty line between the last field in the header and the message body.

30.1.4.5 SendTool via the Executive window

The **SendTool.~** command is used to bring up an instance of the **SendTool**.

>SendTool.~recipient/switch

The name <recipient> will be placed into the **To:** field of the new SendTool. If the switch 'f' is supplied then the name will be treated as a file from which a form will be loaded in place of the standard empty mail form.

30.1.4.6 User.cm entries

Some MailTool parameters can be set from the User.cm. These are listed below with sample values.

```
[MailTool]

TOCLines: 6                                -- number of initial lines
                                                displayed in the table of
                                                contents (TOC)

MailFile: Active.nsMail                      -- name of initial mail file

DisplayOnNewMail: FALSE                      -- do an automatic Display!
                                                after mail is retrieved.

FlushRemote: TRUE                           -- flush remote mail after
                                                retrieval

MessageFont: LaurelFont.strike              -- if omitted, the built-in
                                                Tajo font is used

TOCFont: Gacha8.strike                      -- if omitted, the built-in
                                                Tajo font is used

AutoDisplay: FALSE                          -- if TRUE, next message is
                                                displayed when current
                                                message deleted

NeedReplyTo: AddToForm                      -- choose from DontSend,
                                                SendAnyway, AddToForm
                                                (section 26.4.1)

OutputToFile: FALSE                         -- if TRUE, output is written to
                                                a file instead of the
                                                appropriate printer

OutputFile: MailTool.interpress             -- name of output file to be
                                                used when OutputToFile is
                                                TRUE
```

You can also specify the printing characteristics to be used by the **Hardcopy!** command. If no printing entries are made in your MailTool **User.cm** section, the values from the **[Hardcopy]** section will be used. Refer to the Print chapter for further information about the different entries.

SeparatePages: FALSE	-- if TRUE, each message will start at the top of a new page
Sides: SingleSided	-- controls whether the printer should do two-sided printing or not
InterPress: Nevermore	-- name of the default InterPress printer to use
LandscapeFont: Gacha6	-- name of the default font to use when in landscape mode
PortraitFont: Gacha8	-- name of the default font to use when in portrait mode
Orientation: Landscape	-- default output orientation
PrintedBy: \$	-- name to place on the banner sheet when output is printed. The special token "\$" indicates that the current login name should be used

Several SendTool parameters may also be set from the **User.cm**. These are listed below with sample values.

[SendTool]	
Font: LaurelFont.strike	-- if omitted, the built-in Tajo font is used
NeedReplyTo: DontSend	-- choose from DontSend, SendAnyway, AddToForm. This is used by those instances not brought up through the MailTool

30.1.4.7 Trouble shooting

If you find that the MailTool has trouble distinguishing your password (for example, you receive the message "Invalid password" upon invoking **New Mail!**), check to see that your workstation has the correct time. You may need to reset your clock.

30.2 MailFileScavenger

30.2.1 Files

Retrieve **MailFileScavenger.bcd** from the Release directory.

30.2.2 User interface

MailFileScavenger runs in the Executive window. To invoke it, type **MailFileScavenger MailFile.nsMail**, where **MailFile.nsMail** is the name of the mail file to be scavenged (if you type a name without a period, **.nsMail** will be added to the name automatically). Terminate the name with **RETURN**. MailFileScavenger will proceed to copy your mail into its scratch file, printing out the number of every fifth message as it is processed.

When anomalies are detected in your mail file, MailFileScavenger will print out a short message such as **Message 53: existing count was 231 bytes too small**. This message indicates that the formatting information present in the mail file used to distinguish individual messages was inconsistent with what MailFileScavenger believes to be distinct messages.

When MailFileScavenger is finished, it is a good idea to check any messages it complained about. These messages may be missing several characters or be malformed in other ways. You should also check neighboring messages—some of the characters in those messages might really be part of other messages.

After MailFileScavenger has finished copying and reformatting your mail into its scratch file, it will pause and ask if it should copy that file back into the original mail file. If there are not many error reports, type **Y** to confirm. MailFileScavenger will copy the scavenged mail file back into the original mail file, delete the scratch file, and quit. You may then invoke the scavenged mail file in your MailTool Options window. However, if there have been many error reports, you might want to copy the original file before allowing the MailFileScavenger to scavenge your file. To do this, abort the command with **N**, copy the file, then run MailFileScavenger on the copy.

The mail file that MailFileScavenger produces should give you a readable mailfile, i.e., one that the MailTool will not complain about. However, this mail file may have messages that are fragments of messages in the original file and/or duplicate messages. If you copied the original file before running the MailFileScavenger, you can compare the scavenged version to the original in order to determine if any text was lost. If you edit the scavenged mail file, you will have to run the scavenger again.

30.3 Maintain

Maintain is the NS-based administrative interface for the Clearinghouse database. Using Maintain, you can inspect and modify information in the data base about message system users and distribution lists. This data base is described in this section.

30.3.1 Files

To run the Maintain program, retrieve **Maintain.bcd** from the Release directory.

30.3.2 User interface

Maintain interacts through a message subwindow, a form subwindow, and a log maintained in a file subwindow. By executing commands you can manipulate items in the Clearinghouse database.

30.3.2.1 Message subwindow

The message subwindow is used for feedback and to show progress in the completing of the command invoked by the user.

30.3.2.2 Form subwindow

The form subwindow contains the fields and commands used to invoke the various functions that are available through Maintain. In general the top half of the form contains those items used to manipulate Clearinghouse groups, and the bottom half contains items used for changing parameters associated with a particular individual.

Level is an enumerated item that governs which commands are available to you. The value may be **normal**, **owner**, or **administrative**. The following subsections discuss what is available at each level.

30.3.2.2.1 Group commands: normal level

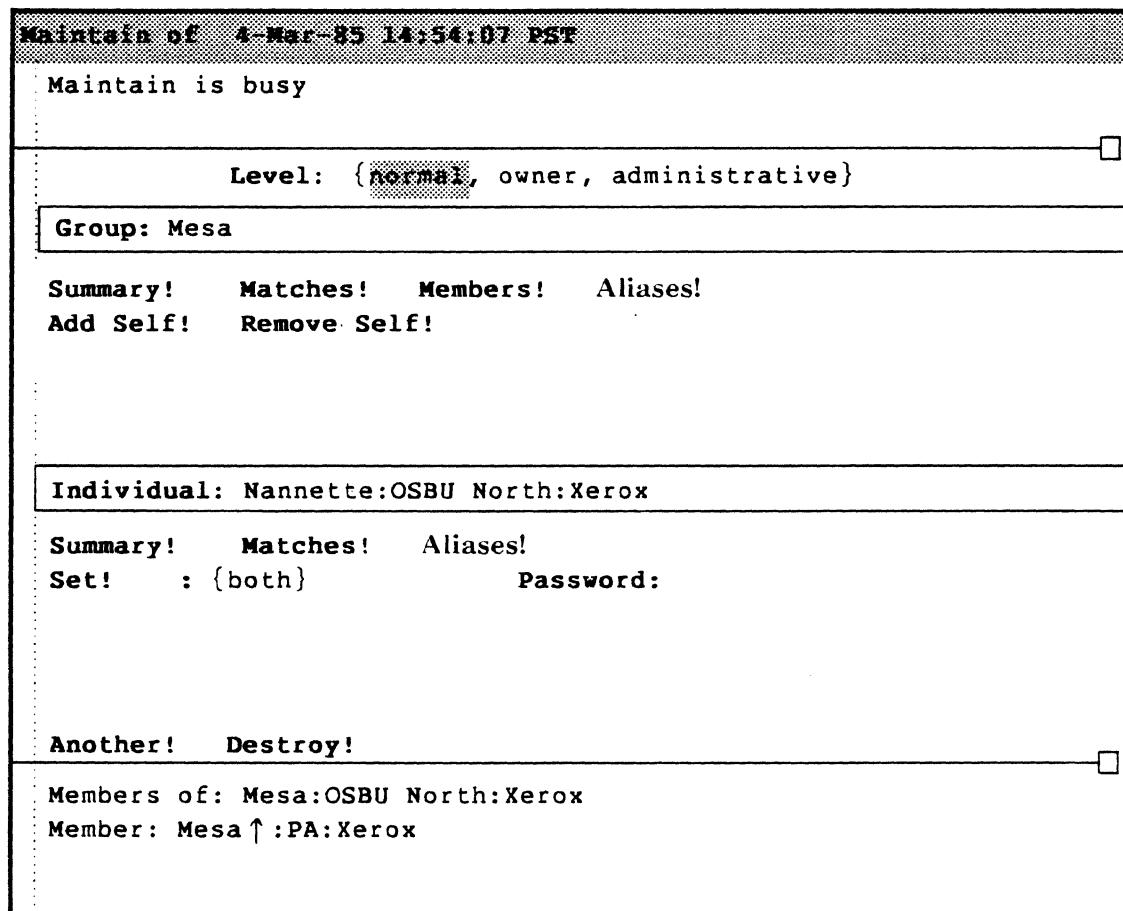


Figure 30.2: Maintain tool window (normal level)

- Group:** contains a list of Clearinghouse distinguished names and patterns that is the argument to each of the commands that acts on groups. If the domain or organization is not specified, the defaults from the Profile are used. (See Profile tool.)
- Members!** lists the members of the group in the **Group:** field in the file subwindow.
- Summary!** shows the user-visible components (the distinguished name, remark field and number of members) of the group in the **Group:** field .
- Aliases!** shows the distinguished name and any aliases for that entry.
- Add Self!** adds the currently logged in user to the group in the **Group:** field.
- Remove Self!** Removes the currently logged in user from the group in the **Group:** field.

- Individual:** contains a Clearinghouse name that is the argument to each of the commands that acts on individuals. This field is initialized to the currently logged in user's distinguished name.
- Password:** contains the new password for the individual in the **Individual:** field.
- Summary!** shows in the file subwindow the user-visible components (the distinguished name, user remark, and file service) of the individual in the **Individual:** field.
- Set! Password:** sets the password of the individual in the **Individual:** field to be the value in the **Password:** field.

30.3.2.2 Group commands: owner Level

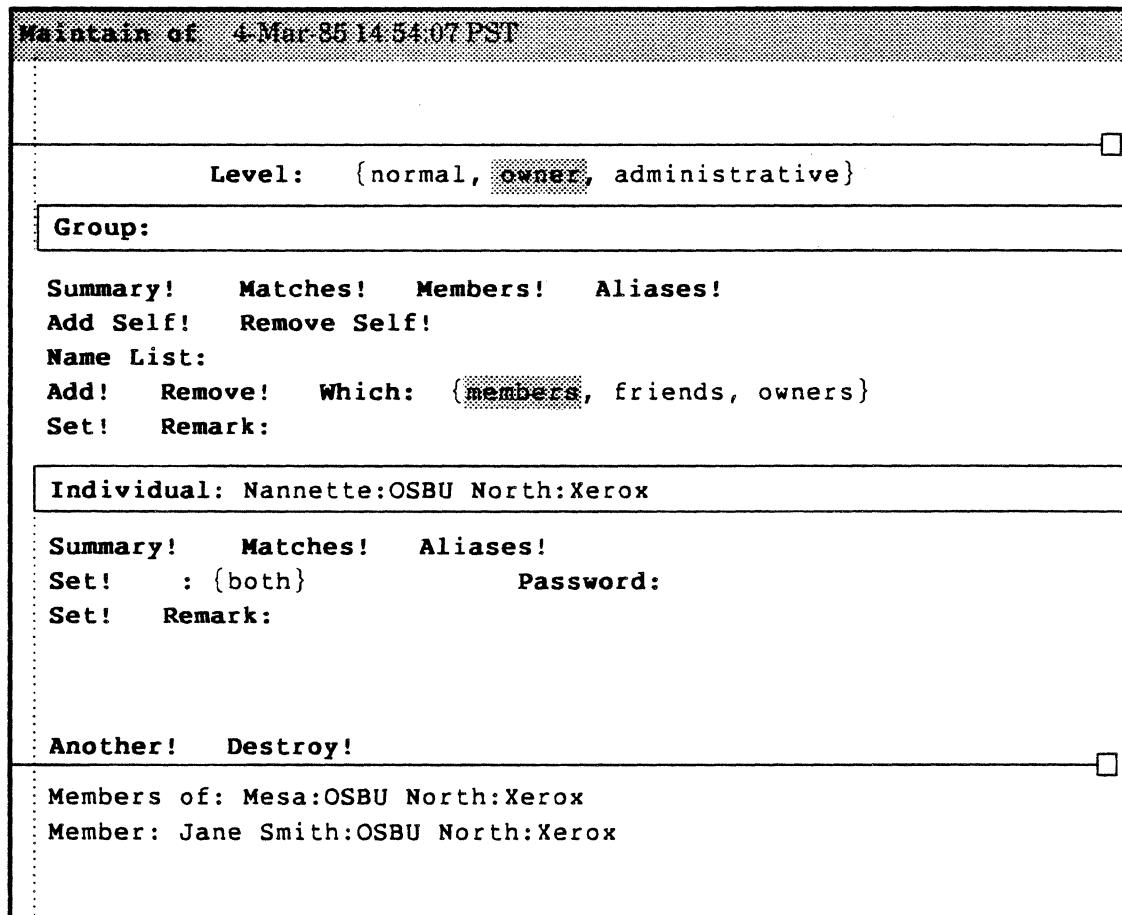


Figure 30.3: Maintain tool window (owner level)

All the commands available at the normal level are also available at the owner level. The following additional group-related commands and field are available.

- NameList:** contains a list of Clearinghouse patterns that are to be added to or removed from the group in **Group:**. Aliases in

this list are resolved to the corresponding distinguished name.

Which: determines whether the elements in **NameList:** refer to members, friends, or owners of the list (see 30.3.3.1 Rules for accessing the data base).

Add! adds the elements in **NameList:** to the group specified in by **Which:**. You should set the friends before setting the owners if you will not be an owner. Once the owner's list is set, the members', friends', and owners' lists cannot be changed except by an owner. Also note that the friends' and owners' lists both default to the list of domain administrators for the group's domain.

Remove! removes the elements in the **Name List:** field from the group in the **Which:** field.

Set! Remark: sets the group remark to the text in **Remark:** which is typically a description of the group.

30.3.2.2.3 Group commands: administrative level

Alias: contains a list of aliases to be added or removed from the aliases of a group.

Add! adds the aliases to the database.

Remove! removes the aliases from the database.

Details! gives the distinguished name, the aliases, the group remark, and the lists of members, owners, and friends

Create! creates a group. The remark is initialized to the text in **Remark:**. If **Which:** is members, the members for the group is initialized to the names in **Name List:**. For a long list of members, this is much faster than adding the members after the group has been created.

Delete! deletes a group.

30.3.2.2.4 Individual commands: normal level

Individual: is a list of Clearinghouse distinguished names and patterns which is the argument to each of the commands that act on individuals. If the domain or organization is not specified, the defaults from the Profile are used.

Summary! gives the distinguished name, user remark and file service.

Matches! list each individual whose name contains a match of the pattern (which may contain wild cards) in **Individual:**

Aliases! gives the distinguished name and aliases.

Set! Password: sets the password for the individual. The enumerated type determines whether strong, simple, or both passwords are set.

30.3.2.2.5 Individual commands: owner level

Set! Remark: set the remark to be the text in Remark:

30.3.2.2.6 Individual commands: administrative level

Add! Remove! Mailbox: not implemented.

Alias: contains a list of aliases to add or remove from the aliases of an individual.

Add! adds aliases for an individual

Remove! removes aliases for an individual

Detail! gives detailed information on individuals including the distinguished name, the aliases, the user remark and the file service.

Create! creates an entry in the Clearinghouse for an individual. The remark is initialized to the text in Remark:

Delete! deletes an entry in the Clearinghouse for an individual.

30.3.2.2.7 Tool commands

Level is an enumerated item that governs which commands are available at any given time.

Anyentry is a Boolean which determines which Clearinghouse entries are available using Maintain. If it is false, then only entries with the primary properties userGroup and user are available. If it is true, all Clearinghouse entries are available.

CheckNames is a Boolean which when true maps aliases to distinguished names and expands patterns.

Another! creates another instance of Maintain.

Destroy! destroys current instance of Maintain. Maintain can also be unloaded.

UseBackground

is a Boolean which when true runs commands in the background. If false, it holds the notifier and runs commands synchronously in the foreground.

30.3.2.3 File subwindow

The result of executing the command is logged to the file subwindow.

30.3.3 The Clearinghouse data base

All items in the Clearinghouse data base are identified by a fully-qualified name. A Clearinghouse name has three components, *Name:Domain:Organization*. For example, Randall:OSBU North:Xerox and Secretaries:OSBU North:Xerox, are fully-qualified names for an individual and a public distribution list, respectively.

See the *Services 8.0 Programmer's Guide* for a more complete description of the Clearinghouse.

30.3.3.1 Rules for accessing the data base

Any logged-in user of Maintain can invoke any command that reads information out of the data base. Changes to the data base are controlled by the owners and friends lists for a group. The rules for controlling the data base are as follows:

Individuals can set the password and set the connect site of their own entries.

Friends of a group can add and remove their own names from the membership list of that group.

Owners of a group can add and remove owners, friends, and members for the group. An owner also can set the remark.

MFileServer

The MFileServer provides the server side of communication using the XNS Filing protocol. The XNS Filing protocol involves two parties: a *user* who makes requests and a *server* who honors (or rejects) them. The MFileServer allows other machines (using the FileTool or FTP) to connect to your machine and store, delete, list, or retrieve files.

31.1 Files

Retrieve **MFileServer.bcd** from the Release directory.

31.2 User interface

The MFileServer window has a form window containing variables that can be used to control its actions:

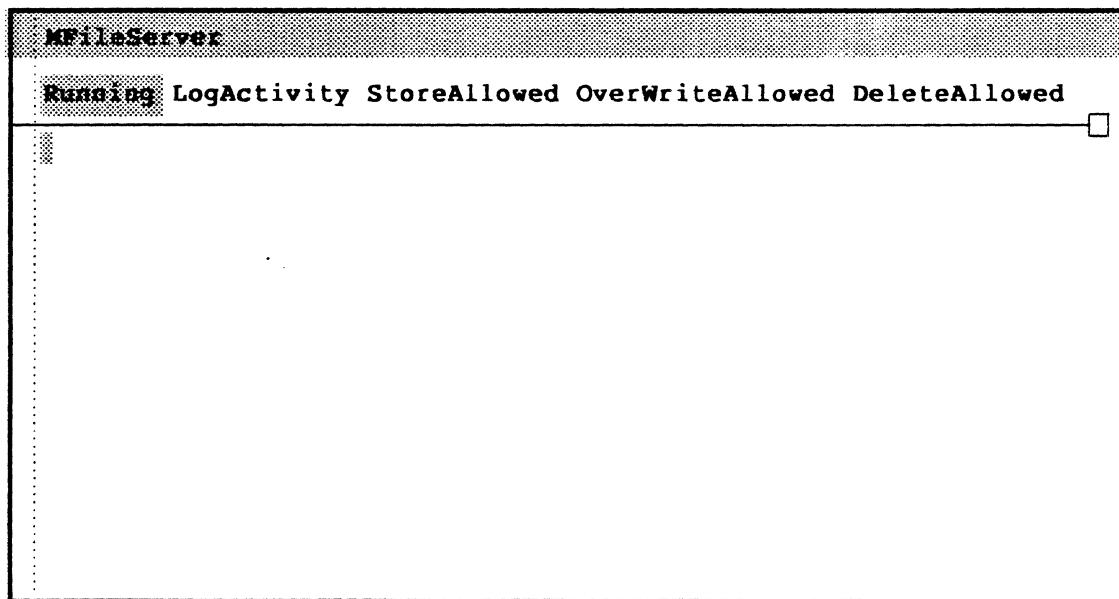


Figure 31.1. MFileServer window

31.2.1 Form subwindow

Running	is a Boolean that controls whether the server will accept connections at all. Changing it to FALSE will disallow future connections, but will not terminate current connections (default = TRUE).
LogActivity	is a Boolean that controls whether MFileServer logs its activity in its subwindow. When it is FALSE , no log is kept (default = TRUE).
StoreAllowed	is a Boolean that controls whether store operations are allowed. When it is FALSE , files may be retrieved, but may not be stored (default = FALSE).
DeleteAllowed	is a Boolean that controls whether delete operations are allowed. When it is FALSE , files may not be deleted (default = FALSE).
OverwriteAllowed	is a Boolean that controls whether existing files may be modified. When it is FALSE , new files may be stored, but old files may not be overwritten, deleted, or renamed (default = FALSE).

Making the tool tiny does not affect the state of the server (in particular, it does not disable the server). Making the tool inactive aborts all its current connections and turns the server off so that it will not accept any new connections.

31.2.2 Executive commands

The MFileServer registers the command `MFileServer.~` with the Executive. If the command is invoked with no arguments, it prints out the current state of the MFileServer's variables. The command can be used to change the variables of the MFileServer by taking a series of arguments of the form *variable/value*. All values must be either *on* or *off*. Hence the following command line sets the value of **StoreAllowed**, **OverWriteAllowed**, and **LogActivity**.

```
>MFileServer StoreAllowed/on OverWriteallowed/on LogActivity/off
```

31.3 User.cm entries

The MFileServer initializes the variables in its form window from the `[MFileServer]` section of your `User.cm`. The window box of the tool, its tiny position, and its initial state are also controlled by entries in this section:

<code>Running:TRUE</code>	-- default value of <i>Running</i>
<code>LogActivity:TRUE</code>	-- default value of <i>LogActivity</i>
<code>StoreAllowed:FALSE</code>	-- default value of <i>StoreAllowed</i>
<code>DeleteAllowed:FALSE</code>	-- default value of <i>DeleteAllowed</i>
<code>OverwriteAllowed:FALSE</code>	-- default value of <i>OverwriteAllowed</i>

```
WindowBox: [x: 362, y: 628, w: 662, h: 150]
-- location of tool's window box

TinyPlace: [x: 720, y: 778] -- location of tool's tiny box

InitialState: Active -- initial state of tool
```

31.4 Operational notes

When the remote directory is specified as empty angle brackets, "<>", MFileServer uses the search path. (The remote directory refers to the directory field of the FileTool or the directory specified in the FTP command line.) For files not on the search path, the directory must be explicitly stated.

Storing and retrieving files require a non-empty remote directory.

The workstation running MFileServer must be registered in the Clearinghouse.



Network executive tools

The network executive tools provide ways to communicate with other workstations and terminals on your network. These tools are Chat, NTerminal, Remote Executive, and TTYTajo.

Chat lets you talk to other machines via a teletype style user interface. NTerminal lets you communicate with other machines using terminal emulation (VT100). NTerminal also allows communication with dialup facilities available on your network (CIU and ECS facilities). The Remote Executive allows remote workstations to use the facilities of an XDE via Chat. TTYTajo is a server which has the Remote Executive function built into the bootfile.

32.1 Chat

Chat provides a simple TTY-emulation capability in the development environment, similar to Telnet in the DARPA realm. It runs on a standard Tajo or CoPilot bootfile.

Chat has three modes of operation. First, with a Remote Executive on the other end, Chat allows one-way communication with other XDE machines. The second mode allows communications with the Interactive Terminal Service (ITS). The ITS is a network service that allows you to read and send mail or to create and store files. Finally, Chat's Remote System Administration mode allows monitoring and administration of servers such as the Clearinghouse, file, and print servers.

32.1.1 Files

Retrieve **Chat.bcd** from the Release directory.

32.1.2 User interface

Chat registers the command "Chat.~" with the executive. The simplest form of the command is:

```
>Chat.~
```

This command either activates an inactive Chat if there is one, or it creates a new one if not. The full form of the Chat command is:

```
>Chat.~ [host]/[switch]
```

host tries to open a connection to that host (see the **Connect!** command below). *switch* tells what type of host *host* is. The values of *switch* are:

s	-- Remote System Administration
i	-- ITS
e	-- Remote Executive

After you type this command to the Executive, a Chat tool window will appear. Chat's tool-style interface has a message subwindow, a form subwindow, and a TTY subwindow.

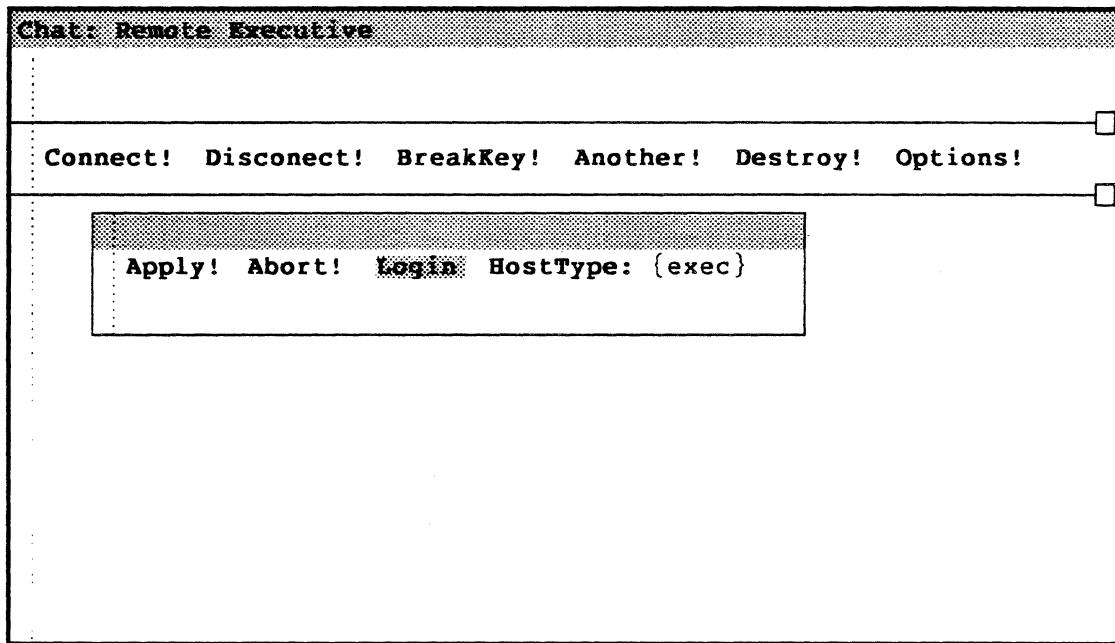


Figure 32.1: Chat

32.1.2.1 Message subwindow

The message subwindow is used for one-line messages. Chat tries to make sure that the last message in this window agrees with the present state of the Chat world.

32.1.2.2 Form subwindow

The form subwindow contains several commands:

Connect! using the current selection as a host name or address, **Connect!** tries to open a connection to that host. After a connection has been established, a message to that effect is posted in the message subwindow so you can start typing. As a shorthand for this, typing a host name in the file subwindow followed by **DOIT** takes the last word typed as the host name and invokes the **Connect!** command. Note that the **Connect!** command behaves

slightly differently depending on the values of some of the fields described below.

Disconnect! if there is a connection open, **Disconnect!** deletes the connection for the network stream, collects and throws away the tool's various processes for managing the data stream, and returns the tool to a quiescent state.

BreakKey! simulates a terminal's break character.

Another! starts up another Chat window, using the same options as the current Chat window.

Destroy! destroys the Chat window. No confirmation is required, since you can get another tool window using the exec **Chat.~** command.

Options! creates a Chat options window. The options are:

Apply! sets your chosen options and destroys the options window.

Abort! cancels any changed options and destroys the options window.

LogIn If this Boolean is **TRUE** and both **Profile.User** and **Profile.Password** are non-null, Chat will try to log you in on the remote host using these values. If the Boolean is false, Chat will not try to log you in. The default value is **TRUE**. (You can set this value in the [**Chat**] section of your **User.cm** file. Or, if the **LogIn** Boolean in the Options window is selected, you will be logged in automatically.)

HostType: selects the desired host type (any, **sa**, **exec**, or **its**) from the **HostType:** menu. Then select **Apply!**

32.1.2.3 TTY subwindow

Chat also has a TTY subwindow in which the dialogue with the remote system takes place. When a connection is established, characters sent from one machine to another appear in the TTY subwindow.

An alternate way to connect to a host (rather than using the **Connect!** command) is to type the host name into this subwindow, and hit the **DOIT** key (the one labelled **MARGINS** on the Dandelion keyboard).

32.1.3 Special keys

Chat makes use of the following special keys:

COMPLETE: sends an Ascii **ESC**.

DELETE: sends an Ascii **DEL**.

BS:	sends an Ascii BS (CONTROL-H).
BW:	sends an Ascii ETB (CONTROL-W).
ABORT:	does a Stream.SendAttention on the current connection, in an attempt to simulate the Break key found on some terminals. Note: The RemoteExec uses Break to simulate the ABORT key; to abort an action in a Chat connection to a RemoteExec, you press ABORT .

32.1.4 Chat User.cm

Chat will read the following **User.cm** options:

```
[Chat]
LogIn: TRUE FALSE
HostType: sa any exec its
```

32.2 NSTerminal

NSTerminal allows you to connect to any service exporting the Gateway Access Protocol (GAP). Services that export GAP include the Communications Interface Unit (CIU), the External Communication Service (ECS), the network executive used for remote system administration on all network services, the Interactive Terminal Service (ITS), and the XDE Remote Executive.

NSTerminal provides a capability that is basically the same as VT100 terminal emulation in Star. NSTerminal is more flexible in that it allows you to communicate with any GAP service on the network via the same window. NSTerminal provides terminal emulation for a number of terminal (given below in the User Interface section) including DEC's VT100.

For more information about the services mentioned above, please refer to the *Services 8.0 Programmer's Guide* and the *OS 5.0 System Administration Library*.

32.2.1 Files

Retrieve **NSTerminal.bcd** from the Release directory.

32.2.2 Setting up

Before running NSTerminal, you should be logged in. After doing some initialization, a Chat-style window will appear. NSTerminal will create a file **NSTerminal.cache** the first time it is executed. This file caches clearinghouse entries for ports on the network that you can communicate with. Should new ports be added or changed, the **NSTerminal.cache** file should be deleted (via the FileTool or the Executive) and will be re-created automatically the next time NSTerminal is executed.

32.2.3 User interface

NSTerminal registers the command "NSTerminal.~" with the executive. To create a new instance of the tool, type into the Executive:

```
>NSTerminal.~
```

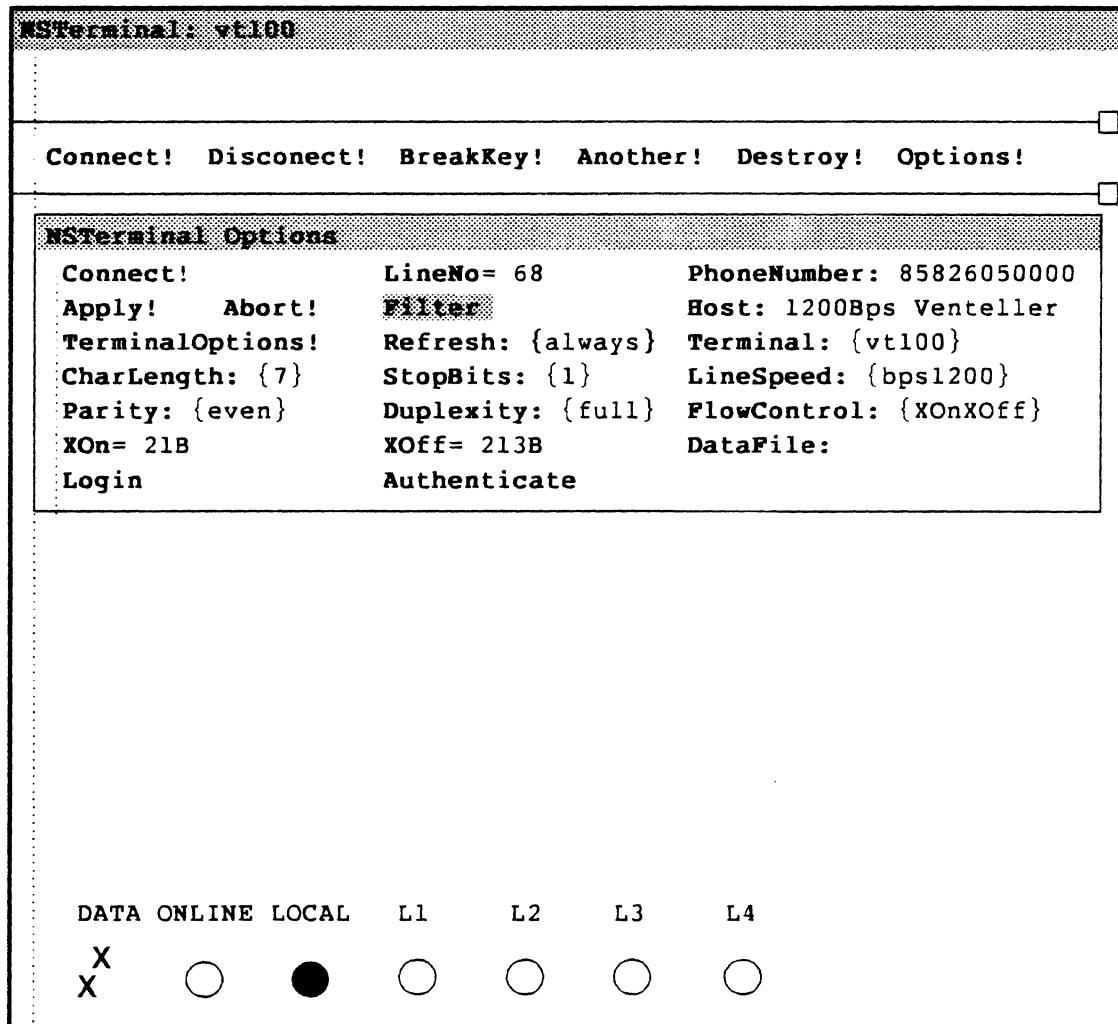


Figure 32.2: NS Terminal

The NSTerminal window has three subwindows, a message subwindow, a form subwindow, and a terminal emulation subwindow.

The message subwindow is used for various one lined messages.

The form subwindow contains the following commands:

Connect! takes the current selection as a host name or address and attempts to open a connection to that host. This command has the same semantics as the **Connect!** command on the NSTerminal Options window (see **Options!** below). The **Connect!** command on this form should only be used if the options are properly set.

Disconnect! will close the connection if there is one open. Closing the connection will collect and throw away the connection's various processes for managing the data stream, and return the tool to a quiescent state.

BreakKey! simulates a terminal's break key.

Another! creates a new NSTerminal window. The new window will use the User.cm default values for it's option window.

Destroy! will destroy the NSTerminal window.

Options! creates a NSTerminal Options window. Using the Options window is the standard way to open a connection to a host. The options that affect only the parent NSTerminal window are:

Connect! will open a connection to the host specified in the **Host:** field. This command will also cause the Options window to be destroyed.

LineNo= takes a numeric value, the line number of the service you want to talk to. Line numbers can be thought of as virtual sockets, and on a given host, a different line number corresponds to a different service. Some well known line numbers include:

32001	Remote System Administration function
32002	XDE Remote Executive function
32003	Interactive Terminal Service (ITS)

PhoneNumber: this field is used when the service that you are connecting to has a dial out function. The phone numbers are entered without any punctuation, ie the number (415) 555-5555 would be entered as 4155555555.

Apply! will set the tool's options to what is displayed in the Options window. The Options window will then be destroyed.

Abort! will reset the tool's options to it's state before the Options window was opened. The Options window will then be destroyed.

Filter will cause NSTerminal to mask out the high bit of every byte before printing the character. This function is useful if the remote host uses seven bit characters with some parity, and your receiving communications unit uses an eight bit no parity option.

Host: is the host name or network address of the service you wish to open a connection to.

TerminalOptions! will create an Options window which will allow you to change the terminal emulator subwindows properties.

Refresh:{}

this enumerated allows the user to specify the way the emulator subwindow will display the incoming characters. The user can specify (via a pop up menu) display modes from display each character has it is received to deferring the painting to a later time. The refresh options are:

<i>always</i>	update screen on every character
<i>never</i>	update only if nothing else is happening
<i>half</i>	force an update when the screen is half invalid
<i>full</i>	force an update when the screen is all invalid

The recommend options is *always*, although when using the *never* options, NSTerminal can handle data transfer rates of 9600 baud continuous. The *never* mode can be used to transfer files to the your workstation since all characters received are stored in **NSTerminal.log**.

Terminal:{}

this item has a pop up menu with the various terminals that can be emulated. The enumerated items represent the following terminals:

addrinfo	General Terminal
adm3	Lear Siegler Adm3
adm3a	Lear Siegler Adm3A
cdc456	Control Data 456
dm1520	Data Median 1520, 1521
gt100	General Terminal 100A
h1000	Hazeltine 1000
h1420	Hazeltine 1420
h1500	Hazeltine 1500
h1510	Hazeltine 1510
h1520	Hazeltine 1520
h2000	Hazeltine 2000
isc8001	Interactive Systems
soroc	Soroc 120
teletec	Teletec Datascreen
trs80	Radio Shack
vc303	Volker-Craig 303
vt100	DEC VT100
vt50	DEC VT50
vt50h	DEC VT50H
vt52	DEC VT52

x820	Xerox 820
other	use the DataFile: terminal

The next eight items on the NSTerminal Options window are only applicable to connections to the local port of an External Communications Service. The Communications Interface Units have these options hard-wired and their values are reflected when a RS232C port is selected from the RS232C Ports pop up menu of the Options window.

CharLength:{}	this item has a pop menu with the different character lengths you can request your host to use.
StopBits:{}	this item has a pop menu with the number of stop bits you can request your host to use.
LineSpeed:{}	this item has a pop menu with the baud rates you can request your host to use.
Parity:{}	this item has a pop menu with the parity options you can request your host to use.
Duplexity:{}	this item has a pop menu with the duplexity options you can request your host to use.
FlowControl:{}	this item has a pop menu with the flow control options you can request your host to use.
xOn=	is the character used to initiate the flow control.
xOff=	is the character used to terminate the flow control.
DataFile:	is a user provided terminal that NSTerminal will emulate when the <i>other</i> option is chosen in the Terminal:{} field. This file contains a finite state automata that represents the character sequence necessary to invoke a terminal action. Further explanation of this file is beyond the scope of this document.
LogIn	if this Boolean is selected, NSTerminal will automatically log you in when you connect to a service.
Authenticate	if this Boolean is selected, NSTerminal will gather credentials to be used in the opening of a connection. This is only necessary when a particular service has an access group associated with the service.

The third subwindow in the NSTerminal window is the terminal emulator subwindow. The emulator subwindow is not a standard Tajo TextSW or TTYSW. Selections can be made using Point and Select to define the boundaries of the selection. There is no selection tracking as in regular text subwindows, and the selection disappears once new text is written to the screen. Selection can be stuffed into other windows using the STUFF (labeled OPEN on the Dandelion) button, and text from other windows can be stuffed into the

emulator subwindow. There are no scrollbars on the emulator subwindow, to see the full context of the window one must grow the window to be large enough. Hitting Adjust in the emulator subwindow will cause the window to become the input focus if it does not already contain a selection. A log is kept in the file **NSTerminal.log**.

At the bottom of the emulator subwindow are some bells and whistles. The **DATA** one is a set of flippers that are inverted every time some data is sent to the emulator subwindow. The **ONLINE** and **LOCAL** buttons tell you if you have a connection opened. The **L1**, **L2**, **L3**, and **L4** buttons are settable by the host in the VT100 mode.

Special keys for the terminal emulator subwindow are:

- The **CNTL** key is **CONTROL(PROPS)**
- The **ESC** key is **COMPLETE(right arrow)**
- The **DEL** key is **DELETE**
- Cursor motion keys: Up, Down, Left, and Right are **HELP**, **DOIT(MARGINS)**, **NEXT**, and **UNDO**

If you are in the VT100 mode, there are several KeyPad and Programmable Functions Keys available to you. With the built in **Emulator.TIP** file, you have the following:

The VT100 KeyPad functions are invoked by:

- 0-9 are 0-9 with **COMMAND** held down
- Enter is **COMMAND-RETURN**
- (period) and , (comma) are . and , with **COMMAND** held down

The VT100 Programable Function Keys are invoked by:

PF1-PF4 are **MENU(CENTER)**, **SCROLLBAR(BOLD)**, **JFIRST(ITALICS)**, and **JSELECT(UNDERLINE)**

By changing the <>**TIP>Emulator.TIP** file and rebooting, you can assign these function to any key or key combination. See the *Mesa Programmer's Manual* for more on TIP tables.

32.2.4 Opening a connection

To open a connection to a CIU, open the options window by hitting **Options!**. If you bring up a menu over the option sheet, a menu called "RS232C Ports" appears. Selecting one of the RS232 ports causes the option sheet to change values. If you are talking to a CIU, fill in the **PhoneNumber:** field; if there is no dialer on the other end, keep the **PhoneNumber:** field empty. With the CIU, the communication parameters (such as, **CharLength=**, **StopBits: {}**, etc.) are ignored because the CIU uses the clearinghouse to get them. Hit **Connect!** on the option sheet to start a connection, after which the option sheet should disappear. If it does not disappear, you have hit the wrong **Connect!** button (on the NSTerminal window).

To open a connection to the local port of an ECS, fill in the **Host:**, **PhoneNumber:** (if there is a dialer connected to the local port), and all the communication parameters

(i.e., **CharLength=**, **StopBits: {}**, etc) and set the **LineNo:** field to 0 (zero). Hit **Connect!** and a connection will be opened.

To open a connection to the GAP services that Chat talks to (such as, remote system administration, the XDE remote executive, and the interactive terminal service) fill in the **Host:** and **LineNo=** fields. All other parameters are ignored. The line number for remote system administration is 32001, the XDE remote executive is 32002, and ITS is 32003.

32.2.5 NTerminal User.cm

NTerminal does extensive **User.cm** parsing. In addition to the standard entries, **User.cm** entries include.

```
[NTerminal]
Authenticate: <TRUE FALSE>
Host: <string using quote if name contains spaces. For example,
"Dialer:OSBU North:Xerox">
PhoneNumber: <string without punctuation. For example: (415)
    123-4567 becomes 4151234567>
CharLength: <5 6 7 8>
DataFile: <name of terminal file. Used only by wizards>
Duplexity: <full half>
Filter: <TRUE FALSE>
FlowControl: <none xOnXOff>
LineNo: <number, 0-65535, decimal format>
LineSpeed: <bps50 bps75 bps110 bps134p5 bps150 bps300 bps600 bps1200
    bps2400 bps3600 bps4800 bps7200 bps9600 bps19200 bps28800
    bps38400 bps48000 bps56000 bps57600>
Login: <TRUE FALSE>
Parity: <none odd even one zero>
Refresh: <always never half full>
StopBits: <1 2>
Terminal: <addrinfo adm3 adm3a cdc456 dm1520 gt100 h1000 h1420
    h1500 h1510 h1520 h2000 isc8001 soroc teletec trs80 vc303
    vt100 vt50 vt50h vt52 x820 xvt52>
XOn: <number, 0B - 177777BB, octal>
XOff: <number, 0B - 177777BB, octal>
```

32.2.6 User.cm example

Here is an example [NTerminal]User.cm section:

```
[NTerminal]
PhoneNumber: 2324343
Host: "1200Bps Venteller Port B1"
LineNo: 68
Terminal: vt100
Refresh: always
FlowControl: XOnXOff
```

XOn: 21B
XOff: 23B

32.3 Remote Executive

The Remote Executive is an executive service that permits users to connect to a remote machine and issue commands as if they were typing into an Executive. The Remote Executive supports an arbitrary number of connections from an arbitrary number of users. The Remote Executive is typically used to access integration machines, but it may also be run in the XDE to permit remote access to other workstations.

32.3.1 Files

Retrieve **RemoteExec.bcd** from the Release directory.

32.3.2 User interface

The Remote Executive is accessed from Chat on your local machine. For example, to connect to a machine named Yamamoto, running Remote Executive via Chat, you would type:

>Chat Yamamoto/e

Once connected, you are asked to log in to the Remote Executive for authorization purposes or to quit. You must log in with a legal user name and password. The list of authorized users is controlled by the AccessGroups entry in the **User.cm** for the Remote Executive. see the Remote Executive **User.cm** section in this chapter.

An authorization log in may not log you in to a machine. Since a machine can maintain one logged-in name at a time, you will be logged in to the machine only if there is no other user already logged in. If there is another user logged in, the system will print a message telling you the name of that user.

After connecting to the Remote Executive, only three commands are available: **LogIn.~**, **Quit.~**, and **ShowAccessList.~**(explained below). This initial **LogIn.~** command is different from the standard Executive **LogIn.~** command in that it will accept a fully qualified user name. After the initial log in, the **LogIn.~** command reverts to the standard Executive **LogIn.~** command. For example:

```
Login
Name: Yamamoto:OSBU North:Xerox
Password: *****
```

After the initial log in, more commands are made available (explained in the next section).

32.3.3 Commands

In addition to all the standard Executive commands (see the Executive chapter), the Remote Executive has the following additional commands:

BootFromFile. ~	allows you to boot a bootfile that is resident on the local file system. It takes one argument, the name of the bootfile.
BootVolume. ~	is the the Boot from menu of the Herald Window. It takes the name of a logical volume (plus optional switches) as its argument; if no argument if given, it acts like the boot button and boots the entire physical volume.
CallDebugger. ~	call the debugger (equivalent to pressing SHIFT-STOP).
ListRemoteHosts. ~	lists all currently connected users.
Online. ~	takes a physical volume or volumes as it arguments and brings the specified volume(s) on line.
Offline. ~	brings the specified physical volume(s) off line.
Quit. ~	causes the remote user to be disconnected.
RemoteExec. ~ [arg]	sets the Remote Executive on or off, based on the value of arg (which can have values "on" or "off"). If there is no argument, this command tells whether Remote Executive is on or off.
ShowAccessList. ~	shows the list of groups that can connect to your machine, as given in the User.cm file. (See the section below.)
Time. ~	gives the current time.
VolumeStatus. ~	provides information about the logical volumes of the machine. It lists the following data. type (normal, debugger, or debuggerDebugger), state of the volume (open or closed), and the number of disk pages occupied out of the total available. If no argument is given, it provides the information for each of the logical volumes on the disk. If given the name(s) of a specific logical volume as an argument, it provides the above information for only the specified volume(s) alone.

32.3.4 Remote Executive User.cm

The Remote Executive searches the [**System**] section of the **User.cm** file for the entry **AccessGroups**; this entry is a list of the names of individuals or groups permitted to use the machine through the Remote Executive. An entry looks like:

```
[System]
AccessGroups: "AnyGroup:OSBU North:Xerox" Smith Jones Johnson
```

If the domain and organization are left out, the defaults are used from the Profile Tool. If there are spaces in a name, the name must be quoted. If the entry "::*:::*" is used,

anyone may have access to the Remote Executive. To allow anyone to have access to your workstation, your User.cm entry would look like:

```
[System]
AccessGroups: *:*:*
```

Note: The access list is processed from left to right, so it is most efficient to put the most frequent users or user groups on the left side and those users who access the machine less often on the right side.

32.4 TTYTajo

An integration machine is a workstation configured with a very large disk. The design of the Dandelion makes it impossible to run both a very large disk and a large-format display at the same time. As a result, an integration machine is connected to a glass terminal rather than to a large-format display.

You cannot run the standard XDE boot files on an integration machine, since they depend upon the large-format display. TTYTajo is a boot file that runs on a machine (typically an integration machine) and provides the basic facilities of the development environment. It supports only a TTY-style interaction with the user, either through a simple terminal or through the Remote Executive.

32.4.1 Files and installation

Retrieve **TTYTajoTriDlion.boot** from the Release directory if your machine has a Trident disk, otherwise retrieve **TTYTajoDLion.boot** if your machine has a Shugart or Quantum disk.

A sample **User.cm** file is on the Release directory. Retrieve TTYTajoUser.cm and rename it to **User.cm**.

The recommended boot switches (which you can set via Othello) for TTYTajo are: {}]

32.4.2 User interface

You can communicate with TTYTajo either by typing into the simple keyboard attached to the integration machine or by using the Remote Executive (see the Remote Executive section). Characters typed into the keyboard are sent to the local Executive. The Executive, the Remote Executive, and FTP are built into TTYTajo.

The Remote Executive recognizes the following character codes (defined in the interface **Ascii.mesa**) as special editing characters: **Ascii.BS**, **Ascii.ControlC**, **Ascii.ControlW**, **Ascii.ControlX**, **Ascii.DEL**, **Ascii.ESC**, and **Ascii.Tab**. The Remote Executive's interpretation of these characters is described in the Executive chapter. You should consult this manual for your simple terminal to see how to generate these characters from that terminal. The abort function, provided by the **STOP** key for a local executive, is provided by the **Break** key on most simple terminals.

32.4.3 Commands

In addition to the standard Executive commands (see the Executive chapter) and Remote Executive commands, TTYTajo has the following command:

FTP.~

is built into TTYTajo. This command allows you to transfer files between the workstation and remote file servers. The documentation for this command can be found in this manual.

32.4.4 User.cm

A sample User.cm is given below. (TTYTajoUser.cm from the Release directory).

```
[User.cm]

[System]
AccessGroups: *;*;*
Debug: No
Domain: OSBU North
InitialCommand: MfileServer;
Organization: Xerox
User: TTYTajo

[Executive]
CodeLinks: FALSE
Priority: 1
UseBackground: TRUE

[HardCopy]
Columns: 2
Interpress: Nevermore
Orientation: Landscape
PreferredFormat: Interpress

[MfileServer]
Running: TRUE
StoreAllowed: TRUE
OverWriteAllowed: TRUE
DeleteAllowed: TRUE
```

32.4.5 Program interface

The following interfaces are exported by TTYTajo. Programs that use only these interfaces can run in the TTYTajo environment

Common software interfaces:

```
Format
Real
RealFns
```

String
Time
TTY

Tajo interfaces:

AddressTranslation
Atom
BlockSource
BodyDefs
BTree
CmFile
Date
DiskSource
Event
EventTypes
Exec
Expand
FileTransfer
HeraldWindow
MFile
MFileProperty
MLoader
MSegment
MStream
MVolume
PieceSource
Profile
ScratchSource
StringLookUp
StringSource
TajoMisc
TextSource
Token
Version

Pilot interfaces: all.



V

TCP/IP Related Tools and Applications



ARPA Getting Started

33.1 Installing the ARPA network protocols in XDE

To install the ARPA protocols in XDE, follow these steps:

1. Retrieve the file **HOSTS.TXT** so you can address machines with logical names rather than internet addresses, and to provide start up information.

This file has a list of all registered machines on your network. (If your machine is not registered in the table, you must register it with your network's copy of the **HOSTS.TXT** file).

Edit **HOSTS.TXT** as follows:

- a. Find the line of text: **MY-HOST: 0.0.0.0**
- b. Replace the address 0.0.0.0 with the address of your machine which is found by looking at your entry in the table. If this is not done, or the file is not found on your machine, the MP panel will read 982.
- c. Find the line of text: **MY-GATEWAY: 0.0.0.0**
- d. Replace 0.0.0.0 with the host number of your local gateway if it has one.

The addresses must be in the same address class as other entries for your network in the table. The network number is set by either looking in the **MY-HOST** field and extracting the network number, or by querying the local gateway specified in the field **MY-GATEWAY** for the network number. If either of these methods fails, then let the network number default to network number zero.

If your machine address is on a subnet, you must use the field **SUBNET-MASK** in the **HOSTS.TXT** file to find the subnet masking bits. Or you can use the ArpaRouteTool. See the documentation on the ArpaRouteTool for details.

2. Retrieve the rest of the ARPA software. The software you need is:

ArpaComm.bcd: implements the Arpa transport protocols (IP, UDP, and TCP) and address resolution protocols. The interfaces exported by this module are TcpStream,

ArpaPort, ArpaAddressTranslation, ArpaAddressCache, ArpaHostTable and ArpaConstants.

ArpaTelnetConfig.bcd: provides the Telnet application level protocol which the File Transfer Protocol (FTP) depends upon. There are currently no public exported interfaces in this module.

ArpaFiling.bcd: provides the ARPA FTP and Trivial File Transfer Protocol (TFTP) file transfer protocols. The interfaces exported by this module are TFTPL, ArpaFTP, ArpaFTPServer and ArpaFilingCommon.

ArpaMailing.bcd: provides the ARPA Simple Mail Transfer Protocol (SMTP). The interfaces exported by this module are ArpaSMTP, ArpaSMTPServer and ArpaMailParse.

(If you are running in the XDE environment, the file, **ArpaConfig.bcd**, provides the same functions as the above four files. By using ArpaConfig.bcd all the protocols are loaded in the right order and the Arpa protocols won't be started unless there is a **HOSTS.TXT** file present on the current directory.)

ArpaFileServer.bcd: implements a window or command line interface for the ARPA file server. See the *XDE User Guide* for details.

ArpaFileTool.bcd: implements a window interface much like the XDE FileTool, which allows you to retrieve, store, list, and so on, on an ARPA file server. See the *XDE User Guide* for details.

ArpaChat.bcd: implements a TTY window interface to remote hosts' telnet processes. See the *XDE User Guide* for details.

ArpaRemoteExec.bcd: provides an executive to remote users who connect to the local workstation using the Telnet protocol. See the *XDE User Guide* for details.

ArpaMailTool.bcd: implements a window interface much like the NS based XDE MailTool. It allows you to send and receive mail. See the *XDE User Guide* for details.

ArpaCacheAddress.bcd: provides a Mesa executive interface to the **HOSTS.TXT** information. See the *XDE User Guide* for details.

To load the above modules in the proper running order, use this command line:

```
Run.^ ArpaComm ArpaTelnetConfig ArpaFiling ArpaMailing  
ArpaFileServer ArpaFileTool ArpaChat ArpaRemoteExec ArpaMailTool  
ArpaCacheAddress;
```

OR

```
Run.^ ArpaConfig ArpaFileServer ArpaFileTool ArpaChat  
ArpaRemoteExec ArpaMailTool ArpaCacheAddress;
```

ArpaCacheAddress

The ArpaCacheAddress provides a user interface to the address cache and host table parsing mechanism for the Arpanet addressing scheme.

34.1 Files

Retrieve **ArpaComm.bcd** and **ArpaCacheAddress.bcd** from the Release directory.

34.2 User Interface

ArpaCacheAddress registers the command "**ArpaCacheAddress.~**" with the XDE executive. The following command arguments are understood:

- Flush** removes any current information out of the address cache.
- List** enumerates all entries in the address cache with the entry name followed by the address corresponding to that name.
- Load/file** takes the file specified in the **file** field and parses and loads all relevant entries from this file into the cache.
- AddEntry name/address**
takes the **name**, **address** pair and adds them to the address cache.



ArpaChat

ArpaChat provides simple TTY-emulation in the development environment. It runs on a standard Tajo or CoPilot bootfile and is based upon the Telnet protocol of the TCP/IP family of protocols.

35.1 Files

Retrieve **ArpaChat.bcd** from the Release directory.

35.2 User Interface

ArpaChat registers the command "**ArpaChat.~**" with the executive. The simplest form of the command is:

>ArpaChat.~

This command either activates an inactive ArpaChat if there is one, or it creates a new one. The full form of the ArpaChat command is:

>ArpaChat.~ [host]

host tries to open a connection to that host (see the **Connect!** command below).

After you type this command to the Executive, an ArpaChat tool window appears. ArpaChat's tool-style interface has a message subwindow, a form subwindow, and a TTY subwindow.

35.2.1 Message subwindow

The message subwindow is used for one-line messages about the current state of the Telnet connection.

35.2.2 Form subwindow

The form subwindow contains several commands:

- Connect!** using the current selection as a host name or address, **Connect!** opens a connection to that host. After a connection is established, a message is posted in the message subwindow so you can start typing. A faster way is to type a host name in the file subwindow followed by pressing **DOIT**. It takes the last word typed as the host name and invokes the **Connect!** command. The **Connect!** command behaves slightly differently depending on the values of some of the fields described below.
- Disconnect!** if there is a connection open, **Disconnect!** deletes the connection for the network stream, collects and throws away the tool's various processes for managing the data stream, and returns the tool to a quiescent state.
- BreakKey!** simulates a terminal's break character.
- Another!** starts up another ArpaChat window, using the same options as the current ArpaChat window.
- Destroy!** destroys the ArpaChat window. No confirmation is required.
- Options!** creates an ArpaChat options window. The options are:
- Apply!** sets your chosen options and destroys the options window.
 - Abort!** cancels any changed options and destroys the options window.
- LogIn** If this Boolean is **TRUE** and both **Profile.User** and **Profile.Password** are non-null, ArpaChat logs you in on the remote host using these values. If the Boolean is **FALSE**, ArpaChat won't log you in. The default value is **TRUE**. (You can set this value in the [ArpaChat] section of your **User.cm** file. Or, if the LogIn Boolean in the Options window is selected, you are logged in automatically.)
- Interrupt!** sends the Telnet Interrupt character to the connected host.
- Abort!** sends the Telnet Abort character to the connected host.
- AreYouThere!** sends the Telnet AreYouThere character to the connected host. A response is returned by the connected host.
- EraseChar!** sends the Telnet erase character character to the connected host.
- EraseLine!** sends the Telnet erase line character to the connected host.
- GoAhead!** sends the Telnet go ahead character to the connected host.
- Echo** when this boolean is set to **TRUE**, the connected host echos characters to the user rather than the local terminal emulator.

LocalEchoOn when set to **TRUE**, the local keystrokes are echoed to the user. If set to **FALSE**, the user's keystrokes are not echoed. The default is **TRUE**. When remote echoing is enable, this boolean is not available.

PortType gives the type of port that ArpaChat connects to. The defualt is Telnet and the other options are FTP, SMTP and other. If the other option is selected, the port connected to is taken from the **Port** field of the tool.

Port gives the decimal value of the port to connect to. The default is decimal 23, the Telnet port.

35.2.3 TTY subwindow

ArpaChat also has a TTY subwindow in which the dialogue with the remote system takes place. When a connection is established, characters sent from one machine to another appear in the TTY subwindow.

An alternate way to connect to a host (rather than using the **Connect!** command) is to type the host name into this subwindow, and hit the **DOIT** key (the one labeled **MARGINS** on the 8010 keyboard).

35.2.4 Special keys

ArpaChat makes use of the following special keys:

COMPLETE: sends an Ascii ESC.

DELETE: sends an Ascii DEL.

BS: sends an Ascii BS (CONTROL-H).

BW: sends an Ascii ETB (CONTROL-W).

ABORT: does a Telnet abort on the current connection.

35.2.5 ArpaChat User.cm entries

ArpaChat reads the following **User.cm** options:

[ArpaChat]
Login: TRUE|FALSE

ArpaRemoteExec

The Remote Executive is an executive service that permits users to connect to a remote machine and issue commands as if they were typing into an Executive. The Remote Executive supports an arbitrary number of connections from an arbitrary number of users. The Remote Executive is typically used to access integration machines, but it may also be run in XDE to permit remote access to other workstations.

36.1 Files

Retrieve **ArpaRemoteExec.bcd** from the Release directory.

36.2 User Interface

The Remote Executive is accessed from ArpaChat on your local machine. For example, to connect to a machine named Inferno, running Remote Executive through ArpaChat, you would type:

>**ArpaChat Inferno**

Once connected, you are asked to log in to the Remote Executive for authorization purposes or to quit. You must log in with a legal user name and password. The list of authorized users is controlled by the AccessGroups entry in the **User.cm** for the Remote Executive; see the Remote Executive **User.cm** section in this chapter.

An authorization log-in may not log you in to a machine. Since a machine can maintain one logged-in name at a time, you will be logged in to the machine only if there is no other user already logged in. If there is another user logged in, the system prints a message telling you the name of that user.

After connecting to the Remote Executive, only three commands are available: **LogIn.~**, **Quit.~**, and **ShowAccessList.~** (explained below). This initial **LogIn.~** command is different from the standard Executive **LogIn.~** command in that it accepts a fully qualified user name. After the initial log in, the **LogIn.~** command reverts to the standard Executive **LogIn.~** command. For example:

Login**Name:** Joe:Accouting:UCB**Password:** *****

After the initial log in, more commands are made available (explained in the next section).

36.2.1 Commands

In addition to all the standard Executive commands (see the Executive chapter), the Remote Executive has the following additional commands:

- BootFromFile.**~ allows you to boot a bootfile that is resident on the local file system. It takes one argument, the name of the bootfile.
- BootVolume.**~ is the Boot from menu of the Herald Window. It takes the name of a logical volume (plus optional switches) as its argument; if no argument is given, it acts like the boot button and boots the entire physical volume.
- CallDebugger.**~ calls the debugger (equivalent to pressing SHIFT-STOP).
- ListRemoteHosts.**~ lists all currently connected users.
- Online.**~ takes a physical volume or volumes as arguments and brings the specified volume(s) on line.
- Offline.**~ brings the specified physical volume(s) off line.
- Quit.**~ disconnects the remote user.
- RemoteExec.**~ [arg] sets the Remote Executive on or off, based on the value of arg (which can have values "on" or "off"). If there is no argument, this command tells whether Remote Executive is on or off.
- ShowAccessList.**~ shows the list of groups that can connect to your machine, as given in the User.cm file. (See the section below.)
- Time.**~ gives the current time.
- VolumeStatus.**~ provides information about the logical volumes of the machine. It lists the following data: type (normal, debugger, or debuggerDebugger), state of the volume (open or closed), and the number of disk pages occupied out of the total available. If no argument is given, it provides the information for each of the logical volumes on the disk. If given the name(s) of a specific logical volume as an argument, it provides the above information for only the specified volume(s) alone.

36.2.2 Remote Executive User.cm

The Remote Executive searches the [System] section of the **User.cm** file for the entry AccessGroups; this entry is a list of the names of individuals or groups permitted to use the machine through the Remote Executive. An entry looks like:

```
[System]
AccessGroups: "AnyGroup:Accounting:UCB" Smith Jones Johnson
```

If the domain and organization are left out, the defaults are used from the Profile Tool. If there are spaces in a name, the name must be quoted. If the entry "***:*:***" is used, anyone may have access to the Remote Executive. To allow anyone to have access to your workstation, your User.cm entry would look like:

```
[System]
AccessGroups: *:*:*
```

Note: The access list is processed from left to right, so it is most efficient to put the most frequent users or user groups on the left side and those users who access the machine less often on the right side.

ArpaFileTool

The ArpaFileTool provides a user interface to the Arpanet based file transfer mechanisms commonly called FTP (File Transfer Protocol) and TFTP (Trivial File Transfer Protocol).

37.1 Files

Retrieve **ArpaComm.bcd**, **ArpaTelnetConfig.bcd**, **ArpaFiling.bcd** and **ArpaFileTool.bcd** from the Release directory.

37.2 User Interface

The ArpaFileTool communicates through a form subwindow, a command subwindow, a log subwindow and an options window.

37.2.1 Form subwindow

The fields used as arguments to a command are listed in the form subwindow:

Host: is the name of the host to be used for remote files and operations.

Directory: is the remote directory relative to the default directory.

Source: is a list of files for the next command to act upon. File names may include wildcard/expansion characters. Any files appearing in this field should conform to the syntax of file names for the file system that is the source of the transfer

Dest 'n: is the file name for the destination of a transfer. It should conform to the syntax of the file system that is the destination of the transfer.

LocalDir: means that all references to the local disk will only occur within this directory. If the directory is not a complete path name (if it does not begin with <), it is assumed to have a <> prefixed.

Name: is the name of the user on the remote machine.

Password: is the password of the user on the remote machine. This is echoed with asterisks.

Account: is the account number of the user on the remote machine.

Protocol: is an enumerated type giving the filing protocol to be used with the remote machine. The protocols supported are:

TFTP uses the Trivial File Transfer Protocol.

FTP uses the File Transfer Protocol.

37.2.2 Command subwindow

The following commands are available for either the FTP or TFTP protocols:

Retrieve! transfers the file name or names specified in **Source** from the remote file system to the local disk. If **Dest'n** is blank, the file name of the copy made on the local disk is the source file name stripped of all host and directory qualifiers.

Store! transfers the file name specified in **Source** from the local disk to the remote host. Development environment file name conventions apply to the local file.

Options! creates an Options window if one does not already exist.

The following commands are only available to the FTP protocol:

Remote-List! lists all files on the remote file system corresponding to the name or names in **Source**. These names must conform to the file-naming conventions on the remote host. You may designate multiple files by the use of '*' only to the extent that the remote server supports it.

Remote-Delete! deletes the file name or names specified in **Source** from the remote file system. You may designate multiple files by the use of '*' only to the extent that the remote server supports it.

Close! closes any currently open connection, freeing any resources needed to maintain it.

Rename! renames the file name specified in **Source** to the file name specified in **Dest'n** on the remote file system.

ReInitialize! allows the user to start the current session over again without breaking the connection to the remote host.

Abort! stops the current filing transaction. This can also be done by using the **STOP** key over the ArpaFileTool window.

37.2.3 Options window

The Options window is created by the **Options!** command. The options window uses two subwindows, a command subwindow and a form subwindow.

37.2.4 Options command subwindow

The following are the commands of the options command subwindow:

- | | |
|---------------|---|
| Apply! | applies the currently set options and closes the window. |
| Abort! | closes the options window without applying the set options, maintaining the initial options settings. |
| Reset! | resets the options to their initial values. |

37.2.5 Options form subwindow

The options form subwindow sets options that effect the command in the ArpaFileTool command subwindow. These options are dependent upon the protocol selected in the ArpaFileTool form subwindow. The following are the options for the FTP protocol :

- | | |
|-----------------------|--|
| FileDelimiter: | is the character prefixed to the directory file name boundary if it does not currently have this separator. |
| FileType | defines the way a file is stored. The following are the acceptable types:

Ascii for plain text.

Image for binary files.

Other for files that have some byte representation other than what can be accommodated by Image . |

When the **Ascii** type is selected, an additional field appears:

- | | |
|-------------------|---|
| FileFormat | is an enumerated type formating schema that is used in the stored or retrieved file. The following are the format types that are acceptable:

Non Print a file which contains no specific formating information.

Telnet a file which contains vertical format controls (such as <CR>, <LF>, <NL>, <VT>, <FF>).

ASA a file which has ASA (FORTRAN) formating. (See RFC740 or <i>Communications of the ACM Vol. 7, No. 10, p. 606, October 1964</i>). |
|-------------------|---|

When the file type **Other** is chosen, the following field appears:

- ByteSize** is the logical byte size of the file being transferred.
- FileStructure** gives the structure of the file to be transferred. The following is the list of values:
- File** is the XDE file structure.
- TransmissionMode** is an enumerated type giving the method of transfer. The following is a list of possible transfer options:
- Stream** is a stream of data bytes.
 - ListOutput** gives the amount of information desired as a result of the **Remote-List** command. The following options are available:
 - verbose** gives as much information about the file as possible.
 - terse** gives just the file name.
 - Debug** allows debugging information to be printed during filing operations when set to TRUE.
 - CheckDate** checks for creation time information. The file server must support the following format to the response of the remote list command in verbose mode:


```
<file information> created: <date in RFC822 format> <file information> <CRLF>.
```

When the protocol is set to TFTP, in addition to the fields **FileDelimiter** and **Debug** described above, the following fields are also provided:

- FileType** is the type of file to be transferred. The following is a list of the available file types:
- NetAscii** eight bit Ascii code.
 - Octet** eight bit binary data.
 - Mail** netascii characters to be sent to a user rather than a file.
- Retransmission Timeout (per-packet, in seconds)** gives the time interval between TFTP data packets.
- Total retransmission interval (in-seconds)** gives the total timeout interval for a TFTP connection attempt.

Both the above values should be experimented with when the defaults do not work.

37.3 User.cm entries

The standard **User.cm** entries **InitialState**, **TinyPlace** and **WindowBox** are supported.

37.4 References

RFC740 NETRJS Protocol - Appendix C, Braden, November, 1977.

RFC822 Standard for the Format of ARPA - Internet Text Messages, Crocker, August, 1982.

An RFC can be copied from the <RFC> directory at SRI's machine:

SRI - NIC.ARPA

using FTP with username, ANONYMOUS, and password, GUEST.



ArpaFileServer

The ArpaFileServer provides a means of turning a workstation or integration machine into a FTP (File Transfer Protocol) and TFTP (Trivial File Transfer Protocol) file server.

38.1 Files

Retrieve **ArpaComm.bcd**, **ArpaTelnetConfig.bcd**, **ArpaFiling.bcd** and **ArpaFileServer.bcd** from the Release directory.

38.2 User Interface

The ArpaFileServer can be run as an Executive based tool or as a window based tool according to your **User.cm** entries. Running the ArpaFileServer on a machine that does not support a large format display will cause it to register commands with the executive.

38.2.1 Tool window interface

If the window mode is used, the ArpaFileTool communicates through a file subwindow and a command subwindow

The fields in the command subwindow are:

Start	Enables FTP and TFTP listeners and enables current settings. The tool starts with the RetrieveAllowed option set when it is loaded unless otherwise set in the User.cm entry.
LogActivity	Enables the activity and debug logs for TFTP and FTP.
StoreAllowed	Enables file storing.
RetrieveAllowed	Enables file retrieval.
DeleteAllowed	Allows file deletion.
OverWriteAllowed	Allows the overwriting of a file.

38.2.2 Executive interface

When run as an Executive tool, the following command is registered with the Executive:

ArpaFileServer.~ command command ...

The following is a list of commands:

- l** used to log server activity
- s** allows you to store files
- d** allows you to delete files
- r** allows you to retrieve files
- o** allows you to overwrite files on storing
- state** displays the current setting of all the above.

A '-' in front of any of a command disables it.

For example, **ArpaFileServer.~ -l s d r**, stops logging but allows storing, retrieving and deleting of files.

38.2.3 Server activity log

Three kinds of messages are printed in the ArpaFileserver activity log. Messages of TFTP and FTP connections are printed with the originating host ID and for TFTP the file name of the transaction. Messages sent by the FTP session are displayed with the symbols ">>>" appended to them and messages received by the FTP server are displayed as they are received. Messages are printed only if the log server activity boolean is set to true. If the window version of the tool is used, then messages are printed in the tool window, otherwise they are printed in the default log window, either the herald or the Executive window.

38.3 User.cm Entries

The **User.cm**, in addition to the standard **InitialState**, **TinyPlace** and **WindowBox** entries, includes:

[ArpaFileServer]

Window: **TRUE|FALSE** This boolean determines if the tool is initialized as a window interface or an Executive command line interface. The default is **TRUE** in an XDE window environment and **FALSE** in an XDE nonwindow environment, such as an integration machine.

LogActivity: **TRUE|FALSE** This boolean determines if the tool displays a log of connection activity. If the tool is in a window mode, the log is

displayed in the FileSW, otherwise log messages are displayed in the default subwindow, either the herald or the screen.

DeleteAllowed: **TRUE|FALSE** This boolean enables deletion of files by any connected user if set to TRUE.

StoreAllowed: **TRUE|FALSE** This boolean enables storing of files by any connected user if set to TRUE.

RetrieveAllowed: **TRUE|FALSE** This boolean enables retrieving of files by any connected user if set to TRUE.



ArpaMailTool

The ArpaMailTool is a SMTP (Simple Mail Transport Protocol)-based mail reading and sending tool. The ArpaMailTool allows you to retrieve, read, send, forward, save, move, delete, and answer mail. In order to receive mail at your local host, you must include your name in the valid recipient list (See "WillAcceptMailFor" entry under **User.cm** in 1.1.4.6) of the **User.cm**.

If your mail file becomes damaged, you may be able to save it by running **MailFileScavenger**. **MailFileScavenger** restores the internal structure of your mail file to a consistent state. It copies the damaged mail file into a scratch file as it operates, so you must have enough free disk pages available for this scratch file in addition to the number of disk pages that your damaged mail file already occupies. **MailFileScavenger** warns you if there is not enough room.

39.1 Files

Retrieve **ArpaMailTool.bcd** from the Arpa Release directory.

39.2 User Interface

The ArpaMailTool has its own window consisting of a message subwindow, two text subwindows and a form subwindow, as shown in Figure 1. Information and error messages are posted in the message subwindow. The table of contents for the currently active mail file is displayed in the text subwindow directly below the message subwindow. The form subwindow lists commands for manipulating your mail. The lower text subwindow displays individual mail messages. The name stripe of this window indicates when the last mail was received for this host.

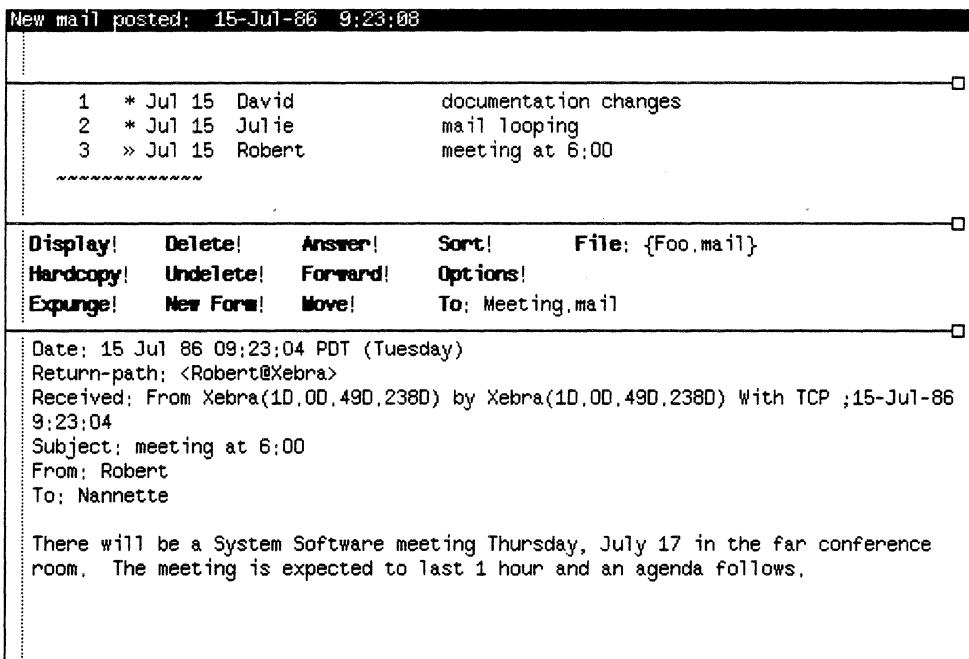


Figure 39.1: ArpaMailTool

39.2.1 Text subwindow – Table of contents

An index of all messages in this mail file, called the Table of contents (or TOC), appears in the upper text subwindow of the ArpaMailTool window. Each entry contains header information, which includes the message number, the date it was sent, the name of the sender, and the subject of the message.

You can have more than one mail file to store and organize your messages in. The *current* mail file is the one whose TOC is displayed and the one to which new messages are retrieved. Its name is displayed in the **File:** field described below. When the ArpaMailTool starts up, it reads the mail file specified by the **User.cm** or **Active.mail** if none is specified. You can change the current mail file by chording and selecting from the **File:** field.

The currently displayed message is marked by a ***** character after the date column. Deleted messages have a line through their entries in the TOC. Unexamined messages are marked with an asterisk (*).

If a one character selection is made for the first character in a TOC line, then the next character typed becomes the "flag" character for that entry. This flag has no semantic meaning to the ArpaMailTool, but may be used for whatever purpose you want. For instance, you might mark all those messages you need to answer with the character "A," or you might mark those that are urgent with the character "U."

39.2.2 Form subwindow

By making a text selection that spans a number of lines in the Table of Contents, it is possible to select a range of messages. Those messages are said to be the *current* messages. The ArpaMailTool uses the current messages as an argument for most commands. If there is no selection in the TOC, the current message is the displayed message.

- Display!** displays the first of the current messages if there is a selection in the TOC; otherwise, it displays the next message. The next message is the first undeleted message following the displayed message.
- Hardcopy!** formats the current messages for printing and either spools them to a printer or writes them into a local file. **Print** is loaded as needed. **Note:** An NS-based printer is required to hardcopy mail messages.
- Delete!** marks the current messages for deletion by drawing a line through their entries in the TOC. Messages are not removed from the message file immediately, but only when expunged (see **Expunge!** below), after which there is no way to restore them. Before deleted messages are expunged, they may be restored by the **Undelete!** command.
- Undelete!** restores the current messages marked for deletion.
- Expunge!** permanently removes messages marked for deletion from the mail file.
- Forward!** produces a form containing a message body that is a copy of the current message and header fields that can be filled in by hitting the **NEXT** key.
- New Form!** produces a blank form with header fields that can be filled in by hitting the **NEXT** key.
- File:** {**Active.mail**, ...} is an enumerated item which indicates the current mail file (the file where new messages are stored and whose TOC is displayed). You may choose a different message file as the current file by selecting from the menu under this item. Only .mail files are shown, and if there are duplicates in the search path, only the first is found. The default mail file can be set from the **User.cm** or from the Options window.
- Options!** activates the Options window .
- Sort!** sorts the messages in a mail file by the date and time each was sent.
- Move!** moves the current messages to the mail file named in the **To:** item. This feature allows you to organize your messages for easy reference. The extension .mail is assumed if there is no period in the name.

Note: Any selection in the TOC is cleared if you edit the **To:** field. You must fill in that field before selecting the messages to be moved. If you are merely moving a displayed message, this problem does not occur.

To: contains the name of the mail file that is the destination for **Move!** The extension is defaulted to **.mail**. You can also fill in this field by pressing **MENU** and selecting a name from the currently existing **.mail**.

39.2.3 Options window

The Options window contains the following items. For most options default initial values can be specified in the ArpaMailTool section in **User.cm**.

Apply! causes the fields in the Options window to take effect and closes the Options window.

Abort! closes the Options window without making any changes.

Debug! activates a window used primarily for debugging the SMTP protocol. The protocol exchange is visible through this window. (See Figure 2.)

ArpaName: specifies the user name used by the ArpaSendTool in determining return fields.

AutoDisplay is a Boolean that, if **TRUE**, causes the next message to be displayed when the current message is deleted or moved. The default is **FALSE**.

Mail File: names the mail file you wish to work with. This file becomes the current mail file when you invoke **Apply!** The extension defaults to **.mail**. You can also fill in this field by pressing **MENU** and choosing the name from the currently existing mail files. If you invoke **Apply!** when the **Mail File** field is blank, the value defaults to **Active.mail**.

-- Hardcopy Options --

OnePerPage is a Boolean that, if **TRUE**, causes each message to start on a separate page. The default is **TRUE**.

OutputToFile is a Boolean that, if **TRUE**, causes the output from **Hardcopy!** to be written to a file instead of being spooled to a printer. The default is **FALSE**.

Sides: {PrinterDefault, SingleSided, DoubleSided} tells the printer whether to do two-sided printing or not. If the printer does not support two-sided printing, this option is ignored. The default is **SingleSided**.

Orientation: {Portrait, Landscape} specifies the orientation of the output. **Landscape** output is two columns per page. **Portrait** is one column per page. The default is **Portrait**.

Landscape Font: Portrait Font: are two fields to indicate which fonts to use when messages are printed. The default font when printing in **Portrait** orientation is Gacha6; for **Portrait**, Gacha8.

Printer: specifies the name of the interpress printer where the hardcopy is sent.

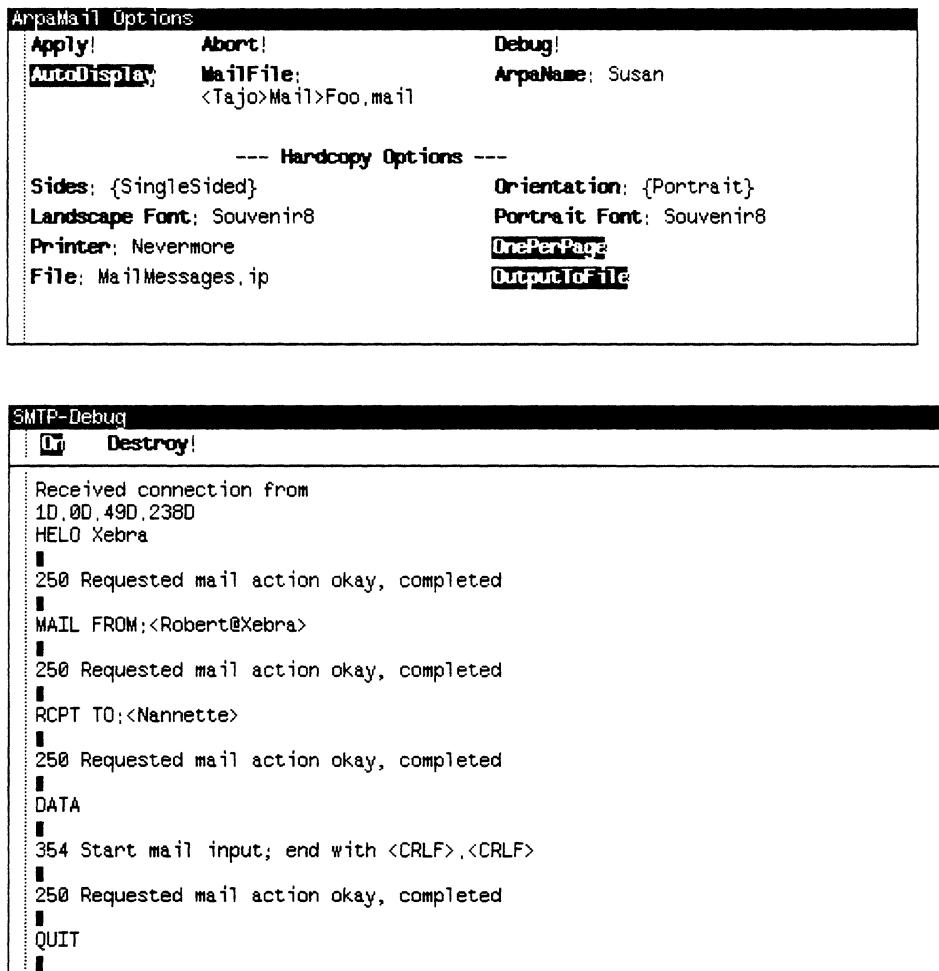


Figure 39.2: ArpaMailTool Options Window and SMTP Debugger

39.3 ArpaSendTool

The ArpaSendTool is used to send messages. A blank mail form is created by either invoking **New Form!**, **Answer!**, or **Forward!** in the ArpaMailTool window or invoking **Another!** in an open ArpaSendTool window. The ArpaSendTool has a message subwindow, a form subwindow, and a text subwindow.

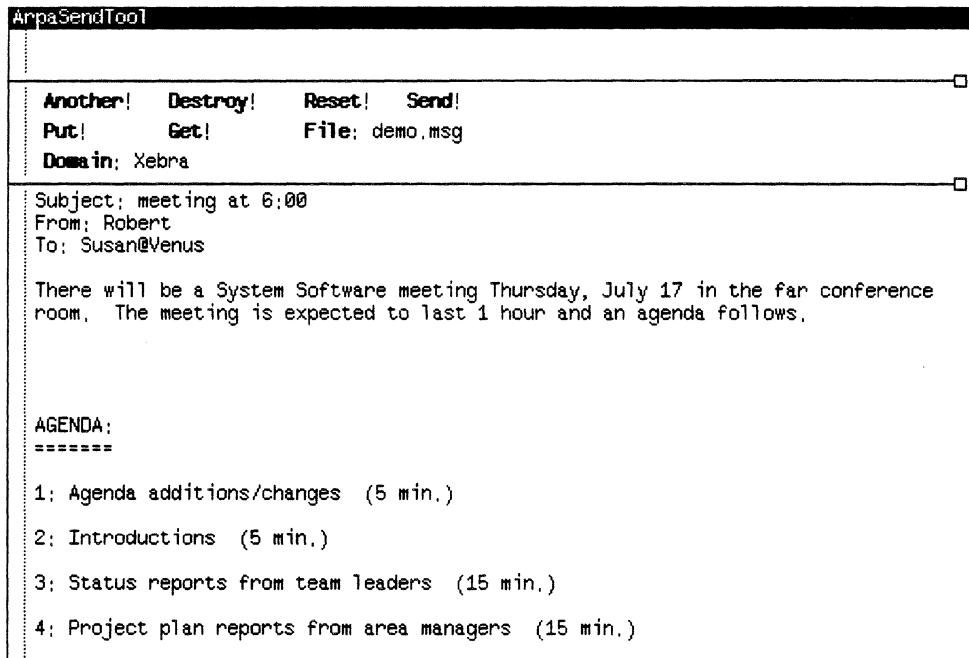


Figure 39.3: ArpaSendTool

39.3.1 Form subwindow

These items are always available in the form subwindow:

- Another!** creates another instance of the tool.
- Destroy!** destroys this instance of the tool.
- Reset!** leaves the tool window open but clears it of text.
- Put!** writes the contents of the tool window to the file named in the **File:** field.
- Get!** replaces the contents of the tool window with the contents of the file named in the **File:** field. If the form has been edited but not sent, this command requires confirmation.
- File:** is used to hold the name of the file used in the **Put!** and **Get!** commands.
- Domain:** contains the name of the local host. If any recipient names in the **To:**, **cc:**, or **bcc:** fields lack host fields, this value is automatically appended before delivery. (See below.)
- Send!** sends the mail to the recipients indicated in the **To:**, **cc:**, and **bcc:** lines of the message. A list of invalid recipients is posted to the message subwindow.

39.3.2 Text subwindow

The text subwindow contains the text of the message, including a header part and a message body part. The header part includes **Subject:**, **To:**, **bcc:**, **Reply-To:**, and **cc:** fields that are used to send the message.

39.3.2.1 Subject: field

The topic of your message goes in the **Subject:** field. The topic should express the content of your message so that interested people take the time to read the message, but uninterested people can delete it without reading it. For example, if your message contains ideas for improving the ArpaMailTool, the topic might be "Suggestions: improving ArpaMailTool," *not* "Suggestions."

39.3.2.2 To: field

The **To:** specifies who is to receive your message. A recipient is specified by a name@host entry. The ArpaMailTool allows you to omit the host name for recipients who are at your same host. For example, Somebody@LocalHost, can send the following

```
Subject: Meeting at 2:00 Wednesday
To: Person1, Person2
cc: Person3, Person4@AnotherHost
```

The ArpaMailTool assumes that names lacking host fields are at the sender's host, which in this case is **LocalHost**. Since **Person4@AnotherHost** includes the host, **AnotherHost** is used by the ArpaMailTool. In this case, the message goes to **Person1@LocalHost**, **Person2@LocalHost**, **Person3@LocalHost** and **Person4@AnotherHost**.

Distribution Lists:

Distribution lists are currently not implemented.

39.3.2.3 Reply-To: field

The **Reply-To:** field works with the **Answer!** command. **Answer!** initializes a message form to reply to the message selected in the Table of Contents. If the message being answered contains a **Reply-To:** field in its header, then only those recipients in the **Reply-To:** field are included in the **To:** field constructed by **Answer!**. The **Reply-To:** field limits those who automatically receive answers to messages. A recipient of such a message can change the recipient fields constructed by **Answer!**.

39.3.2.4 cc: field

The **cc:** (carbon copy) field identifies others who are to receive your message. Names should be separated by commas. When you send your message, these people automatically receive it along with the person or persons specified in the **To:** field.

39.3.2.5 bcc: field

The **bcc:** (blind carbon copy) field identifies others who are to receive your message, but whose names do not appear in the recipient list of the header.

39.3.2.6 Message body

The message body (the actual content of the message) follows the header. There must be an empty line between the last field in the header and the message body.

39.3.2.7 User.cm entries

Some ArpaMailTool parameters can be set from the **User.cm**. These are listed below with sample values.

```
[ArpaMailTool ]  
  
ArpaName: Bill                                -- name of the user  
  
TOCLines: 6                                    -- number of initial lines  
                                                displayed in the table  
                                                of contents (TOC)  
  
MailFile: Active.mail                          -- name of initial mail  
                                                file  
  
MessageFont: Snail6.strike                    -- if omitted, the built-in  
                                                Tajo font is used  
  
TOCFont: Gacha8.strike                      -- if omitted, the built-in  
                                                Tajo font is used  
  
AutoDisplay: FALSE                            -- if TRUE, next message is  
                                                displayed when current  
                                                message deleted  
  
WillAcceptMailFor:["John" "Mary" "Bill"] -- list of valid recipients  
                                                for mail delivered to  
                                                this local host
```

```
NewForm:  
  "Subject: <>  
  From: Bill  
  To: <>  
  Reply-To: Bill@Clover
```

```
«Message»
```

```
-- Bill"
```

*-- the quoted text is used
by NewForm! to customize
the send window*

You can also specify the printing characteristics used by the **Hardcopy!** command. If no printing entries are made in your ArpaMailTool **User.cm** section, the values from the **[Hardcopy]** section are used. Refer to the Print chapter for further information about the different entries.

OutputToFile: FALSE	<i>-- if TRUE, output is written to a file instead of a printer</i>
OutputFile: MailMessages.interpress	<i>-- name of output file used when OutputToFile is TRUE</i>
OnePerPage: FALSE	<i>-- if TRUE, each message starts at the top of a new page</i>
Sides: SingleSided	<i>-- controls whether the printer does two-sided printing or not</i>
InterPress: Nevermore	<i>-- name of the default InterPress printer to use</i>
LandscapeFont: Gacha6	<i>-- name of the default font to use when in landscape mode</i>
PortraitFont: Gacha8	<i>-- name of the default font to use when in portrait mode</i>
Orientation: Landscape	<i>-- default output orientation</i>
PrintedBy: \$	<i>-- name to place on the banner sheet when output is printed. A dollar sign (\$) means the current login name is used</i>

39.4 MailFileScavenger

39.4.1 Files

Retrieve **MailFileScavenger.bcd** from the Mesa Release directory.

39.4.2 User interface

MailFileScavenger runs in the Executive window. To invoke it, type **MailFileScavenger MailFile.mail**, where **MailFile.mail** is the name of the mail file to be scavenged. Terminate the name with **RETURN**. MailFileScavenger copies your mail into its scratch file, printing out the number of every fifth message as it is processed.

When anomalies are detected in your mail file, MailFileScavenger prints out a short message such as **Message 5: existing count was 21 bytes too small**. This message means that the formatting information in the mail file used to distinguish individual messages was inconsistent with what MailFileScavenger believes to be distinct messages.

When MailFileScavenger is finished, it is a good idea to check any messages it complained about. These messages may be missing several characters or be malformed in other ways. You should also check neighboring messages—some of the characters in those messages might really be part of other messages.

After MailFileScavenger has finished copying and reformatting your mail into its scratch file, it pauses and asks if it should copy that file back into the original mail file. If there are not many error reports, type **Y** to confirm. MailFileScavenger copies the scavenged mail file back into the original mail file, deletes the scratch file, and quits. You may then invoke the scavenged mail file in your ArpaMailTool Options window. However, if there have been many error reports, you might want to copy the original file before allowing the MailFileScavenger to scavenge your file. To do this, cancel the command with **N**, copy the file, then run MailFileScavenger on the copy.

The mail file that MailFileScavenger produces should give you a readable mailfile. However, this mail file may have messages that are fragments of messages in the original file and/or duplicate messages. If you copied the original file before running the MailFileScavenger, you can compare the scavenged version to the original in order to determine if any text was lost. If you edit the scavenged mail file, you must run scavenger again.



A

Othello

Othello is a utility for managing Pilot volumes. A volume is an array of disk pages. A logical volume includes some overhead for keeping track of the array (such as the logical volume root page). A physical volume is the structure corresponding to a disk pack, and also includes some overhead information (such as the bad page table). Othello is used to initialize physical and logical volumes, to install boot files on logical volumes, to invoke a boot file on a particular logical volume, and to initiate scavenging of logical volumes. Othello handles all types of disks known to Pilot.

A.1 Files

Retrieve **OthelloDLion.boot** (for a Dandelion) or **OthelloTriDLion.boot** (for a Trident disk) from the Release directory.

A.2 Running Othello

Othello is booted from the disk in the normal development cycle. However, at certain times (such as at the beginning of the world, or whenever the disk is erased) Othello is booted from the Ethernet or a bootable floppy disk (on Dandelions).

To boot Othello from the Ethernet on a Dandelion, perform an AltBoot 3 (standard Etherboot), AltBoot 4 (diagnostic Etherboot), or AltBoot 6 (alternate Etherboot).

A.3 User interface

When Othello is ready to accept commands, it announces itself with the version and date, lists the processor ID in hexadecimal, octal, and NS standard formats, and reports the size of physical memory on the machine:

```
Othello 11.1 of 30-Dec-84 at 15:11:04
Processor = 0AA000176H = 25200000566B = 2-852-127-094
Memory Size = 768K bytes
>Online
Drive Name: RDO
```

A

Othello

After announcing itself, Othello waits for commands. Commands are entered on the command line. Othello displays the default when prompting (if a default is both reasonable and known). **RETURN** or **SPACE** will accept the default; **BS** or **BW** will allow editing of the default; any other character will erase the default and begin a new string.

Generally, **DELETE** will abort the current command and a question mark will list available options. A **RETURN** ends any command, and a space will end most commands. There are also a **Help** command and the usual command completion facilities.

Othello can also be run from a command file, invoked by

>@[requiredFileServer]<OptionalDirectory>CommandFileName.ext⁶

or by

>@
Command file: [FileServer]<OptionalDirectory>CommandFileName.ext⁶

This command causes the text of the remote file to be taken (almost) as if it were input typed to Othello. Command files can be up to 4096 bytes long and not nest (although they may chain). Command files terminate when an end of file is reached, or when Othello detects an error due to either a nonexistent command, an impossible-to-perform command, or an exceptional condition (such as volume needs scavenging or remote file unknown). During command files confirmation is suppressed; thus, command files should not contain any answers to "Are you sure?".

Although Othello is itself a Pilot client, it is prepared to run without relying on the existence of any volumes. As a result, it cannot write a typescript file.

Othello commands are presented below according to logical category.

Fine point: If an uncaught signal occurs (indicating an unexpected error), Othello displays the signal and message in octal. **CONTROL-P** will abort the current command. Consult **OthelloDLion.signals** for the meanings of unexpected signals. Depending on the problem, Othello may not be able to proceed after the error.

A.3.1 Accessible disk drives

The **List Drives** command displays the names of all the drives accessible by Othello.

> List Drives⁶
Rd0, ...

The drive names can be any of the following (# stands for a single digit):

RD# Shugart 1000 or 4000, Trident 80, 300, or 315, or Quantum 2040 or 2080

If a drive does not appear in the list, a hardware problem is likely. If the drive name *UnknownType#* appears in the list, Pilot is internally inconsistent (or the hardware or firmware is lying about device types).

A.3.2 Checking a pack

If you believe there are problems with your disk (which contains valuable data), you can use the **Check Drive** command to cause Othello to read every page on the disk, including the initial microcode area (see below).

```
>Check Drive€
Drive Name: drive€
Are you sure? [y or n]: Yes
Bad pages found:
10 1000 5540
```

Check Drive takes about a minute on a healthy Quantum 2040 and can be aborted with the **STOP** key.

The following errors may occur:

```
Drive not found!
Too many bad spots.
Othello can't check this device.
Can't access disk.
```

These errors are indicative of a bad pack. Attempting to format the pack again may correct the problem. If it does not, get a new pack.

A.3.3 Physical volumes

The physical volume on a drive must be brought on-line, making it known to Pilot, using the **Online** command before it can be used by Othello. The commands **List Drives**, **Create Physical Volume**, and **Check Drive** do not require that the pack be on-line. **Format** requires the drive be off-line.

```
>Online€
Drive Name: drive€
```

Once the volume is on-line, you may refer to it using its physical volume name (assuming it is unique). If the pack on the drive was just formatted, its name will be "Empty."

The operations on physical volumes are listed below. When a physical volume is specified, either a drive name, a physical volume name, or a combination of the two separated by a colon is acceptable. If there are two on-line volumes with the same name, the drive name must appear (perhaps by itself).

```
>List Physical Volumes€
Rd0:Trinity, ...

>Describe Physical Volumes€
Physical Volume Trinity on drive Rd0 (Shugart 4000) contains:
Volume Othello (type = debuggerDebugger) is 800 pages long
    starting at physical address 1
Volume Pilot (type = normal) is 15081 pages long
    starting at physical address 801
Volume CoPilot (type = debugger) is 15081 pages long
```

```
starting at physical address 15882
```

```
...
```

List Bad Pages is available to handle bad pages. It uses (decimal) page numbers, not disk addresses, to identify bad spots. These are the same numbers that CoPilot prints when unrecoverable disk errors occur.

```
>List Bad Pages€  
Physical Volume Name: drive:volume€  
10      4400      5000 ...
```

Use the **Describe Physical Volumes** command to determine which logical volume contains the bad page.

When the processing of a pack is complete, the volume may be taken off-line (unless the last operation was a **Boot**) with the command:

```
>Offline€  
Physical Volume Name: drive:volume€
```

The following messages may appear when using the above commands:

```
Not Found  
No physical volumes found  
No known bad Spots  
Bad Number!  
SpecialVolume.Error[notPilotPhysicalVolume]  
Ambiguous; please specify Device:PhysicalName  
Type ControlP to muddle on.....
```

The **Not Found** indicates that the last parameter (volume or drive name) could not be located. The **No physical volumes found** error probably indicates the correct drive was not on-line.

A.3.4 Logical volumes

Othello will configure a physical volume into a number of logical volumes. You specify the name, size, and type of each. You cannot add a logical volume to a physical volume without reconfiguring. The entire pack and the physical and logical volumes on it must be rebuilt together. This is accomplished with the **Create Physical Volume** command. An example appears below.

```
>Create Physical Volume€  
Drive Name: Rd0€  
Shall I try to find an old Bad Page Table?: Yes  
New physical volume name: Trinity€  
Number of logical volumes: [1..10]: 4€  
Logical volume 0  
Name: Othello€  
Pages: [50..45144]: 800€  
Type: debuggerDebugger€  
Logical volume 1  
Name: Pilot€  
Pages: [50..44394]: 15081€
```

```
Type: normal6
Logical volume 2
Name: CoPilot6
Pages: [50..29363]: 150816
Type: debugger6
Logical volume 3
Name: CoCoPilot6
Pages: [50..14332]: 150836
Type: debuggerDebugger6
Are you sure? [y or n]: yes
```

The **Create Physical Volume** command destroys all old information on the disk . If Othello discovers that the pack contains the remnants of a non-empty Pilot volume, it will ask for double confirmation before proceeding, since the **Create** operation will destroy the contents of the pack. In addition to the logical volumes, a physical volume name must be specified. The whole process takes about three minutes for a complete Quantum 2040 pack.

Up to ten logical volumes can be put on a single physical volume. One of the primary uses of logical volumes is to separate the debugger's working storage from the client's. Logical volumes therefore have the following types:

normal	a normal, client volume
debugger	a separate volume for CoPilot
debuggerDebugger	a volume for CoPilot's debugger
nonPilot	a "foreign" volume (which cannot contain Pilot files)

Once created, the following operations can be performed on logical volumes. In specifying a logical volume, the drive name is optional if the volume name is unique among the packs currently on-line.

```
>List Logical Volumes6
Rd0:Pilot, Rd0:Copilot, ...
```

An individual logical volume can be erased with the **Erase** command. All of its pages (except the bad spots) are marked free. Erasing a 15,000-page volume on a Quantum 2040 takes about 30 seconds.

```
>Erase6
Logical Volume Name: drive:volume6
Are you sure? [y or n]: Yes
Erase....complete
```

The Pilot internal scavenger can be invoked on an individual logical volume using the **Scavenge** command. This rebuilds the Pilot data structures on the volume and marks all known bad pages busy. Scavenging a volume may take a long time. The **Scavenge** command summarizes information in the scavenge log and displays any problems.

```
>Scavenge6
Logical Volume Name: drive:volume6
Are you sure? [y or n]: Yes
Scavenging....complete
volume repaired, log file complete
```

```
5 files on volumd  
No problems found
```

Fine point: The scavenger does not support repair of malformed files or client-level data structures.

The **Physical Volume Scavenge** command invokes the Physical Volume Scavenger, which reaps This Scavenger puts the physical volume so it can be brought on-line. This scavenger has two modes of operation: check and repair.

```
>Physical Volume Scavenge  
Drive Name: drive  
Repair? Yes  
Are you sure? [y or n]: Yes  
Scavenging...Complete  
No problems detected
```

The temporary files on a logical volume can be deleted using:

```
>Delete Temporary Files  
Logical Volume Name: drive:volume
```

The following messages may appear when using the above commands:

```
Not Found  
Drive not Found!  
This name is already in use; please choose another  
Illegal Type  
Bad Number!  
Volume's size decreased (because of bad pages) to nnn.  
No logical volumes found  
Ambiguous; please specify Device:LogicalName
```

A.3.5 Initial microcode, Pilot microcode, diagnostic microcode, germ, and boot files

Othello can be used to load various types of files from the network onto physical and logical volumes. The types of files of interest are: initial microcode, Pilot microcode and diagnostic microcode, germ, and boot.

Initial microcode resides at a fixed location on the disk (outside of any logical volumes). Pilot microcode files contain microcode to run the Mesa emulator and the devices in a hardware configuration. Diagnostic microcode, which need not be present, is used to detect faults within the machine. The germ is a small program that loads (and snapshots) Pilot core images when booting and when swapping to and from the debugger. Boot files are Pilot-plus-client programs produced by **MakeBoot**.

Pilot microcode, germ, and boot files can be installed on any Pilot-type logical volume. Although users can store germ and microcode files on each logical volume, only one file of each type, the one associated with the physical volume, can be loaded by the initial Pilot microcode.

Each physical volume contains a pointer to one of each of the following types of files: Pilot microcode, diagnostic microcode, germ, and boot. While each of these may come from a different logical volume in that physical volume, it is customary for all of the physical volume pointers to point to files on the same logical volume. A pointer is normally set to

the file of a given type most recently installed on the physical volume (see the different **Fetch** commands below to find out which have this option). The pointers may also be set with the **Set Physical Volume Boot** command. The microcode and germ files used are those read at the time of the last boot-button boot. If changed, microcode and germ files become effective only after the next boot-button boot is performed.

Because the files are retrieved from the network, the following commands (similar to commands in FTP or File Tool) are available. Standard communications-type messages will also appear.

```
>Login€  
User: user€  
Password: password€  
  
>Clearinghouse€  
Domain: domain€  
Org: organization€  
  
>Open€  
Open connection to server€  
  
>Floppy Open€  
Open connection to floppy disk€  
  
>Directory€  
Directory: directory€  
  
>List Remote Files€  
Pattern: pattern€  
  
>Close€  
closed
```

The **Clearinghouse** command sets the default domain and organization of both the logged-in user and the server to be connected to. It can be overridden in the **Login** and **Open** commands by specifying these values explicitly: user:domain:organization or server:domain:organization.

The **Floppy Open** command informs Othello that any subsequent remote file commands should be directed to the floppy disk drive. This closes any existing connection to a file server.

The **Close** command terminates any existing floppy or server connection. This command is automatically invoked by the **Boot**, **Power Off**, **Open**, **Floppy Open**, and **Quit** commands.

Fetch Boot File, **Pilot Microcode Fetch**, **Germ Fetch**, **Initial Microcode Fetch**, and **Diagnostic Microcode Fetch** all display the file's remote name (including creation date and version). In addition, if a "*" is detected in either the directory name or the file name, each matching remote file is displayed for you and a confirmation is required. If you confirm, that file is retrieved and the command terminates. If you do not confirm, you will be prompted with the next matching remote file.

The **Initial Microcode Fetch** command retrieves an initial microcode and installs it on a Shugart 4000 or 1000, Quantum, or Trident. It also formats the initial microcode area. The microcode will be used in the next boot:button boot.

```
>Initial Microcode Fetch
Drive Name: drive
Are you sure? [y or n]: Yes
File name: name
Formatting...Fetching...Installing...done
```

Fetching initial microcode takes about five seconds from a local server. The cursor displays FTP and twiddles while files are being transferred. In addition to FTP's messages, the following may also appear when using this command:

```
Not found!
Please open a connection first
Othello can't install microcode on that device.
Note page ddd is bad. (non-fatal)
Initial microcode page bad. (fatal)
Microcode too long. (fatal)
Flakey microcode page. (fatal; can't re-read a page just written)
```

There is no recovery from errors marked **fatal**. To proceed, the situation must be corrected.

The **Fetch** command will retrieve a boot file (in fact, it can *only* be used to retrieve Pilot boot files) onto a logical volume and make it the boot file for the logical volume. The logical volume can then be booted using the **Boot** command described below. **CoPilotDLion.boot** (for a Dandelion) should always be made the boot file for the debugger and debuggerDebugger volumes. Any Pilot boot file can be made the boot file for normal volumes.

```
>Fetch Boot File
Logical Volume Name: drive:volume
Boot file name: name
Fetching...Installing...done
```

A common way to update boot files is using a command file such as:

```
OSBU North
Xerox
Open FileServer
Directory BootFiles
Fetch Othello
Othello>Public>OthelloDLion.boot
Set Boot Othello
8}
Fetch Tajo
Tajo>Public>TajoDLion.boot
Set Boot Tajo
8}
Fetch CoPilot
CoPilot>Public>CoPilotDLion.boot
```

```

Set Boot CoPilot
8}
Set Physical Othello
Close

```

The time required to fetch a boot file depends on its size. Whenever possible, Othello installs new boot files on top of old boot files, saving space but resulting in the old file being lost. If the fetch fails, the old file is thus very likely to be invalid. In addition to FTP's messages, the following may appear when using the **Fetch** command:

```

Not found!
Please open a connection first

```

Once fetched, a boot file can be invoked using the **Boot** command.

```

>Boot€
Logical Volume Name: drive:volume€
Switches: switches€

```

The **Set Boot File Default Switches** command permits setting the default boot switches in a volume boot file. Othello will then supply the boot file default switches as the initial value of the switches when it prompts you for the **Switches:** part of the **Boot** command. You may then replace or edit those switches. Boot switches that cannot be entered from the keyboard may be specified in the form \nnn, where nnn is the octal code for the switch. Switches specified by octal numbers must be exactly three digits.

```

>Set Boot File Default Switches€
Logical Volume Name: drive:volume€
Switches: switches€
Are you sure? [y or n]: Yes

```

Pilot microcode, germ, and boot files on a logical volume may be deleted by

```

>Delete Boot Files€
Logical Volume Name: drive:volume€

```

The **Delete Boot Files** command is sometimes needed if **Fetch** is either interrupted while retrieving or used to get a file that is not a microcode, germ, or boot file.

The **Pilot Microcode Fetch**, **Diagnostic Microcode Fetch**, and **Germ Fetch** commands are similar to the **Fetch** command. However, as there is usually just one Pilot microcode file and one germ file for an entire physical volume, the question "Shall I also..." is asked:

```

>Pilot Microcode Fetch€
Logical Volume Name: drive:volume€
Pilot Microcode file name: name€
Fetching...Installing...done
Shall I also use this for the Physical Volume?: Yes

```

```

>Diagnostic Microcode Fetch€
Logical Volume Name: drive:volume€
Pilot Microcode file name: name€

```

```
Fetching...Installing...done
Shall I also use this for the Physical Volume?: Yes

>Germ Fetch6
Logical Volume Name: drive:volume6
Germ file name: name6
Fetching...Installing...done
Shall I also use this for the Physical Volume?: Yes
```

Fetching microcode and germ files is fairly fast since the associated files are small. The error messages are the same as for **Fetch**. The system will not begin using the retrieved pilot microcode, diagnostic microcode, or germ until the next boot-button boot. An installed debugger is invalidated when you change germs, so that when you begin using a germ different from the one in use when any debuggers were installed, you must reinstall all debuggers.

The **Set Physical Volume Boot Files** command designates the microcode, germ, and boot files to be associated with a physical volume. These files are the ones used by a boot-button boot.

```
>Set Physical Volume Boot Files6
Logical Volume Name: drive:volume6
Set physical volume boot file from this logical volume: Yes
Set physical volume diagnostic microcode file from this logical
volume: Yes
Set physical volume microcode file from this logical volume: Yes
Set physical volume germ file from this logical volume: Yes
Are you sure? [y or n]: Yes
```

The questions are asked only for files that exist on the logical volume. If none of the files exist, **Logical volume has null boot files** is printed and the command aborts. The files are not actually set until you answer "**Are you sure.**"

Fine point: When a boot file is re-fetched, the space is re-used. Thus, updating a logical volume boot file also updates the physical volume boot file if they are the same. If this is not desired, you should redo the **Set Physical Volume Boot Files**.

A.3.6 Time

While initializing, Othello tries to get the current time from an NS time service. If Othello fails to obtain the current time, it demands that you input time and time zone information before it allows any other commands.

The **Time** command displays the current date and time:

```
>Time6
Current time Tuesday 29-Dec-81 11:45;26 PST
```

The **Set Hardware Clock Upper Limit** command is used when a Pilot Client (i.e., CoPilot, Tajo, Services, or Workstation, as opposed to a UtilityPilot Client such as Othello or Prometheus) is booted, and time is not available from the network. Pilot will not believe a time provided by the system element clock that is greater than the expiration date stored

in the boot file. It will, however, believe a time from an NS time service that is greater than the expiration date stored in the boot file.

A.3.7 Routing tables and echo user

Othello has two commands for inquiring about the local Ethernet:

```
>Echo User€
Echo to: echoServer€
[My.net.address]=>[Gateway.net.address]=>[Server.net.address]
```

`echoServer` should be of the form `net.addr.socket`. Othello runs the test until you press any character, at which point it prints statistics and accepts further commands. As the test runs, "!", "#", and "?" are printed: "!" indicates a successful echo, "?" indicates timeout waiting for the echo, and "#" indicates reception of an unexpected packet. Often the sequence "?#" indicates a packet late in being returned.

The command

```
>Routing Tables€
```

causes Othello to display the current routing tables.

A.3.8 Accessing the debugger during early initialization of Pilot

During later stages of initialization, Pilot searches for an installed debugger. It looks on all volumes of a type one higher than the one on which the boot file resides. Thus if the boot file is on a volume of type normal, Pilot looks on volumes of type debugger; if the boot file is on a volume of type debugger, Pilot looks on volumes of type debuggerDebugger. Occasionally you may want Pilot either to get to a debugger early during its initialization or to use an installed debugger other than its normal choice. In these cases, use the `Set Debugger Pointers` command to set up the information Pilot needs. It will also accept an empty string for debugger, meaning to clear any debugger pointers that may have been set. You must also use this command to allow Utility Pilot clients (such as Othello itself) to use a debugger on the local disk.

```
>Set Debugger Pointers€
for debuggee Logical Volume: drive:volume€
for debugger Logical Volume: drive:volume€
Are you sure? [y or n]: Yes
Done...
```

This command takes the information needed for Pilot to find the debugger and writes it into the debuggee boot file.

When this command has been given, the debuggee boot file will continue to use the specified debugger until the debuggee boot file is erased or overwritten or the information is cleared.

The pointers written remain valid until you next erase the debugger volume or fetch other than a CoPilot boot file onto the debugger volume.

If these pointers have not been set up or are invalid, an early debugger call stops with an error code in the maintenance panel.

Fine point: This command allows you to have a client and a debugger on volumes of the same type. However, if any other systems are rooted on volumes of the same type as an installed debugger, it is necessary to always boot them (and good practice to boot the debugger itself) with the open-system-volume-only "%" boot switch. Otherwise, running one of the other boot files will delete the temporary files from underneath the installed debugger, leading to a Disk Label Check when the debugger is next used.

A.3.9 Exiting Othello

There are three ways to finish an Othello session. To exit normally, use

```
>Quit  
Are you sure? [y or n]: Yes
```

which will cause Othello to clean up after itself, and then (programmatically) press the boot button for you. The system will begin using the initial microcode, pilot microcode, germ, and physical volume boot file currently installed on the volume.

If you really want to stop the machine, use the command

```
>Power Off  
Are you sure? [y or n]: Yes
```

On some machines, this will actually turn the power off. On others, it will merely display a code in the maintenance panel and blank the display.

Finally, exit from Othello can be made by booting another program.

A.3.10 Special commands

Some special commands and options are available only to wizards. The **Wizard Mode** command enables these additional commands.

Getting started/Operations guide

This appendix describes how to start your new machine and how to use it.

The first thing to do is boot your Dandelion; that is, start it so that you can use it.

We assume here that the machine delivered to you already has a disk set up with a development environment and that the boot button will take you to Othello, the Pilot disk utility. (For more information about Othello, see Appendix A.) It is quite likely that you are in Othello now: that is, at the top of your screen there is a message of the form

```
Othello 10.0 of 20-Dec-82 12:13:14 PST
Processor = X'AA000176' = 25200000566B = 2-852-127-094
Memory Size = 512K Bytes
>Online RDO
```

You may already be in another logical volume, such as CoPilot, Tajo, or Star. The herald window (the long narrow window across the top of your screen) will tell you this. Or you may be staring at a blank display; if so, read the next section to learn how to boot your machine.

B.1 Booting

A complete software system is bound into a runnable package called a *boot file*. The process of invoking a boot file is called *booting*. Boot files may be stored on a rigid disk attached to the processor, on a floppy disk, or on an Ethernet *boot server*. Boot files may be invoked by the machine's *boot button*; by Othello, Tajo, CoPilot; or by another software system. Booting procedures vary somewhat on different Mesa processors.

Normally, the rigid disk is set up so that the boot button directly invokes Othello. Until the disk is initialized, the only ways that Othello may be invoked are by booting it from the Ethernet or a floppy disk on the Dandelion.

One important side-effect of a boot-button boot is that Mesa microcode and the Pilot boot loader are loaded into the machine from the source of boot data (rigid disk, floppy disk, Ethernet). This is the only time that they are loaded; thus, the ones loaded by the boot button will remain in use until the next boot-button boot. Software may initiate a boot-button boot; a *Quit* command to Othello does this.

B.1.1 The maintenance panel

On the front of every Mesa processor (behind the flap under the floppy drive) is a three- or four-digit numerical display called the *maintenance panel*, or MP. It is used to display status and error codes, particularly when the system is unable to give a more helpful error report. Next to the numerical display, you will find two buttons, marked **RESET** and **ALT B**. These two buttons are more commonly known as the *boot buttons*. They are referred to several times below.

Fine point: MP codes are displayed by the microcode and microcode diagnostics during booting; by Pilot's boot loader when it is running; and by Pilot during its initialization. The MP codes displayed by Pilot and its boot loader are the same on all Mesa processors. MP codes occurring normally during Pilot operation are described in the next section; codes displayed by Pilot to indicate unusual conditions and errors are listed elsewhere in this chapter. MP codes displayed by microcode and microcode diagnostics are specific to the processor being used and are described in other documents.

B.1.2 Standard booting

We will discuss now how to boot your machine from your disk or the Ethernet so that you are brought to Othello.

B.1.2.1 Disk booting

If an Othello has been installed on your machine, you can boot directly from your own hard disk.

To boot your machine, just press the left boot button on the maintenance panel. This causes diagnostics to run for about a minute and then boots the rigid disk. If you boot your machine often and want to skip the diagnostics, you can do an Alt-Boot-1, known as a One Boot. This is done by pressing both buttons, then releasing the left button. The lights will cycle from 1 to 10 until you let go of the right button; these numbers are the *boot options*. To boot from the hard disk without diagnostics, release the right button when 0001 is displayed by the maintenance panel lights. You may find the timing a bit tricky at first. If you haven't gotten it right, you can try again immediately.

The boot options are:

```
0000 SA4000, SA1000, or Trident drive 0 diagnostic boot (normal boot)
0001 SA4000, SA1000, or Trident drive 0 non-diagnostic boot
0002 Floppy non-diagnostic boot
0003 Ethernet non-diagnostic boot of Othello
0004 Ethernet diagnostic boot of Othello
0005 Floppy diagnostic boot
0006 Reserved for Ethernet boot of experimental microcode/software
0007 Trident drive 1 diagnostic boot
0008 Trident drive 2 diagnostic boot
0009 Trident drive 3 diagnostic boot
0010 Floppy head cleaning function
```

The screen will flash as your Dandelion boots. Watch the maintenance panel numbers. They should run through a sequence of diagnostic numbers like 910, 920, 930, 990. You may not see all of these, but if the machine hangs with some other number in the lights, your boot has probably not been successful (see below). Shortly after the 990 appears, Othello should announce itself and prompt for input:

```
Othello 11.1 of 20-Dec-84 12:13:14 PST
Processor = OAA000176H = 25200000566B = 2-852-127-094
Memory Size = 768K Bytes
>Online RDO
```

If booting is not successful, chances are either that your machine doesn't have an Othello on it or that it has hardware problems.

B.1.2.2 Ethernet booting

To boot Othello from the Ethernet (without running diagnostics), proceed as described above, only this time perform an Alt-Boot-3: wait until 0003 is displayed in the maintenance panel lights before you release the right button. (An Alt-Boot-4 will perform a diagnostic Ethernet boot.)

B.1.2.2.1 Time setting requirements

If a Dandelion is not connected to an Ethernet with an operational NS time service on it, Othello will require that you set the date, time, and local time zone parameters before proceeding. You should be careful to give accurate information, because other users and programs depend on it.

B.1.2.3 Floppy booting

To boot from a floppy, insert an Othello-bootable floppy disk in the floppy drive (label side up) until it locks in, close the drive panel, and perform a "5" or a "2" alternate boot. If you are unable to boot Othello either from the Ethernet or from a floppy, consult your local system administrator.

Note: It is important to remove floppy disks from the drive when not in use to lengthen the life of both the disk and the drive.

To clean the floppy disk drive, do the following: Insert a floppy cleaning diskette in the drive and do a "10" alternate boot. When the MP displays 0076, press the ALT B button. 0077 will be displayed for about 15 seconds while the drive is being cleaned; then 0076 will be displayed once again. Remove the cleaning diskette from the drive.

B.1.2.4 Maintenance panel codes during initialization

When a boot file is invoked, Pilot displays a sequence of maintenance panel codes to indicate progress during its initialization:

```
900  boot loader entered
910  boot loader action running (such as inLoad, outLoad)
920  boot loader driver running (such as disk, Ethernet)
930  Pilot Control and MesaRuntime components being initialized
```

```

940  Pilot Store component being initialized
947  waiting for disk drive to become ready
950  logical volume being scavenged (If a logical volume being booted or
      opened is in an inconsistent state, Pilot will display 950 while it scavenges
      (verifies the contents of the physical volume. The amount of time required depends
      on the size, occupancy, and fragmentation of the logical volume.)
960  temporary files from previous run being deleted
970  client and other non-boot-loaded code being mapped
975  transaction crash recovery
980  Pilot Communication component being initialized
990  PilotClient.Run called

```

B.1.2.5 Maintenance panel codes during XDE initialization

The Xerox Development Environment may also display maintenance panel codes.

```

9950 Mesa file system being verified. (If a logical volume being booted or opened is in
an inconsistent state, the Mesa file system will display 9950 while it scavenges;
that is, verifies the contents of the logical volume. As with scavenging the
physical volume, the amount of time required depends on the size, occupancy, and
fragmentation of the logical volume.)

```

B.1.2.6 Booting other volumes from Othello

Once Othello has been started, you can boot any of the other logical volumes from inside it. To boot another volume such as Tajo or Star from within Othello, you need to give the boot command followed by the name of the logical volume you want. For example, to boot your Tajo volume, you would say:

```

>Boot6
Logical Volume Name: Tajo6
Switches:6                                --a carriage return gets the default switches for the volume

```

B.2 Setting up volumes: initializing your system

From Othello you can examine the structure of your disk to see the logical volumes it contains. Briefly, a volume is an array of disk pages. A logical volume includes some overhead for keeping track of the array. A disk or disk pack is called a physical volume. Each physical volume is divided into one or more logical volumes. Different logical volumes may be used to contain different systems, such as Star, Tajo, CoPilot, and Othello. The logical volume is conventionally given the name of the system it contains. Separate logical volumes may also be used to segregate the data of a system into useful subsets; for example, Star typically stores all user data on a separate volume named User.

Each logical volume has a volume type. This type is used to keep the working storage of the debugger separate from the system that it is debugging . Logical volumes have the following types:

Othello Keyword	Example	Volume type
normal	Star, Tajo	a normal client volume
debugger	CoPilot	the debugger for normal volumes

debugger	Debugger	CoCoPilot	debugger of a debugger volume volume does not contain Pilot files
	nonPilot		

B.2.1 Example of initializing volumes

The chart below gives examples of configuring an SA4000 or SA1000 disk starting from scratch. It assumes that there is nothing of value on the disk and that the entire disk is to be used for Pilot-based systems.

These alternatives each will create a number of logical volumes on the single physical volume. Each of these logical volumes will hold the files for semi-independent Pilot worlds. There is considerable room for individual taste, depending on your needs and which systems you wish to be able to run.

Generally, it is best to allocate as much space as possible for the volume where you plan to do most of your work or store most of your files.

Programmer, Quantum 2040 disk (64,000 pages after formatting):

alternative	1	2	3	4	minimum	type
Othello	1400	1400	1400	1400	1200	normal
Tajo	25,600	18300			4000	normal
CoPilot	1100	18300	33600	30600	7500	debugger
CoCoPilot	9000	9000	12000	12000	9000	debugger
Star	8000	8000	8000	11000	8000	normal
User	9000	9000	9000	9000	9000*	normal

*allows approximately 4000 free pages after installation of data files.

Programmer, SA4000 disk (45,000 pages):

alternative	1	2	3	4	minimum	type
Othello	1200	1200	1200	1200	1200	normal
Tajo		5000			4000	normal
CoPilot	17600	13800	29300	18500	7500*	debugger
CoCoPilot	9000	9000			9000	debugger
Star	8000	8000	8000	8000	8000	normal
User	9000	9000	9000	18600	9000	normal

*7500 is minimum for 768K memory. Add 1000 pages for each additional 256K bytes.

Non-programmer, SA1000 disk (16,000 pages):

alternative	1	2	3	4	minimum	type
Othello	1400	1400	1200	1200	1200	normal
Tajo	14600			6400	4000	normal
CoPilot			7800		7500	debugger
Star		14600	7000		6700	normal
User				8500	8500	normal

Note: The minimum of 1200 pages for the Othello volume includes boot file, microcode, and germ storage. If you store microcode in another volume, decrease the minimum by 80

pages. Storing the germ elsewhere reduces the minimum by an additional 40 pages. If you wish to store diagnostic microcode on Othello, add at least 160 pages to the minimum figure.

At present there is no way to alter the distribution of space among the logical volumes without recreating the entire physical volume (and destroying its contents!). Therefore, the initial distribution of space should be considered carefully. It is critical to make each volume at least the specified minimum size because some systems can fail in ungraceful ways if the volume is full.

B.2.2. Booting volumes from other volumes

If you are in either CoPilot or Tajo, you can boot another volume by moving the mouse until the cursor is in the Herald window and holding down both mouse buttons to get the **Boot From:** menu. (If you have a three-button mouse, holding down the middle button performs this function.) While you hold down the buttons, move your mouse until the name of the volume you want to boot is highlighted, and then release the buttons. You will see your cursor transformed into the image of a mouse. This is the system's way of asking you to confirm or abort the boot request. To confirm, click the left mouse button. Clicking the right mouse button causes the boot request not to be executed.

B.2.3 Boot switches

Boot switches control the initialization of the Xerox Development Environment. The switches are passed to Pilot by the agent that invoked the boot file (such as Othello). The agent invoking a boot file normally has a mechanism for the user to specify switches. A default set of switches may also be written into a boot file, and the invoking agent may choose to use them. If the invoking agent is the boot button, the default switches are always used.

Fine point: Boot switches are eight-bit characters. Various ranges of the switches are allocated for Pilot, for the Xerox Development Environment, and for other product systems like Star. The current assignments for Pilot and the XDE are given below.

The most common boot switches are:

% Open only the system logical volume.

Normally Pilot opens all logical volumes on the system physical volume that have the same volume type as the volume being booted. This switch causes Pilot to open only the volume being booted.

* Bring all physical volumes on-line automatically.

During normal initialization, Pilot automatically brings on-line only the physical volume that contains the system being booted; other physical volumes then may be brought on-line by client software. This switch causes Pilot to bring on-line all attached physical volumes.

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

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

5 Go to the Ethernet for a debugger.

This switch instructs Pilot to go to the Ethernet when it needs a debugger. This supersedes the presence of an installed debugger on the attached disk and/or debugger pointers that may have been set in the boot file. If Pilot needs to map log (see below), it will use an Ethernet debugger to do so.

6 Turn owner checking on for `Heap.systemZone` and `Heap.systemMDSZone`.

7 Disable map logging.

For the debugger (CoPilot or CoCoPilot) to access the Pilot virtual memory, it must be aware of the current mappings between virtual memory and backing storage. It does this by consulting the virtual memory map log normally produced by Pilot. Map logging is disabled by this switch, thus increasing performance, but seriously limiting the ability of the debugger to diagnose problems.

8 Create a keyboard interrupt key watcher.

This switch instructs Pilot to call the debugger when the `LOCK` and both `SHIFT` keys are held down and the `STOP` key is pressed. The debugger will report "Pilot Emergency Interrupt." Since the Pilot process doing the job runs at the highest priority, this feature is useful for debugging Pilot itself and user input handlers. You should not attempt to Interpret Call from the debugger back into the debuggee because of the high priority level involved.

NOTE: The keytop name `STOP` is for the American Level IV keyboard; consult the keyboard mapping documentation for the equivalent key on other keyboards. Since the keys used are on the standard keyboard, a system with only a character terminal attached cannot access this feature.

9 Simulate a 256K Dandelion memory size.

This switch is useful for doing performance testing of product software on large-memory machines and for saving inload and outload file space.

- { Use a small data space backing storage cache.
- | Use a medium data space backing storage cache.
- } Use a large data space backing storage cache.

Pilot allocates a cache of file space to be used for backing storage for data spaces. (The file space is allocated on the system volume.) Poor performance may result if this cache is too small for an application's needs. These switches allow you to specify the size of this cache. If no switches are given, Pilot will use an amount based on the size of the system volume. In the current version of Pilot, the actual number of pages used are: small-- 750; medium-- 1400; large-- 2000; default-- 1/16th of the pages of the logical volume, with a minimum of 250 and a maximum of 1000.

Many Pilot boot switches are normally of interest only to the Pilot implementors themselves:

- \$ Go to debugger early in Transaction initialization.
- & Hang with a maintenance panel code in lieu of going to the debugger.
- 0 Go to debugger as soon as possible ("Key Stop 0").

To use the 0 switch, you must have Set Debugger Pointers in the boot file or be using an Ethernet debugger.

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

The debugger usually will not be able to set breakpoints in code until it has been map logged. Also, note that from the time that the boot button is pushed on a Dandelion up to shortly after key stop 1 in the system being invoked by the boot button, only an Ethernet debugger may be used; if you try to use a local debugger, you will get an MP code of 902 (see MP code list).

- :
- :
- :
- < Act as if there is no Ethernet 1 attached to the system element.
- = Do not initialize the Communication package at system start-up.
- > Act as if there is no Ethernet attached to the system element.

\375\ Disable map logging.

Full map logging is the default case when Pilot is booted, if there is a debugger present. Full map logging includes occasionally going to the debugger to clean up the log. If there is no debugger present, map logging proceeds until the log file fills up and then logging is disabled. This situation may be forced by setting key switch 375. Key switch 7 will cause Pilot to stop map logging when **PilotClient.Run** is called; this switch overrides the 375 switch.

\376\ Delete Pilot boot loader ("germ") and reclaim the memory it occupies.

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

B.2.4 Xerox Development Environment boot switches

The Xerox Development Environment boot switches are listed below. Unless otherwise stated, they apply both to Tajo and CoPilot. The switches must be upper-case letters, as shown

- N Do not process **User.cm** during initialization.
- S Reserved for the **CommandCentral** tool.
- T Reserved for the **CommandCentral** tool.

- V** Force a scavenge of the XDE file system. The file system scavenger produces a log file that describes the problems found and the recovery action taken. The file is named **MScavenger.log** on the root directory of the system volume.

If the development environment knows that its file system may be inconsistent, it automatically verifies the contents ("scavenges") the contents of its file system during initialization. This switch can be used in exceptional circumstances to force it to scavenge even though it believes that the file system is consistent.

- W** Install CoPilot and remain executing there.

Normally, CoPilot ends installation by booting the physical volume. If CoPilot is booted with the **W** switch, it will install itself and then wait for commands. This is useful if you wish to teledebug immediately or use some development environment tool in the CoPilot volume.

B.3 Installing boot files

Each of the logical volumes on your disk can have a boot file installed on it. The boot file is the program that receives control when the volume is booted.

Often you will want to update your boot files. For example, when a new version of a system is released, you may want to convert to that new version. To do so, you will need to know how to fetch and install boot files. Appendix A contains full instructions and examples for installing boot files with Othello.

B.3.1 Initializing debuggers

If you will be programming in XDE, you should initialize your debugger volumes by booting them. Booting a debugger volume is unlike booting other volumes; it causes the debugger to be installed, but will return you to Othello immediately afterwards (unless the **User.cm** on the debugger volumes say to boot another volume). Boot the debugger volumes one by one. It is best to boot CoCoPilot before booting CoPilot.

```
>Online€
Drive Name: RDO€
>Boot€
Logical Volume Name: CoCoPilot€
Switches:€
```

The maintenance panel numbers will sequence through 910, 920, 930, 940, 950, 960, 970, 975, 980, 990, although you may not see all of them go by. You will then see CoCoPilot initializing itself (creating windows, unload files, herald, etc.). When it has finished, it will boot your physical boot volume, which is Othello, setting you up to repeat the procedure for your CoPilot volume.

If you boot your debugger volumes by hand using Othello (as described above), remember to boot CoCoPilot before booting CoPilot. Each debugger (and debuggerDebugger) volume is designed to boot its client volume after it has finished initializing itself. Thus, CoCoPilot will always attempt to boot CoPilot, and CoPilot will always try to boot Othello. Because most programmers will want to work in the CoPilot volume, the following entries

in the **User.cm** file will set things up to initialize all debuggers quickly each time you boot, but to remain executing in CoPilot:

```
[CoCoPilot: Debugger]
Boot: CoPilot/W
InitialState: Active
--this "/W" switch boots CoPilot and
remains executing there

[CoPilot: Debugger]
Boot: Othello/2
InitialState: Tiny
--called "Key Stop 2"; initializes
Othello but breaks before
presenting user interface,
to return to CoPilot interface.
```

If no boot switch is given in the **User.cm** file, Othello will boot and run. For more information about debuggers and boot switches, see the section about the Debugger.

B.3.2 Setting debugger pointers

Programmers often find it convenient to do their development work in CoPilot. To access CoPilot for this purpose, after installing the debugger, you will want to set up your Othello volume so that you can enter CoPilot (and your development environment) immediately. To do this, you must set the debugger pointers for your Othello volume:

```
>Online
Drive Name: RD0
>Set Debugger Pointers
for debugger Logical Volume: Othello
for debugger Logical Volume: CoPilot
Are you sure? [y or no]: Yes
Done...
```

The debugger pointers will not be effective until Othello is booted, so boot Othello now. You are now ready to go to work; if you are using Tajo, just boot it as before. To invoke CoPilot, simultaneously depress the **SHIFT** and **STOP** keys (henceforth to be called **CALL-DEBUG**).

B.4 Installing the development environment

The Xerox Development Environment provides the tools for manipulating files, building and developing programs, and handling text and other objects.

The rest of this section assumes that you have some knowledge of the XDE world, including the edit and file tools. If you are unfamiliar with the user interface to tools, you should refer to the General Tools introduction.

B.4.1 Tools

You will want to install the tools that compose the development environment in either or both of your CoPilot and Tajo volumes. To do so, you will need to retrieve a number of files from the file server where they reside. The File Tool is designed to do this.

When you boot your Tajo or CoPilot volume, a copy of the File Tool (along with other tools) is automatically loaded for you by the boot file. Run your File Tool from the Executive and use it to retrieve any other tools or files you may need.

A common way to load recent versions of frequently used tools is using a command file. For example:

```
FTP FileServer
  Directory/c <Tools>
    Retrieve/ua
      Binder>Public>Binder.bcd
      Brownie>Public>Brownie.bcd
      Compare>Public>Compare.bcd
      Complier>Public>Compiler.bcd
      DebugHeap>Public>DebugHeap.bcd
      FileSystem>Private>RFileServer.bcd
      Find>Public>Find.bcd
      Formatter>Public>Formatter.bcd
      FTP>Public>FTP.bcd
      IncludeChecker>Public>IncludeChecker.bcd
      Lister>Public>Lister.bcd
      Other>Friends>Install.bcd
      Packager>Public>Packager,bcd
      Print>Public>Print.bcd
      ReleaseTools>Public>Statistics.bcd
      Spy>Public>Spy.bcd
```

B.4.2 The user command file

The user command file, `User.cm`, contains default information for many of the Xerox Development Environment services. These defaults are used by the system, some at booting time and some during normal running of the system.

To create one for yourself, use the File Tool to retrieve `SampleUser.cm` from `Doc>` onto your Tajo and CoPilot volumes. Edit it by replacing the fields all currently delimited by angle brackets, and rename it to be `User.cm`. Further information on the `User.cm` can be found in the General Tools introduction.

B.5 Recovering from disasters

When you are running and your system is healthy, you will see either a 990 (if Othello, Tajo, or CoPilot is booted) or an 8000 (Star) code in the maintenance panel lights. These are normal. Sometimes you will encounter a situation that may appear to be abnormal: you may be stuck at an unfamiliar maintenance panel code; your system may be frozen (for example, your mouse has stopped tracking or the system has quit listening to your keystrokes and mouse actions); or you may find that the volume you had been working in has gone away and you have unintentionally landed in the debugger (CoPilot or CoCopilot).

In the first two cases, the first reasonable thing to do is to reboot. If the situation recurs when you try to return to what you were doing, contact your local support person.

If you have suddenly landed in the debugger (the Herald window at the top of the screen will tell you that you are in CoPilot or CoCoPilot), it is often useful to consult with someone to find out what went wrong with the software to bring you there and perhaps submit an Action Request (AR) if the fault is determined to be that of the system. To recover from the situation, you should then reboot the volume you had been using by means of the **CoPilot BootMenu** command.

Sometimes after the system has crashed or you have terminated abnormally, the next time you reboot the volume may take a few extra minutes. This is because the system needs to verify the contents of the volume (scavenge the disk) before proceeding. When this occurs, there will be a 950 in the maintenance panel lights. This is a normal occurrence. If you are sitting in 950, be patient; your system will reboot. The 950 means that Pilot is scavenging the disk to make sure its data structures are correct. Tajo and CoPilot will show 9950 in the maintenance panel while they scavenge the development file system to make sure all its data structures are correct. This may take a while if you have a large number of files on the disk.

B.5.1 Dandelion boot microcode maintenance panel error codes

Following is a list of the most common maintenance panel codes displayed by the Dandelion boot microcode. A full listing of Dandelion MP codes is available in the system administrator's manual for OS 5.0. In general ,MP codes reporting fatal errors blink; codes reporting status or progress through a sequence of actions are steady. Each panel code is followed by an indication of what action you might take to remedy the situation:

- 149 This code normally is displayed for only a few seconds. On machines equipped with an SA4000 disk drive, 149 will be displayed for about 90 seconds after turning on the power. Persistent 149's should be reported to your hardware support group.
- 151 This can occur when booting after turning power on. Try rebooting. If that fails, report it as a hardware error.
- 206 The diagnostic microcode cannot be loaded. Refetch (using Othello's "Diagnostic Microcode Fetch" command) and be sure to answer "Yes" to the question about using it for the physical volume.
- 208 The germ cannot be loaded. Refetch (using Othello's "Germ Fetch" command) and be sure to answer "Yes" to the question about using it for the physical volume.
- 323 Diagnostic microcode will wait displaying 323 if the internal clock has not been set (for example, just after turning power on). Depress the **ALT B** button and hold it until the MP advances to 324 .

B.5.2 Pilot maintenance panel codes for errors

Pilot and its boot loader display maintenance panel codes when it is not possible to get to CoPilot to report an error (typically during booting). These codes are the same on all Mesa

processors. Codes displayed by Pilot during normal operation are described in the system administrator's manual for OS 5.0.

- 901 boot loader out of frames (Pilot bug)
- 902 unexpected trap or kernel function call (Pilot bug or hardware fault)
- 903 attempt to start an already started module (Pilot bug)
- 904 page or write protect fault (Pilot bug)
- 905 boot loader not compatible with initial microcode
- 906 boot loader not compatible with Pilot in boot file
- 909 boot loader SIGNAL or ERROR (Pilot bug)
- 911 boot loader not compatible with physical volume
- 912 boot loader not compatible with MakeBoot used to produce boot file
- 913 no physical boot file installed
- 914 boot file contains invalid data
- 915 Ethernet debugger server in control (see the chapter on the debugger for instructions on teledebugging and the **Re****MOTE debugger** command. Either (1) the "5" boot switch has been used, (2) CoPilot was not correctly installed, or (3) it is too early in initialization to find local debugger. (Use Othello's **Set Debugger Pointers** command and try again.)
- 916 boot file won't fit in real memory
- 919 boot loader has transferred control back to Pilot, who has hung
- 921 hard error on device being booted (physical or logical volume never initialized; hardware error. If booting, refetch the boot file and retry the operation; if going to/from CoPilot, reinstall CoPilot).
- 922 operation on boot device not completed in expected time (see your network administrator or try again).
- 923 broken link in chained boot file (if booting, try re-installing the boot file; if interrupting to CoPilot from Othello, use the **Set Debugger Pointers** command to correct the pointers from Othello to CoPilot; if going to/from CoPilot, reinstall CoPilot).
- 924 Ethernet boot server not responding (see your network administrator)
- 925 unexpected packet sequence number or size (see your network administrator).

926 Ethernet debuggee server trying to find a Pup / EthernetOne 8 bit address

931 Pilot not compatible with MakeBoot used to produce boot file

932 trap before trap handler initialized (verify that you have consistent versions of microcode, germ, and bootfiles).

933 Pilot not compatible with boot loader (refetch boot file).

934 boot file's StartList contains bad data

935 need Ethernet debuggee server but boot loader used does not have that capability(install smarter boot loader and try again)

936 waiting for microcode debugger ("&" or \376\boot switch used)

937 trying to get the time from either hardware clock or Ethernet (If code persists for more than a few seconds, see your Time Server administrator)

938 running clean up procedures (e.g., before going to debugger).

939 System.PowerOff called but no power control relay

948 system physical volume needs scavenging (use Othello's Physical Volume Scavenge command).

965 insufficient file space for data space backing storage (specify smaller size with boot switch).

981 trying to find a Pup/Ethernet-1 8 bit address (you are trying to initiate PUP communications, but there is no PUP name server on your network or your workstation is not registered with it, since PUP is an obsolete Xerox internal protocol.)

B.5.3 Pilot error messages

For some serious errors, Pilot goes to the debugger to report errors. Some of the error messages are listed below:

Address Fault

Address Fault (address past end of processor VM)

An address in virtual memory has been referenced that is neither mapped (such as **Space.Map**), nor implemented by the processor hardware. See the Debugger chapter or the introduction to System Building Tools for debugging procedures.

WriteProtect Fault

An attempt was made to store into an address in virtual memory that is currently read-only.

Mapped off file - Helper

A file has been deleted or shortened when there was a space mapped to it, which is neither permitted nor explicitly checked for by Pilot.

Disk Label Check

The identity of a disk page does not match what Pilot thinks it should be. Pilot attempted to read or write a page on the disk and found that the label on that page described a file and page number other than the one Pilot thought it was accessing. One possible cause of a label check is that a page has been marked bad on a logical volume without subsequently running the client scavenger on that volume. Another common cause of a label check is that a volume containing an installed debugger has been opened for writing by a program running on a client volume. This open causes temporary files on the debugger's volume to be deleted. When the debugger is next invoked, these files--which it had been using--will have disappeared from under it, resulting in a label check. The most common root cause of this situation is having *any* system installed on a volume of the same type as an installed debugger *and* not always booting that system with the "%" boot switch.

Out of VM for resident memory

Pilot has a pool of virtual memory that is used to contain resident data. This message indicates that that pool has been exhausted. Items that are allocated in this pool are: (a) dynamically created local frames, (b) global frames of dynamically loaded configurations that consist of a single module, and (c) Pilot internal data. One possible cause of this error is a procedure recursively calling itself forever, thus requiring an infinite supply of local frames. Another possible cause is the simultaneous use of several procedures that require large local frames; these can exhaust the pool fairly quickly. It is also possible for the Pilot virtual memory system to malfunction, generating this message.

Volume vanished between Descriptor and PageGroup (Helper)

A volume has been put off-line when there was a space mapped to it, which Pilot neither permits nor explicitly checks for.

Unrecoverable disk error on page pageNumber

A disk page could not be read. The standard thing for you to do is to run the Pilot and client scavengers for that volume. However, it is sometimes possible to fix up these pages by rewriting them so that they can be read. This may result in the loss of data.

The tool **PageScavengerTool.bcd** is available for doing this. It allows you to fill in a list of page numbers (usually it should be a page reported by Pilot in its call to CoPilot). If the **Rewrite** switch is left on, then the tool is permitted to rewrite the page. Otherwise, the tool will not permit the page to be written with potentially incorrect data. The tool writes the result of the scavenge, including an indication of what action you should now take.

The indicated action is self-explanatory, except that the action **pvScavenge** is to be interpreted as: run the physical volume scavenger. If it reports no problems, then run the logical volume scavenger on the volume containing the offending page.

Unrecoverable disk error: labelError

A disk page could not be read because of hardware errors. Contact your local support group.

B.6 Ending a Session

When you are done using a software system, you can either put that system in some "idle mode," boot some other system, or turn the power off. For the Xerox Development Environment, a tool called DMT is normally activated when the system is idle for long periods. Star has a similar facility that is automatically invoked when you log off. Avoid turning power off.

To boot some other system (or turn power off), you should use the facilities provided by the system you are using so that it gets a chance to put itself in a consistent and inactive state. To boot some other Pilot logical volume from Tajo or CoPilot, you can use the **Boot from** menu commands available in the window running across the top of the screen.

If you push the boot button while CoPilot is running, you must re-install it. Even if CoPilot is sick, you may still be able to use its **Boot** menu command to immediately re-install it. If you push the boot button while CoCoPilot is running, you must re-install first CoCoPilot and then CoPilot.



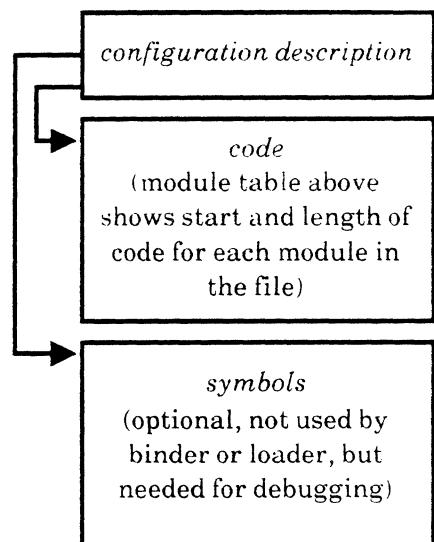
TableCompiler

The TableCompiler is a utility for creating files in the format of Mesa object files (whose filename usually ends in ".bcd"). This allows you to bind information other than programs into a Mesa configuration (fonts or microcode files, for example). The TableCompiler was produced by providing a single user interface to two programs: the ModuleMaker and the StringCompactor. The ModuleMaker takes a file with arbitrary contents, such as a font, and prefixes it with the proper header. The StringCompactor reads stylized Mesa programs consisting of arrays of string constants, squeezes the characters together into an array of characters, and produces auxiliary arrays of offsets and lengths.

C.1 Mesa object file format

In order to understand the operation of the TableCompiler, it is necessary to have some understanding of the format of Mesa object files.

Mesa object files all begin with a data structure called a *binary configuration description* (whence the default file extension ".bcd"). It contains the information need by the binder to resolve external references (imports and exports) and to find the code and symbols for any modules contained in the object file. The compiler creates a file that contains a configuration description for the degenerate configuration (a single module), and which also contains both the code and symbols for that module. Further binding and packaging usually places only the code in the same file with the configuration description, leaving the symbols in their original file, or copying them to a ".symbols" file.



C.2 Using the output

The output of the TableCompiler is an object file whose configuration description names a single module whose "code" is the table compiled information. The **EXPORTS** portion of the description says that a single **PROGRAM** is exported, either to **SELF**, or to a named interface.

The client program imports this **PROGRAM** and calls a runtime procedure that returns the address of the data. For example:

Let **TableData.bcd** be a table compiled module that exports **SELF**:

```
DIRECTORY
  Runtime,
  TableData;
Foo: PROGRAM IMPORTS Runtime, TableData =
  BEGIN
    base: LONG POINTER = Runtime.GetTableBase[TableData];
    ... -- base now points to the data part of the table compiled information
```

Similarly, let **TableData.bcd** export **TableData** to the interface **TableDefs**:

```
DIRECTORY
  Runtime,
  TableDefs USING [TableData];
Foo: PROGRAM IMPORTS Runtime, TableDefs =
  BEGIN
    base: LONG POINTER = Runtime.GetTableBase[TableDefs.TableData];
    ... -- base now points to the data part of the table compiled information
```

The second example, exporting to an interface, allows new data to be table compiled without having to recompile Foo. On the other hand, if the data changes slowly, using the first method means one less interface to keep track of.

C.3 ModuleMaker

The input to the ModuleMaker is a file with arbitrary contents. The only restriction is that it be less than 64K bytes long, as the length of the file must be contained in the body table of the configuration description.

Suppose **TableData.data** is a file to be table compiled. The executive command line

```
>TableCompiler.~ TableData.data/m
```

will create the file **TableData.bcd**, which is a **PROGRAM** exporting only itself.

The executive command line

```
>TableCompiler.~ TableData.data/m TableDefs/i
```

will create the file **TableData.bcd**, which is a **PROGRAM** exporting **TableData** to the interface **TableDefs**.

The name of the **PROGRAM** exported will be the same as the name of the input file. The extension on the input file name (.data in the example) is stripped off, and the root of the file name is used as the module name. N.B. the name given on the command line must be properly capitalized. See section 4 for further operational details.

C.4 StringCompactor

The input to the StringCompactor is a stylized Mesa program. It is most easily understood by looking first at an example.

C.4.1 Example

Consider the Mesa program **ErrorTab**, where most of the **ErrorMessage** entries are omitted for the purpose of this example

```

DIRECTORY
  Log USING [ErrorCode],
  Tree USING [NodeName];
ErrorTab: PROGRAM =
BEGIN
FnName: ARRAY Tree.NodeName[assignx..uparrow] OF STRING = [
  "MIN", "MAX", "LONG", "ABS", "ALL", "SIZE", "FIRST", "LAST",
  "DESCRIPTOR", "LENGTH", "BASE", "LOOPHOLE", "NIL"];
ErrorMessage: ARRAY Log.ErrorCode OF STRING = [
  "FATAL COMPILER ERROR",                                -- compilerError
  "unimplemented construct",                            -- unimplemented
  "unspecified error",                                 -- other
  ... -- and many more (all possible compiler error messages)
  "will use unsigned comparison"];                      -- unsignedCompare
END.

```

The executive command line

```
>TableCompiler.~ ErrorTab.mesa/-a
```

will create the file **ErrorTab.bcd**, which is a **PROGRAM** exporting only itself, and will also produce the file **ErrorTabFormat**, containing the following text that can be inserted into a program or interface.

```

CSRptr: TYPE = LONG BASE POINTER TO CompStrRecord;
CompStrDesc: TYPE = RECORD [offset, length: CARDINAL];
CompStrRecord: TYPE = RECORD [
  stringOffset: CSRptr RELATIVE POINTER TO StringBody,
  FnName: ARRAY Tree.NodeName[assignx..uparrow] OF CompStrDesc,
  ErrorMessage: ARRAY Log.ErrorCode OF CompStrDesc];

```

Suppose now that you want to print the error message associated with some value, say **code**, of the enumerated type **Log.ErrorCode**. The program fragment below shows how to obtain a **String.SubStringDescriptor** for the message.

```

et: CSRptr = Runtime.GetTableBase[ ErrorTab];
...
ss: String.SubStringDescriptor ← [
  base: et.stringOffset,
  offset: et.ErrorMessage[code].offset,
  length: et.ErrorMessage[code].length];

```

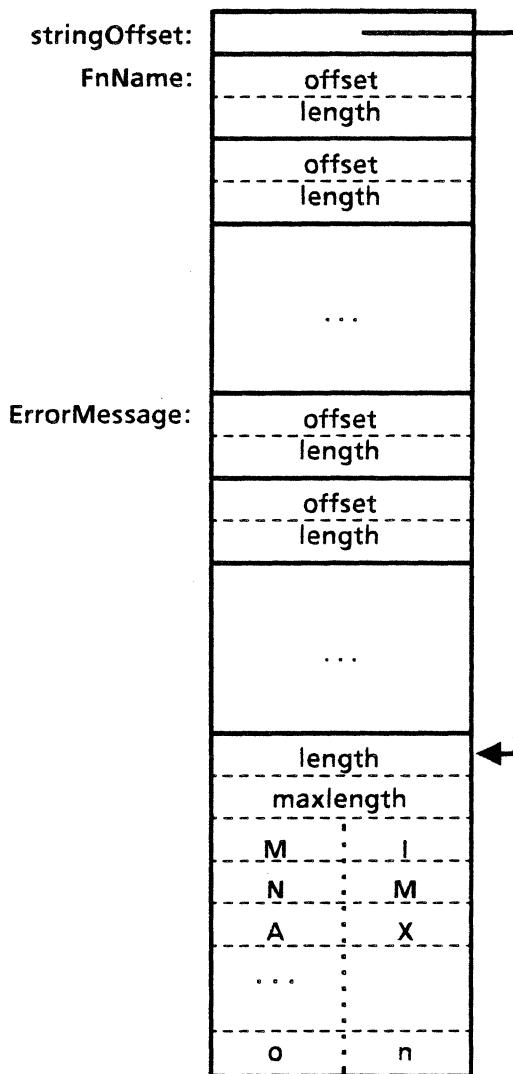
You can now pass @ss to any output routine that takes a **String.SubString**.

C.5 File format

The output of the StringCompactor has a "bcd" header that describes a single module. The "code" for this module is in the format illustrated here.

- The first word is a self-relative pointer to the **StringBody** where all of the string characters have been placed.
- Next comes one or more arrays of **offset**, **length** pairs (**CompStrDesc**) that a client program can use to generate a **SubString** that describes the literal.
- Finally, the file contains a **StringBody**, whose text contains the characters from the entire collection of literals.

The format file output by the TableCompiler contains a declaration (**CompStrRecord**) that describes the beginning of the data. The standard mode of operation is to copy the contents of this file into either the program using the strings or into a definitions file, if several programs are using them. Since the format changes only when the length of the arrays change, you usually ignore the format file when you have simply edited one or more of the string literals. In the case of the example, things are defined in terms of named constants to the extent that the format almost never changes.



C.6 Options

Like the ModuleMaker, the StringCompactor can export its **PROGRAM** either as **SELF** or to a named interface. See section 4 for details.

The StringCompactor has another, little used mode where it doesn't actually "compact" the strings. In this mode, the output file (data portion) consists of one or more arrays of relative pointers to **StringBody**, followed by the **StringBodys**. The output in this format can be used to obtain a file of string literals where the client program can produce a (**LONG**) **STRING** (i.e., (**LONG**) **POINTER TO StringBody**) for each of the literals, instead of having to deal with **SubStrings**. The disadvantage of this format is that there is an extra word of overhead (**maxlength**) associated with each literal.

The input files to the StringCompactor are valid Mesa programs, and can in fact be compiled (unless they overflow the compiler's string literal table). In fact, it is a good idea

to compile them occasionally—the StringCompactor doesn't actually check the number of literals against the declared length of the arrays.

C.7 Command line syntax and switches

The TableCompiler runs in the executive window; when loaded, it registers a command **TableCompiler.**~. It reads a series of identifiers with optional switches.

A single command to the TableCompiler consists of an input file name, with optional switches, possibly followed by auxiliary file names with mandatory switches. The end of the command is denoted either by the end of the line, or by the presence of a /g switch on the last file name of the command.

If the file extension of the input file is ".mesa" (or omitted), the StringCompactor will be run; otherwise the ModuleMaker will be run. This decision can be overridden by switches on the input file name (**m**, **s**, or **t**).

The name of the program exported by the output file is the root name of the input file (exactly as capitalized on the command line). It will be exported to **SELF** unless there is an interface file specified (with a /i switch) in the command. If you export to an interface, the input file must be named the same as the **PROGRAM** declaration in the interface.

The TableCompiler used to generate object files for the Alto world as well. The only observable difference between /a and /-a in this version is whether then declarations in the format file output of the StringCompactor are **LONG** or not. The next version of the TableCompiler will have this feature removed.

C.8 Examples

To run the StringCompactor on Foo.mesa, to make **Foo.bcd** exporting **SELF**:

```
>TableCompiler.~ Foo/-a
```

To run the ModuleMaker on Foo.binary, to make **Foo.bcd** exporting **Foo** to **FooDefs**:

```
>TableCompiler.~ Foo/-a FooDefs/i
```

To run the ModuleMaker on Foo.binary, to make **Foo.bcd** exporting **SELF**:

```
>TableCompiler.~ Foo.binary/m
```

To run the ModuleMaker on Foo.binary, to make **Foo.bcd** exporting **Foo** to **FooDefs**:

```
>TableCompiler.~ Foo.binary/m FooDefs/i
```

To run the StringCompactor on Foo.mesa, to make **Foo.bcd** exporting **SELF**, and then run MakeModule on Baz.binary, exporting **SELF** to **Baz.bcd**:

```
>TableCompiler.~ Foo/-ag Baz.binary/m
```

C.9 Switches on the input file name

Switches are optional on the input file name—the program looks at the input file name extension and chooses (/t if the extension is ".mesa", /m otherwise).

<i>switch</i>	<i>default</i>	<i>meaning</i>
a	TRUE	Alto output—affects the declarations in the format file. This switch should go away, and should be specified as FALSE if you are planning to use the format file output.
c	TRUE	Compact strings—puts the StringCompactor into compacting mode; you should probably use the /t switch instead.
g		Go—there are not auxiliary file names in this command.
m		Run the ModuleMake regardless of the decision based on file extension.
s		Run the StringCompactor (in non-compacting mode) regardless of the decision based on file extension.
t		Run the StringCompactor (in compacting mode) regardless of the decision based on file extension.

C.10 Switches on auxiliary file names

Every auxiliary file name must have at least one switch. The last file name in the command also has a /g switch (unless it is the last thing on the command line). For the purposes of discussion, let *root* be the root of the input file name.

<i>switch</i>	<i>meaning</i>
f	Format file—tells the StringCompactor where to write the format declarations. The default is <i>rootFormat</i> .
g	Go—there are not auxiliary file names in this command.
i	Interface—export to this interface. It must contain a declaration <i>root: PROGRAM</i> .
o	Output file—you can change the name of the output file, but bear in mind that the PROGRAM it exports will still be named <i>root</i> .



Parser Generator System

The parser generator system (PGS) is a tool that translates a LALR(1) grammar into a parse table. More specifically, it analyzes a context-free grammar specified in Backus-Naur form as input to see whether it is LALR(1), and if so, outputs compacted binary tables that can be used in conjunction with the Mesa parser. It also produces ancillary tables that simplify writing lexical routines to recognize the terminal symbols of the grammar. Since one of the main uses of the PGS is in building the Mesa compiler, there is a preprocessor that aids this (see section D.6).

The LALR(1) parsing algorithm has good space and time performance, handles a larger subset of the context-free grammars than other methods in common use, and allows a good syntax error repair capability to be added. Since the LALR(1) condition can be less than intuitively obvious, however, checking that the condition holds requires substantial computation; if the grammar is not, a fair knowledge of the underlying theory may be needed to change the grammar to make it meet the condition.

The most accessible description of LALR(1) parsing is the tutorial article "LR Parsing" by Aho and Johnson in *Computing Surveys*, 6 (1974) 74. The most comprehensive account readily available is in *The Theory of Parsing, Translating and Compiling* Volume 1, Prentice-Hall (1972), by Aho and Ullman. The algorithms used in the PGS are from Anderson, Eve, and Horning, "Efficient LR(1) Parsers," *Acta Informatica* 2 (1973) 12, so the terminology here follows this paper, referred to below as AEH.

D.1 Using the Parser Generator

After **PGS.bcd** has been retrieved, issuing the command **pgs** in the Executive invokes it. The PGS prompts for an input file name. The input file name implicitly defines the names of the various output files. The main part of the input file name is extended by **.echo**, **.log**, **.binary** and **.module** to form the names of the primary output files. However, input files with extension **.mesa** or **.Mesa** are an exception, as discussed in section D.5. Sections D.3 and D.4 discuss the PGS's input and output. Installing the system and assembling it from its components are described in section D.7. An example input file and the resulting **.module**, **.echo** and **.log** files, are given in section D.8.

D.2 Format of the input file

The input file is treated as a sequence of tokens, where a token is defined to be a sequence of characters none of which are in the range [0C..']. Tokens are delimited by sequences of characters in this range. In addition to the tokens ::=, |, ?, C, and the integers that have special roles, there are a number of tokens starting with the character pair || that control the PGS.

The input consists of five subsequences: directives, terminals, nonterminals, aliases, and productions, which must appear in that order.

1. The principal directives and their functions are:

||INPUT - causes the input to be echoed to the .echo file.

||CHAIN - causes an optimization to be performed that speeds up parsing by eliminating all references to productions marked as chain productions

||LISTS - causes the LALR(1) tables to be compacted and output to the .binary and .module files

||PRINTLALR - causes a readable form of the LALR(1) tables to be output to the .log file. (A table for a grammar of about 300 productions contains roughly 400,000 characters. Generating this readable form noticeably slows the PGS.)

2. ||TABLE1 introduces tokens representing the terminal symbols of the grammar. The last token denotes a unique sentence endmarker used only to delimit sentences supplied to the resulting parser; this token should not appear in any production. (The scanner invoked by the parser using the tables generated by the PGS is required to map the end-of-input signal onto this token.)
3. ||TABLE2 similarly introduces the nonterminal symbols of the grammar. The nonterminal symbol appearing first after ||TABLE2 is taken to be the goal symbol of the grammar. The way that the Mesa parsing algorithm terminates entails a weak constraint that neither should the goal symbol appear in the right part of any production nor should any of its productions be designated chain productions.
4. The optional alias sequence, if included, is introduced by ||TABLE3. The terminal symbols of a grammar do not necessarily have the form of identifiers, but lexical or error recovery routines may need to reference them. The alias sequence consists of pairs of tokens, the first of which is a terminal symbol (that is, it appears in the sequence following ||TABLE1), and the second is an alias in the form of an identifier by which it can be referenced. Appropriate constant definitions are constructed for these identifiers and included in the .module file.
5. Finally, the productions are listed in Backus-Naur notation following ||TABLE4. The tokens ::= and | play their usual role; there is no symbol terminating or separating productions. Likewise a production deriving the empty string is specified by the absence of any token after ::= or | and the succeeding |, end of input, or token followed by ::= sequence.

Immediately preceding the |'s or to the left of a token followed by ::= there will usually be an integer, the rule number, which may itself be preceded by the token C (upper case

only). The rule number associates the production with a semantic routine (an arm of a **SELECT** statement) to be invoked by the parsing algorithm whenever a string derived from the associated production is recognized. Some productions, of the form nonterminal ::= nonterminal, have no semantic significance and serve merely to assert which members of one syntactic class are also members of some larger class. Chains of such productions appear in expression grammars (such as, **expr** ::= **term**, **term** ::= **factor**, **factor** ::= **primary**) and can significantly reduce the speed of parsing. The appearance of the token **C** is an assertion that the following production is of the specified form and has no semantic processing associated; this allows the PGS to eliminate all references to it with an increase in speed of parsing.

The input phase of the PGS uses the Mesa parsing routine and parsing tables of its own construction so there is clearly a grammar specifying the form of the input. The description given above was preferred as an introduction to the input format since it covers only the essentials. The PGS will, on request, echo its input interspersed with other information in a formatted form. As there was a requirement that this output should also be acceptable as input, the grammar allows a rather wider class of input forms but information other than that described is simply discarded during re-input. The grammar appears in the appendix.

It should be clear that the terminal symbols of the PGS grammar cannot be used as tokens in a client grammar; a syntax error would result. This is not likely to be a problem to most clients insofar as the symbols starting with || are concerned (that is why this curious system was adopted) but there are also ::=, |, ?, **C** and **GOAL** some of which may well appear in a client grammar. Finally of course the PGS grammar contains all of them. The standard solution is used; if |, ?, or **C** are required as tokens in a client grammar they must be specified as '|', '? and 'C'. Multi-character symbols such as ::= must be specified as "::=". The only special treatment that these quoted symbols receive at the hands of the PGS is in building the tables for use by the scanner; any two character token beginning with a single quote has the quote removed; similarly any token of three or more symbols where the first and last are double quotes has these bracketing quotes stripped.

Occasionally, it is convenient to be able to flag certain productions in the input text. The PGS will ignore ? when looking for **C** or a rule number; the ? should precede **C** when a rule number is also present.

D.3 Output of the Parser Generator

Four output files are normally constructed: a record of the input, a log, a binary output file containing the parser and lexical tables, and a module file that contains definitions of aliased terminal symbols and some ranges defining the sizes of the various arrays constituting the binary file. The module file is a Mesa module named **ParseTable**. It must be compiled and bound with the Mesa parser, a suitably modified version of the Mesa scanner, and the definitions module that describes the invariant parts of the binary parse tables.

Examples of these latter modules exist in the PGS. The files **pgsptabdefs.mesa**, **pgsscan.mesa**, **pgsparse.mesa** and **pgs1.mesa** contain respectively ParseTable, the scanner and semantic processing routines for the PGS, the Mesa parser and, finally, definitions of the invariant part of the binary tables. For operational reasons, the low-level routines interfacing with I/O, storage management, etc. have been removed from **pgsparse.mesa** and **pgsscan.mesa** to the control module of the PGS in the file

pgscon.mesa. Nonetheless these modules should provide a model for anyone needing it. In particular, the code for loading and unloading the binary tables and invoking the parser can be found in the main line code of the module **PGScon**.

D.3.1 The input record file

This file is produced if the directives in the input stream contain **||INPUT**. The name of the file is obtained by appending .echo to the main part of the input file name.

The record of the input differs from the true input in that the directives may be displayed in a different order and the terminal and nonterminal symbols are displayed one to a line each preceded by an integer. (The integers are allocated sequentially starting at one for the first terminal symbol. Each production is displayed starting on a new line; each is preceded by two integers, the second being the rule number from the input. If a **C** was specified on the input it appears between the two integers. the first integer is simply a unique label for the production. The first production is labelled one and again the PGS simply labels the productions with ascending integers. These labels are used in some of the diagnostic messages output by the PGS. A production with the implicit label zero is constructed and output before the others, it has the form,

GOAL ::= goal eof

where **goal** stands for the goal symbol and **eof** for the end of sentence marker. A check during input ensures that these symbols occur to the right of **::=** in no other production.

D.3.2 The Log file

This file contains error messages and various items of supplementary information output during the generation of the parsing tables. If error or warning messages are logged then, immediately prior to the end of execution, the message, "**Errors or warnings logged**" is displayed followed by the usual invitation to type any character to terminate processing.

D.3.2.1 Error messages

Most error messages occur during input of the grammar. Those messages prefixed by the word **ERROR** cause the program to terminate after completing input and checking for further errors. **WARNING** messages allow processing to continue. Each message is accompanied by a fragment of input text with a pointer to the current input token.

The warning messages are:

1. Overlength symbol (increase **TOKENSIZE?**) truncated to -
2. Not a chain production -
3. Unused(**U**) or undefined(**D**) symbols (refer to **TABLE1** and **2**)

These messages illustrate some general points; messages ending with a dash are followed by further information. For message 1, it is the truncated form of the token, for message 2

it is the integer label appended to the offending production in the echoed input. After message 2, processing continues as though no chain indication had been specified.

Messages such as the first that indicate that an internal field size is too small also specify the compile time constant (in `pgscondefs.mesa`) that controls the field size. Currently tokensize constrains tokens to 14 characters.

The third message occurs at the end of input if there are symbols in the **TABLE1** and **2** sequences that do not appear in any production (unused symbols) or if a symbol in the **TABLE2** sequence does not appear to the left of `::=` in the productions (undefined symbols). This message is followed by a list of integers that designate symbols in **TABLE1** and **2** using the numeric labels appended in the echoed input. Each integer is tagged with **U** and/or **D** to indicate whether the corresponding symbol is unused or undefined or both.

The **ERROR** messages are:

4. Nonterminal with too many rules (increase **ALTERNATELIM?**) -

Only 31 productions are allowed for any nonterminal; if this is not enough it would be better to introduce a new nonterminal and split them into two groups rather than increase the limit of 31.

5. Multiple definitions of symbol -

This message occurs either because the same symbol appears more than once in **TABLE1** and **2** or because the same symbol occurs to the left of `::=` more than once. The offending symbol is printed after the message.

6. Symbol not defined -

This message also appears in two contexts, either a symbol appears in a production that does not appear in **TABLE1** or **2** or alternatively message 3 was issued with undefined symbols. in the first case the message is followed by the symbol in question, in the second by "see previous message".

7. Terminal precedes `::=` -

The terminal symbol follows the message.

8. Too many rhs symbols in production (increase **RHSLIM?**) -

Fifteen symbols in the right part of a production is unlikely to be exceeded; if it is, change the grammar, increasing rhslim would involve consequential changes in the binary table formats.

9. Internal field will overflow - increase **PSSLIM**

This one is unlikely, it would involve a grammar with 1024 symbols or productions. An increase up to 2047 would be possible without changing the binary table formats.

10. Aliased symbol not a terminal symbol -
The symbol follows the message.
11. Aliases must not be terminal symbols -
The symbol follows the message.
12. Goal symbols used in rhs
The goal symbol or end of sentence marker appear in a production right part.
(See the paragraphs numbered 1 and 2 in section D.3).
13. Number greater than MAXRULE
Currently the PGS allows for 255 rule numbers. A relatively minor reformatting of the binary tables would permit it to be increased to LAST[CARDINAL].

D.3.2.2 Output during table construction

During the construction of the parsing tables there is a reasonably remote chance that error message 9 will occur and terminate any further processing, though in this case it implies that more than 1023 parsing states are necessary for the grammar. Much more likely are messages indicating that the grammar is not LALR(1).

In the event of conflicting parsing actions arising for some terminal symbol in a parsing state, all data appertaining to that state is listed. The first heading line specifies,

1. an integer naming the state,
2. a symbol of the grammar (recognition of this symbol causes this state to be entered),
3. a set of p,j pairs defining the state (see AEH), each pair being followed by /.

Below the heading in a four column format is the list of symbols of the grammar that may be encountered in this state. Each symbol is preceded by an encoding of its associated parsing action:

1. unsigned integers denote scan (or shift) entries to the state named by the integer,
2. integers preceded by an asterisk signify reduce operations using the production with the integer as its label,
3. an integer preceded by a minus sign also indicates a production number but implies that the symbol associated with this action must be stacked before the reduce operation; a so-called scanreduce operation. (These marking conventions differ from those used in AEH though they use the same symbols.)

During construction of the tables scan entries are computed before reduce entries, so when conflicting actions arise they are reduce actions that either conflict with an existing scan entry or with a previously computed reduce entry. (It is an inherent property of scanreduce entries that they cannot conflict with another entry.) Conflicts are indicated by lines of the form,

Reduce with <i>n</i> conflicts with	*****
-------------------------------------	-------

where *n* is a production number. Each such line is followed by a list of items of the form, symbol action /, where action is either **SCAN** if a scan action for this (terminal) symbol has already been constructed or is an integer naming the production for that a reduce action has already been constructed.

If the directive **||PRINTLALR** is specified, the output just described occurs for every state whether or not it contains conflicts. The heading, LALR(1) Tables, precedes such output. This output is rarely worth having, it is occasionally of value in tracking down a conflict. Its primary function was in debugging the PGS.

The PGS discards conflicting entries after printing them and it will not form either the binary tables or the module output if there are reduce-reduce conflicts. In the case of reduce-scan conflicts the decision to process scan entries first implies that the scan entry takes precedence. This is occasionally useful. For example it solves the dangling else problem in the preferred way. However, the scan-reduce conflict message is a warning and as such triggers the displayed message directing attention to the .log file.

After generating the tables a heading, *LALR(1) Statistics*, is output and a few counts are printed. Only the first three may be of any general interest, they indicate the number of parsing states and the total number of actions in the tables for both terminal and nonterminal symbols.

D.3.2.3 Output during table compaction

Table compaction and output of the binary tables only occur if the directive **||LISTS** is specified in that case the output described here appears on the .log file.

The earlier stages of the PGS are written in a general way, data structures will expand to accomodate very large grammars; at the cost of recompiling the system and changing compile time constants, the limits on field sizes mentioned previously can be increased. At this stage the objective is to pack information economically into 16-bit words and it is here that the size of fields is an absolute constraint. Final checks are made that could conceivably produce one or more of the self explanatory messages:

FAIL - more than 255 terminal symbols

FAIL - more than 254 nonterminal symbols

FAIL - more than 2047 states or productions

FAIL - more than 15 symbols in a production right part

These are rather unlikely, the tightest constraint is likely to be the limit of 255 on rule numbers. If any of these checks fail, processing terminates.

Assuming no error messages, the only unsolicited output here is a heading, *Parse Table Data*, and one table. A hash-accessed look-up table, for the terminal symbols of the grammar, is created for use by the scanner. As hash functions are notoriously unreliable, the following is printed so that a visual check can be done to avoid problems. The subheading,

Scanner hashtable probe counts (terminal symbol, probe count, hashcode)

followed by a four column layout of triples that, as the heading indicates, show for each symbol the number of probes needed to locate it in the hash table and its hashcode. The technique used is Algorithm C, page 514, Volume 3 of Knuth's *The Art of Computer Programming*, Addison-Wesley (1973). If there are n terminal symbols, they are hashed into a table of size $\text{MIN}[m, 251]$ buckets; m is always an odd integer, either $3*n/2$ or $3*n/2 + 1$. The hash function uses the ASCII values of the first and last character of the symbol and is

$$((127 * \text{first character} + \text{last character}) \bmod \text{buckets}) + 1.$$

The performance of this hash function deteriorates as the number of terminal symbols approaches 255.

If both the directives `||PRINTLALR` and `||LISTS` are specified, a record of the table compaction transformations is produced. This record is typically of interest only for maintaining a system, and familiarity with the compaction techniques described in AEH is assumed in its description.

First, a set of default actions for the nonterminal symbols of the grammar are determined, and a table headed *Nonterminal Default Actions* is printed. Each nonterminal symbol appears preceded by its associated default action encoded in the form already described: unsigned integers represent scan entries, and negative integers represent scan-reduce actions. (Reduce actions never take place on nonterminal symbols.)

After removing all occurrences of these defaulted entries from the LALR(1) tables, the PGS determines those states that have identical symbol-action pairs, first, over the set of terminal symbols and then, independently, over the set of nonterminal symbols. All states reference one copy of the list of symbol-action pairs stored in the binary tables. The table *Table Overlays* has three columns headed *row*, *ttab* and *ntab*; if a row of this table contains (integer) entries a , b and c respectively, then it implies that the terminal entries of state a are the same as those of state b , while the nonterminal entries of state a are the same as those of state c . It is exceptional for both the terminal and nonterminal entries of a state to match those of other states so usually one of the entries b or c is blank. If neither the terminal nor nonterminal entries of a state match those of another state, then it does not appear in this table.

The final transformation is to renumber the states so that all of those states containing (nondefaulted) actions on nonterminal symbols are labelled by integers contiguous to each other and to 1. This is achieved by swapping the highest numbered state with nonterminal actions with the lowest numbered state without nonterminal actions until no more swaps are possible. The table headed, *Renumbered States*, simply records this with entries of the form, a swapped with b .

D.4 The module file

The module file is most readily described with an example. Consider the module file generated by the PGS for its own grammar.

```

ParseTable: DEFINITIONS = [
    Symbol: TYPE = [0..255];
    TSymbol: TYPE = Symbol[0.. 19];
    NTSymbol: TYPE = Symbol[0.. 13];

    -- token indices for the scanner and parser
    tokenID: TSymbol = 1;
    tokenNUM: TSymbol = 2;
    tokenQUERY: TSymbol = 3;
    tokenTAB3: TSymbol = 9;
    tokenTAB4: TSymbol = 10;
    initialSymbol: TSymbol = 3;

    defaultMarker: TSymbol = TSymbol.FIRST;
    endMarker: TSymbol = TSymbol.LAST;

    HashIndex: TYPE = [0.. 29];
    VIndex: TYPE = [0..106];

    State: TYPE = [0.. 26];
    NTState: TYPE = State[0.. 6];
    TIndex: TYPE = [0.. 64];
    NTIndex: TYPE = [0.. 3];
    Production: TYPE = [0.. 37];
];

```

The module defines aliases in the aliases segment of the input. For example, the token **[TABLE3**, that is a terminal symbol of the PGS grammar was given the alias **tokenTAB3**. It is the ninth token in the sequence of terminal symbols in the input file and so internally is encoded within both the PGS and the binary tables as 9.

The ranges **HashIndex**, **TSymbol**, **NTSymbol**, **State**, **NTState**, **TIndex**, **NTIndex**, **Production** and **VIndex** prescribe the dimensions of arrays in the binary tables.

D.5 The binary file

D.5.1 Binary file format

The format of the binary file is captured by a set of Mesa definitions that, since they are of interest to both scanner and parser, can conveniently be specified in the definitions module that constitutes the scanner-parser interface. These definitions are reproduced below:

```

ActionTag: TYPE = MACHINE DEPENDENT RECORD [
    reduce(0: 0..0): BOOL,           -- TRUE if reduce entry
    pLength(0: 1..4): [0..15]];      -- number of symbols in production rhs

```

```

ActionEntry: TYPE = MACHINE DEPENDENT RECORD [
  tag(0: 0..4): ActionTag,           -- [FALSE,0] if a shift entry
  transition(0: 5..15): [0..2047]]; -- production number / next state

ProductionInfo: TYPE = MACHINE DEPENDENT RECORD [
  rule(0: 0..7): [0..256],          -- reduction rule
  lhs(0: 8..15): NTsymbol];        -- production lhs symbol

VocabHashEntry: TYPE = MACHINE DEPENDENT RECORD [
  symbol(0: 0..7): TSymbol,         -- symbol index
  link(0: 8..15): HashIndex];      -- link to next entry

ScanTable: TYPE = ARRAY CHAR['\040..\177] OF TSymbol;
HashTable: TYPE = ARRAY HashIndex OF VocabHashEntry;
IndexTable: TYPE = ARRAY TSymbol OF CARDINAL;
Vocabulary: TYPE = MACHINE DEPENDENT RECORD [ -- a string body
  length(0), maxlen(1): CARDINAL,
  text(2): PACKED ARRAY VIndex OF CHAR];
ProdData: TYPE = ARRAY Production OF ProductionInfo;
NStarts: TYPE = ARRAY NTState OF NTIndex;
NLengths: TYPE = ARRAY NTState OF CARDINAL;
NSymbols: TYPE = ARRAY NTIndex OF NTsymbol;
NActions: TYPE = ARRAY NTIndex OF ActionEntry;
NTDefaults: TYPE = ARRAY NTsymbol OF ActionEntry;
TStarts: TYPE = ARRAY State OF TIndex;
TLengths: TYPE = ARRAY State OF CARDINAL;
TSymbols: TYPE = ARRAY TIndex OF TSymbol;
TActions: TYPE = ARRAY TIndex OF ActionEntry;

initialState: State = 1;
finalState: State = 0;

Table: TYPE = MACHINE DEPENDENT RECORD [
  scanTable: RECORD [
    scanTab: TableRef RELATIVE POINTER TO ScanTable,
    hashTab: TableRef RELATIVE POINTER TO HashTable,
    vocabIndex: TableRef RELATIVE POINTER TO IndexTable,
    vocabBody: TableRef RELATIVE POINTER TO Vocabulary
  ],
  parseTable: RECORD [
    prodData: TableRef RELATIVE POINTER TO ProdData,
    nStart: TableRef RELATIVE POINTER TO NStarts,
    nLength: TableRef RELATIVE POINTER TO NLengths,
    nSymbol: TableRef RELATIVE POINTER TO NSymbols,
    nAction: TableRef RELATIVE POINTER TO NActions,
    ntDefaults: TableRef RELATIVE POINTER TO NTDefaults,
    tStart: TableRef RELATIVE POINTER TO TStarts,
    tLength: TableRef RELATIVE POINTER TO TLengths,
    tSymbol: TableRef RELATIVE POINTER TO TSymbols,
    tAction: TableRef RELATIVE POINTER TO TActions
  ]
]

```

l;

TableRef: TYPE = LONG BASE POINTER TO Table;

The purpose and content of the arrays in **parseTable** are explained in AEH; only the definitions relevant to the scanner are discussed here. Terminal symbols of the grammar represented by a single ASCII character are treated separately from those requiring a string of characters. In **scanTable** there is an array **scanTab**, that can be indexed by characters not in the range [0C..']'; any single character symbol used to extract an element of this array will extract a non zero integer only if it represents a terminal symbol and the integer is its numeric encoding.

The string **vocabBody** contains the character strings representing all other valid terminals stored head to tail. Element i-1 of **vocabIndex** indexes the first character of the string in **vocabBody** that represents the terminal symbol with the encoding i. The hash value of a string that purports to be a terminal symbol can be computed using the hash function given in section D.3.2.3 (identifying buckets there with **LAST[HashIndex]**). The hash value can be used to select an element from the array **hashTab**; the elements of **hashTab** are records, one field of that is used to select an entry of **vocabIndex** (to find the substring in **vocabBody** to compare with the given string), the other field is an index to another element in **hashTab** to be tried when the string comparison fails; **hashTab[0]** is void, a zero index terminates the search indicating that the given string does not represent a terminal.

D.5.1 The LR and first files

As part of the debugging facilities built into the PGS, two other output files can be created.

The first step in building the LALR(1) tables is to construct LR(0) tables. (This is done using the SLR(1) algorithm in section D.3.1 of the AEH paper by omitting the computation of the sets specified in relation (5b).) The LR(0) tables may be output to a file with the extension .lr by specifying the directive ||PRINTLR in the input. The form of the output is similar to that used in the LALR(1) tables, except that, of course, the terminal symbols triggering reduce actions are not known.

Reduce with p

follows the state heading if the production with label p has reduce actions in the state.

The next stage is to compute lookahead symbols for all these incompletely determined reduce actions according to the LALR(1) rules. This is done using Anderson's bewilderingly succinctly stated algorithm on pages 21 and 22 in AEH. It is convenient, as a preliminary to this, to compute, for each nonterminal symbol, the set of terminal symbols that can appear as the first symbol in a string derived from the nonterminal. This transitive closure calculation provides the initial data for computing Anderson's exact right contexts, which is in turn, a transitive closure calculation.

If the directive ||FIRST appears among the input directives in addition to either of the directives ||PRINTLALR or ||LISTS, a file with extension name .first is created that contains a list of all nonterminal symbols each being followed by an (offset) list of the terminals that can start a string derived from it.

D.6 The Preprocessor

The preprocessor is invoked if the input file name has the extension `.mesa`. Each arm of the `SELECT` statement implementing the semantic processing routines in the Mesa compiler has comments displaying those productions associated with this arm. The test preceding the `=>` symbol in the arm is the rule number for these productions. As Mesa evolves, changes are made to the grammar, productions associated with one arm must be moved to another, new productions and new arms are introduced, and periodically the rules must be renumbered in a systematic fashion to find things (there being over 200 arms). The bookkeeping necessary to ensure that the story told by the `SELECT` statement is consistent with that told to the PGS in the input file is tiresome and error prone. Most of the data needed by the PGS is present in the `SELECT` statement and by adding the rest only one copy of the grammar need be maintained and the reassignment of rule numbers can be mechanised.

The preprocessor expects a Mesa program module as input and it scans for the sequence

```
digits = > --
```

after that it expects to find data relevant to it. On encountering a carriage return, it checks whether the next printing characters open a Mesa comment or not; if they do then more data is expected otherwise the end of the data associated with a particular select arm is presumed and a search is instituted for the next.

Supposing the input file name given was `semroutine.mesa`, then during the scan the preprocessor copies the input file to `semroutine.mesa$` and makes a modified copy in a scratch file. The change is a trivial one; as the sequences,

```
digits = > --
```

are encountered, the next non-negative integer (starting with 0) is substituted before the `=>` symbol. At the end of the input scan, the scratch file is written back to `semroutine.mesa`.

Clearly the rather crude procedure used to locate the grammatical information in the program text places constraints on the program module containing it. In particular it precludes comments in a fairly natural place in any other `SELECT` statements that may appear in the module that also use integer tests. On the other hand anything approaching parsing the text is out since it merely replaces one updating problem with another.

Since the PGS constructs tables using the new rule numbers just assigned, arms of the select statement can be reordered to group logically related arms together without making the search for an arm with a given rule number tedious.

In the comments associated with arms `SELECT` (other than the first encountered), the preprocessor expects to find only productions. Here each production is specified in full, its left part token followed by either `::=` or (to designate a chain production) `::= C` followed by the right part tokens.

Comments preceding the first production contain the additional information needed. This information in order of occurrence is,

1. Optionally, and in either order, the tokens **MODULE:** or **BINARY:** may appear each followed only by a token naming the file to contain the corresponding output.
2. Next must appear **GOAL:** followed by the token naming the nonterminal that is the goal symbol of the grammar.
3. Optionally there may appear **TERMINALS:** followed by the tokens representing the terminals. The end of sentence marker should not be included, eof is supplied.
4. Optionally there may appear **ALIASES:** followed by the alias sequence as described in section D.3.
5. Finally **PRODUCTIONS:** must appear before the first production.

When the preprocessor is selected, the output file names are formed by appending the various extensions to pgs rather than the main part of the input file name thus pgs.module and pgs.binary are the default names if the **MODULE:** and **BINARY:** options are not used in the input.

Nonterminal symbols are deduced from the tokens to the left of ::= tokens. If the **TERMINALS:** sequence appears only these symbols are taken to be terminals. In its absence any token in a production that is not a nonterminal according to the previous definition is a terminal. Omitting this sequence means that typing errors define terminal symbols!

From a file of this form (see the example at the end of the appendix), the preprocessor constructs a scratch file in the format specified in section D.3, with the directives ||**INPUT**, ||**CHAIN** and ||**LISTS**, which it passes to the input phase of the PGS.

The preprocessor only generates one error message, "Directives incorrect or out of sequence". No further processing occurs in this situation so the message is displayed, followed by the request to type a character to terminate execution.

When inserting new arms in the **SELECT** statement there is no need (so far as the preprocessor is concerned) to use an integer distinct from those on other arms but it is probably not a good idea. The preprocessor will recognize ? as an alternative to a digit sequence.

D.7 Operation

D.7.1 PGS operation

PGS command line parsing has been revised to handle module identifiers and file names, in the currently approved way and to allow easier parameterization with respect to long or short pointers. The basic idea was to make PGS more like the compiler and binder in these areas (to avoid the current file name hassles and to make PGS more usable by the system modeller).

D.7.1.1 Processing modes

Input: Mesa vs. grammar. PGS can extract the information needed to build parsing tables from comments embedded within standard Mesa source files. Although the above documents this input mode almost as an afterthought, it has become the standard one. The conventional name for the input file in this mode is `<name>.pgs`. The grammar mode has been retained for its occasional utility in experimenting with grammars. See the Appendix.

Output: BCD vs. binary. The usual output mode is BCD; this facilitates packaging the parsing tables with the code that uses them (in a BCD, boot or image file). The binary mode has been retained primarily for situations in that the parsing tables are to be used by non-Mesa programs.

D.7.1.2 Mesa programs as PGS source files

The list of keywords that optionally precede the first production has been revised and expanded as follows:

TABLE:
TYPE:
EXPORTS:
GOAL:
TERMINALS:
ALIASES:
PRODUCTIONS:

The first three must precede all the others but may occur in any order; the next section explains their significance. The last four must appear in the specified order; all but the last may be omitted.

Output Identification

In the source text, the old keywords dealing with file and module names are replaced by

TABLE: *tableId* **TYPE:** *typeId* **EXPORTS:** *interfaceId* -- or **EXPORTS:** SELF

This is supposed to remind you of

programId: <ProgramType> **EXPORTS** *interfaceId*

(sorry about the different treatment of colons, etc.). The names **tableId**, **typeId** and **interfaceId** are module identifiers (capitalization counts) and get put into BCDs and symbol tables.

Examples

The following examples are taken from the current system. The compiler and binder specify the same type name because they use the same parsing module, in which that interface is a (compile-time) parameter. On the other hand, they export different interfaces because loading of the corresponding tables is handled differently (see below).

TABLE: **BCDParseData** **TYPE:** **ParseTable** **EXPORTS:** SELF

-- *binder*

TABLE: MesaTab TYPE: ParseTable EXPORTS: CBinary	-- compiler
TABLE: PGSParseData TYPE: PGSParseTable EXPORTS: SELF	-- PGS

D.7.1.3 Invoking PGS

File Naming

When you invoke PGS, you can arbitrarily associate files and module identifiers using the same command-line conventions that the compiler and binder use. The most general form is:

```
[defs: defsFile, bcd: bcdFile, grammar: grammarFile] ←
    sourceFile[interfaceId: interfaceFile]/switches
```

that puts

```
the source for the interface typeId on defsFile.mesa,
the BCD for the tableId (or binary, if you say "binary:") on bcdFile.bcd (or
.binary),
a summary of the grammar on grammarFile.grammar (only if input was a Mesa
source file)
```

and finds the BCD for **interfaceId** on **interfaceFile.bcd**. Capitalization is ignored. By default

```
defsFile: typeId.mesa
bcdFile: tableId.mesa
grammarFile: tableId.grammar (inhibit with /-g)
error messages: tableId.pgslog
```

(There are further defaults for the cases in which the input is just a grammar file or you omit keyword items for the module identifiers in the source).

Command line examples

The following commands build parsers for the Compiler, Binder, and PGS:

PGS [grammar:Mesa] ← Pass1T.pgs

```
needs CBinary.bcd
produces Pass1T.mesa, MesaTab.bcd, ParseTable.mesa
exports MesaTab as a PROGRAM in the interface CBinary
puts grammar summary in Mesa.grammar
```

PGS [defs: BcdParseTable, grammar: CMesa] ← BcdTreeBuild.pgs

```
produces BcdTreeBuild.mesa, BcdParseData.bcd,
BcdParseTable.mesa
exports BcdParseData directly (no interface)
puts grammar summary in CMesa.grammar
```

PGS [defs: PGSParseTable, grammar: PGS] ← PGSScan.pgs

produces **PGSScan.mesa, PGSParseData.bcd, PGSParseTable.mesa**
 exports **PGSParseData** directly (no interface)
 puts grammar summary in **PGS.grammar**

D.7.2 TableCompiler operation

TableCompiler command line parsing has been similarly revised to resemble that of the compiler and binder.

D.7.2.1 Processing modes

TableCompiler is a program to convert assorted inputs to **Mesa.bcd** files that can be bound into configurations and managed as code segments. If the source file name has extension **.mesa**, it extracts string literals from string array declarations and gives the skeleton of a **DEFINITIONS** file describing the resulting structure; otherwise, it just wraps a **bcd** header around a collection of bits.

D.7.2.2 Invoking TableCompiler

File Naming

When you invoke TableCompiler, you can specify an arbitrary association between files and module identifiers using the same command-line conventions that the compiler and binder do. The most general form is:

**[bcd: bcdFile, format: formatFile] ←
 sourceFile[interface: interfaceFile]/switches**

that puts

the BCD output on **bcdFile.bcd**,
 the record format on **formatFile.format** (only if input was a Mesa source file)

and finds the interface BCD on **interfaceFile.bcd**. Capitalization is ignored here. By default

```
bcdFile:  source.bcd
formatFile:  source.format
grammarFile:  tableId.grammar  (inhibit with /-g)
error messages:  TableCompiler.log
```

where **source** is the root of the **sourceFile** name. Note that the new use of keywords is not compatible with previous use.

Command line example

The following command builds components of the compiler:

TableCompiler ErrorTab[interface: CBinary] DebugTab[interface: CBinary]

needs **CBinary.bcd**
 produces **ErrorTab.format**, **ErrorTab.bcd**, **DebugTab.format**,
DebugTab.bcd
 exports **ErrorTab** and **DebugTab** as **PROGRAMS** in the interface **CBinary**

D.8 Example

An Input File

```
||INPUT ||CHAIN ||LISTS ||PRINTLALR
|TABLE1
id + () * IF THEN OR ELSE := EOF
|TABLE2
g s a i e t p b l
|TABLE3
+ tokenPLUS
|TABLE4
1 g ::= s
Cs ::= a
C | i
2 a ::= id := e
3 i ::= IF b THEN a |
Ce ::= t
4 | e + t
Ct ::= p
5 | t * p
6 p ::= ( e )
7 | id
8 b ::= b OR id
9 | id
10 l ::= =
11 | ELSE s
```

The Resulting Module File

```
ParseTable: DEFINITIONS =
BEGIN -- types for data structures used by the Mesa parser and scanner

Symbol: TYPE = [0..255];

-- token indices for the scanner and parser

tokenPLUS: TSymbol = 2;

HashIndex: TYPE = [0.. 17];
VIndex: TYPE = [0.. 22];

TSymbol: TYPE = Symbol [0.. 11];
NTSymbol: TYPE = Symbol [0.. 10];
State: TYPE = [0.. 17];
NTState: TYPE = State [0.. 5];
TIndex: TYPE = [0.. 23];
```

```

NTIndex: TYPE = [0.. 9];
Production: TYPE = [0.. 15];
END.

```

The Resulting Echo File

||INPUT ||CHAIN ||LISTS ||PRINTLALR

|TABLE1

```

1 id
2 +
3 (
4 )
5 *
6 IF
7 THEN
8 OR
9 ELSE
10 :=
11 EOF

```

|TABLE2

```

12 g
13 s
14 a
15 i
16 e
17 t
18 p
19 b
20 l

```

|TABLE3

+ tokenPLUS

|TABLE4

GOAL ::= g EOF

1	1	::= s
2 C	2 s	::= a
C	3 i	
4	2 a	::= id := e

5	3	i	::= IF b THEN a
6 C	6	e	::= t
7	4		e + t
8 C	8	t	::= p
9	5	t * p	
10	6		::= (e)
11	7		id
12	8	b	::= b OR id
13	9		id
14	10		::=
15	11		ELSE S

The Resulting Log File

LALR(1) Tables

1	0 0/		
2 id	3 IF	0 g	-1 s
-1 a	-1 i		
id	4 1/	4 :=	
3 IF	5 1/	-13 id	5 b
4 :=	4 2/		
-11 id	6 (7 e	8 t
8 p			
5 b	5 2/	12 1/	
9 THEN	10 OR		
6 (10 1/		
-11 id	6 (11 e	12 t
12 p			
7 e	4 3/	7 1/	
13 +	*4 ELSE	*4 EOF	
8 e	4 3/	7 1/	9 1/

13 +	14 *	*4 ELSE	*4 EOF
9 THEN	5 3/		
2 id	15 a		
10 OR	12 2/		
-12 id			
11 e	7 1/	10 2/	
13 +	-10)		
12 e	7 1/	9 1/	10 2/
13 +	-10)	14 *	
13 +	7 2/		
-11 id	6 (16 t	16 p
14 *	9 2/		
-11 id	6 (-9 p	
15 a	5 4/		
17 ELSE	*14 EOF	-5 i	
16 t	7 3/	9 1/	
*7 +	*7)	14 *	*7 ELSE
*7 EOF			
17 ELSE	15 1/		
2 id	3 IF	-15 s	-15 a
-15 i			

LALR(1) Statistics

States = 17

Terminal entries = 37

Nonterminal entries = 19

First OR operation count = 5

Total OR operation count = 33

Maximum number of contexts = 9

Parse Table Data

Nonterminal Default Actions

0 g	-1 s	15 a	-1 i
7 e	8 t	8 p	5 b

-5 |

Entries removed = 0

Table Overlays

row	ttab	ntab
6	4	
13	4	
14	4	
17	1	

Scanner hashtable probe counts (terminal symbol, probecount, hashcode)

id	1 6 IF	1 9 THEN	1 3 OR	1 1
ELSE	10 :=	1 16		

Renumbered States

2 swapped with	17
3 swapped with	14
4 swapped with	13
5 swapped with	6

The PGS Grammar

||CHAIN ||LISTS ||INPUT

|TABLE1

1	symbol
2	num
3	'?
4	'
5	"::="
6	'C
7	" TABLE1"
8	" TABLE2"
9	" TABLE3"
10	" TABLE4"
11	" INPUT"
12	" CHAIN"
13	" LISTS"
14	" PRINTLR"
15	" PRINTLALR"
16	" FIRST"
17	" IDS"
18	"GOAL"
19	eof

|TABLE2

20	grammar
21	head
22	ruleset

23 directives
 24 terminals
 25 nonterminals
 26 aliases
 27 directive
 28 discard
 29 rulegroup
 30 prefix
 31 goalrule

|TABLE3

symbol	tokenID
num	tokenNUM
'?	tokenQUERY
" TABLE3"	tokenTAB3
" TABLE4"	tokenTAB4
'?	InitialSymbol

|

GOAL ::= grammar eof

1	0 grammar	::= '? head ruleset
2	head	::= directives terminals nonterminals " TABLE4"
3	1 " TABLE4"	directives terminals nonterminals aliases
4	2 directives	::=
5	28	directives directive
6	3 directive	::= " INPUT" " CHAIN" " LISTS" " PRINTLR" " PRINTLALR" " FIRST" " IDS"
7	4	
8	5	
9	6	
10	7	
11	8	
12	9	
13	10 terminals	::= " TABLE1" terminals discard symbol
14	11	
15	11 nonterminals	::= nonterminals discard symbol
16	12	" TABLE2"
17	13 aliases	::= " TABLE3"
18	14	aliases symbol symbol
19	15 discard	::=
20	28	num
21	28	'?

```

22      16 rulegroup      ::= symbol "::=" 
23      17                  | prefix symbol "::=" 
24      18                  | rulegroup symbol "::=" 
25      19                  | rulegroup prefix symbol "::=" 
26      20                  | rulegroup '|' 
27      21                  | rulegroup prefix'|' 
28      22                  | rulegroup symbol 

29      23 prefix          ::= num 
30      24                  | num num 

31      24                  | '? num 
32      25                  | discard 'C 

33      26                  | discard 'C num 
34      27                  | '?' 

5 C     28 ruleset         ::= rulegroup 
36      28                  | goalrule rulegroup 

37      28 goalrule        ::= "GOAL" "::=" symbol symbol

```

An Input File For the Preprocessor

Program text has been stripped from within and around the **SELECT** statement implementing the semantic routines of the PGS to expose the grammatical information.

```

SELECT prodData[q[j].transition].rule FROM

0 => -- MODULE: pgspatabdefsnew.mesa BINARY: pgsnew.binary
    -- GOAL: grammar
    -- TERMINALS: symbol num '?' '|' "::=" 'C "|TABLE1" "|TABLE2" "|TABLE3"
    -- "|TABLE4" "|INPUT" "|CHAIN" "|LISTS" "|PRINTLR"
    -- "|PRINTLALR" "|FIRST" "|IDS" "GOAL"
    -- ALIASES: symbol tokenID num tokenNUM '? tokenQUERY
    -- "|TABLE3" tokenTAB3 "|TABLE4" tokenTAB4 '? InitialSymbol
    -- PRODUCTIONS:
    -- grammar      ::= '? head ruleset
BEGIN
END;

1 => -- head          ::= directives terminals nonterminals "|TABLE4"
    -- head          ::= directives terminals nonterminals aliases "|TABLE4"
BEGIN
END;

2 => -- directives    ::=
BEGIN
END;

```

```
3 = > -- directive    ::= "||INPUT"
BEGIN
END;

4 = > -- directive    ::= "||CHAIN"
flags[chain] ← TRUE;

5 = > -- directive    ::= "||LISTS"
flags[lists] ← TRUE;

6 = > -- directive    ::= "||PRINTLR"
flags[printlr] ← TRUE;

7 = > -- directive    ::= "||PRINTLALR"
flags[printlalr] ← TRUE;

8 = > -- directive    ::= "||FIRST"
flags[first] ← TRUE;

9 = > -- directive    ::= "||IDS"
flags[ids] ← TRUE;

10 = > -- terminals   ::= "|TABLE1"
BEGIN
END;

11 = > -- terminals   ::= terminals discard symbol
-- nonterminals    ::= nonterminals discard symbol
BEGIN
END;

12 = > -- nonterminals ::= "|TABLE2"
BEGIN
END;

13 = > -- aliases      ::= "|TABLE3"
BEGIN
END;

14 = > -- aliases      ::= aliases symbol symbol
BEGIN
END;

15 = > -- discard       ::=
|[top] ← InputLoc[]; -- keep the parser error recovery happy

16 = > -- rulegroup     ::= symbol "::="
BEGIN
END;

17 = > -- rulegroup     ::= prefix symbol "::="
|hssymbol[v[top + 1]];
```

```

18 = > -- rulegroup :: = rulegroup symbol ":: = "
  BEGIN
  END;

19 = > -- rulegroup :: = rulegroup prefix symbol ":: = "
  lhssymbol[v[top + 2]];

20 = > -- rulegroup :: = rulegroup '|'
  BEGIN
  END;

21 = > -- rulegroup :: = rulegroup prefix '|'
  prodheader[FALSE];

22 = > -- rulegroup :: = rulegroup symbol
  BEGIN
  END;

23 = > -- prefix      :: = num
  setrulechain[v[top], FALSE];

24 = > -- prefix      :: = num num
  -- prefix      :: = '? num
  setrulechain[v[top + 1], FALSE];

25 = > -- prefix      :: = discard 'C
  setrulechain[prix, TRUE];

26 = > -- prefix      :: = discard 'C num
  setrulechain[v[top + 2], TRUE];

27 = > -- prefix      :: = '? 
  setrulechain[prix, FALSE];

28 = > -- directives :: = directives directive
  -- discard      :: = num
  -- discard      :: = '? 
  -- ruleset       :: = C rulegroup
  -- ruleset       :: = goalrule rulegroup
  -- goalrule      :: = "GOAL" ":: = " symbol symbol
  NULL;
ENDCASE = > ERROR;

```

Sword Debugger

This chapter describes the Pilot-based multilingual debugger, Sword. Sword supports source-level debugging; it allows users to set breakpoints, monitor program execution, display the runtime state, and interpret Mesa statements. The debugger is intended for use by experienced programmers familiar with Mesa; the debugger may be used for Mesa, C and FORTRAN programs. To use the debugger, run **Sword.bcd**.

E.1 Events

The debugger is multi-instance, and each instance is called an *Interpreter*. An Interpreter can be created whenever one of the following abnormal *events* occurs:

1. An uncaught signal is raised.
2. An address fault occurs.
3. A write protect fault occurs.
4. A breakpoint is hit.
5. A client program invokes the debugger through an interface, such as Runtime.
6. The user invokes the debugger by typing **SHIFT-STOP**.

E.2 Styles of debugging

There are three styles of debugging: local debugging, outload debugging, and remote debugging. In local debugging, the debugger shares the same address space as the client, and is located on the same logical volume. In outload debugging, the debugger resides in a different address space than the client, and on a different logical volume. In remote debugging, the debugger and the client are different hosts on the network.

E.2.1 Local debugging

Most applications are debugged locally. However, local debugging is not feasible when the debugger depends on the application being debugged. For instance, it is not possible to local debug the operating system or the window package because the debugger depends on them. In such cases, outload debugging or remote debugging is used. An event cannot be debugged locally if:

1. The user specifies in the Options window that the event should not be handled locally.
2. Procedure calls are disallowed (usually by the operating system).
3. The event is not an uncaught signal, fault, breakpoint, or client program call.

4. The local debugger determines that it does not have resources to handle the event.
5. The user types **SHIFT-STOP** (in other words, **SHIFT-STOP** always bypasses the local debugger).

A sample local debugging session is described in section E.3.2.2.

E.2.2 Outload debugging

In outload debugging, a *world-swap* transfers control between the client and the debugger. This mechanism protects the client and the debugger from each other. A world-swap may take from 30 seconds to a few minutes, and the time is proportional to the amount of real memory in the machine.

When the client volume is booted, the debugger creates files to hold the client's core-image (**Debuggee.outload**) and its own core-image (**Debugger.outload**). The outload files are large, since they must hold copies of real memory. When the client needs to call the debugger, the operating system on the booted volume searches for an installed outload debugger to use, looking on all volumes of type one higher than the one on which the boot file resides. The three volume types are **normal**, **debugger**, and **debuggerDebugger**. For example, if the boot file is on a volume of type **normal**, Pilot searches volumes of type **debugger**. Occasionally, it is desirable to use an installed debugger other than the one that Pilot would normally choose. In these cases, use the **Set Debugger Pointers** command in Othello or the Installer, which allows you to have a client and a debugger on volumes of the same type. Sword is usually run on a volume of type **debugger**. However, Sword will not run on a CoPilot bootfile; it must be run on a Tajo bootfile.

A sample outload debugging session is described in section E.3.2.3.

E.2.3 Remote debugging

It is possible to debug clients over the Ethernet with a remote machine. A client must use a remote debugger if there is not an outload debugger available, or the client volume was booted with the "5" switch (which causes the client to always wait for a remote debugger). While waiting for a remote debugger, the client displays MP code 915, and while communicating with a remote debugger, the client displays MP code 917.

To remote debug, set the client item in an Interpreter to "remote" and give the name of the host (see the section below on the Interpreter form subwindow). A host can be the name of a machine that is registered in the clearinghouse, a net address of the form **netNumber#hostNumber#**, or a processor number of the form **netNumber.processorNumber**. If a **Domain** and **Organization** have been specified in the user profile, they will be used to qualify any partially qualified host names. Otherwise you will have to supply fully qualified host names for any remote clients you wish to debug. Before communications have been established, and whenever the debugger is waiting for the remote machine, it displays: "**Waiting for client...**". This is followed by the message "**Client responds**" when communications are re-established. To stop remote debugging, set the client item to "closeRem". Pressing the **ABORT** key while waiting for the client will also abort remote debugging.

A sample remote debugging session is described in section E.3.2.4.

E.3 User interface

Sword runs two tools, the Sword tool and the Interpreter tool. The Sword tool is used to freeze and thaw processes, and examine the load state. The Sword Tool is useful for debugging programs in infinite loops, or examining transient states of programs that are normally executing. The Interpreter tool is used to debug crashes, control program flow, and examine the runtime state of programs. There is only one Sword tool, but any number of Interpreters.

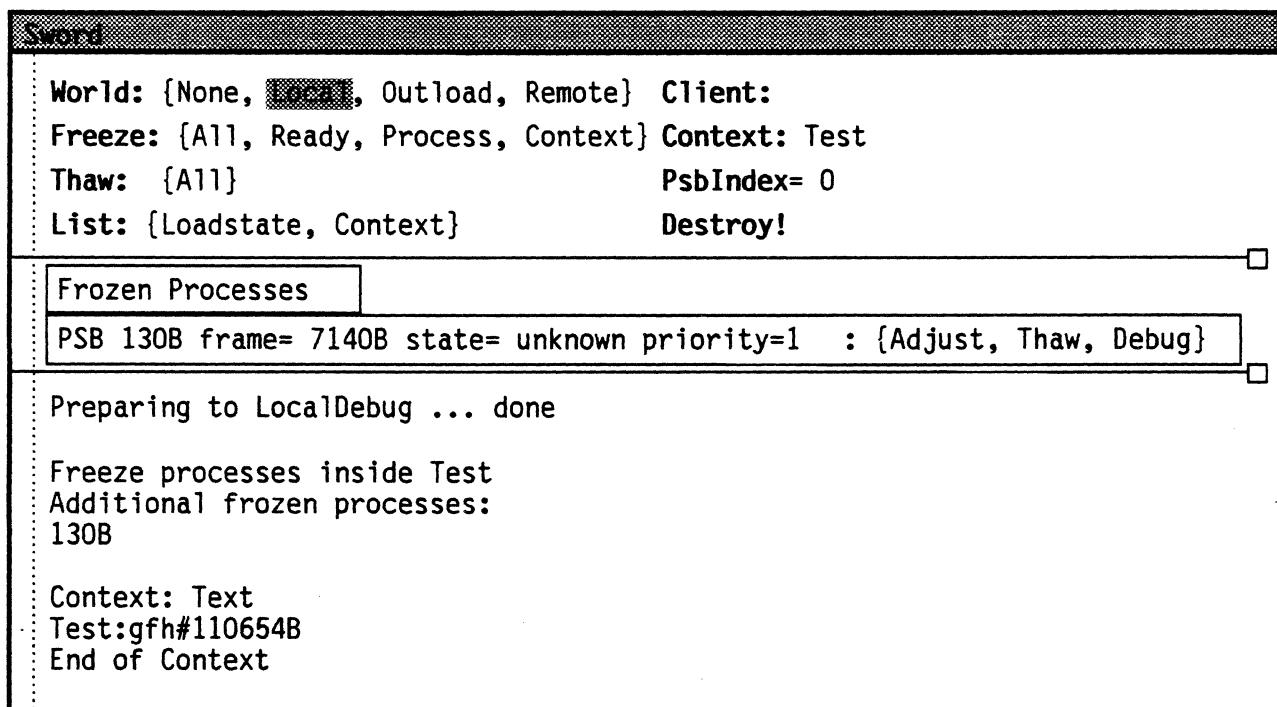


Figure E.1: Sword

E.3.1 Sword Tool

The Sword tool has a form subwindow for commands and a file subwindow for output. No commands will work until a client has been specified with the World item. For outload or remote clients, the user should enter the name of the client in the "Client" field, then bug "Outload" or "Remote" in the "World" item. For local debugging, the user can just bug "Local" in the "World" item. At the bottom right corner of the form subwindow is the "Destroy" command, which destroys the tool.

Loadstate facilities

At the bottom left of the form subwindow are "List LoadState" and "List Context". "List LoadState" enumerates the currently loaded configurations in the client. If you type the name of a configuration in the "Context" string item, then click "List Context", Sword enumerates the modules in that configuration.

Process facilities

The process facilities are based on the concept of *freezing* a Mesa process at a particular frame on its callstack. When a process is frozen, the process continues execution normally, but when it returns to the frozen point on its callstack, the process stops executing until it is *thawed*. If the freeze point is the current frame, the process stops executing immediately. Section E.3.1.1 contains an example of freezing.

The Sword tool allows you to freeze and thaw a process, and to look at the frozen frames of a process (even while the frames of the process hotter than the freezing point are still executing). There is no way to look at a process (or part of a process) that isn't frozen. There are four "Freeze" buttons. "All" freezes all processes at their current frame. "Ready" freezes all ready processes (processes that aren't suspended for a monitor lock, condition variable wait or fault) at their current frame. "Process" freezes the process whose number (PSB index) is given in octal in the "PsblIndex" string item. Finally, "Context" freezes all processes associated with a particular configuration or module, freezing them at the boundary of the context. More precisely, "Context" enumerates all processes, and determines every process that has on its callstack a frame within the specified module or configuration; each such process is frozen at the point where control would return to within the module or configuration; any process currently executing (or waiting or faulted) within the module or configuration stops executing immediately. If you resume the client while some processes are frozen, they really are frozen when the client resumes. This can be important when debugging parallel computations, but it can be dangerous. In particular, beware of resuming the client while critical system processes are frozen! You cannot freeze the ready processes of the local world. You should use the freeze context command as much as possible, so that you don't freeze anything unexpectedly.

Examining frozen processes

For each process that is frozen the Sword tool displays a description of the process's current state and three buttons: "Adjust", "Thaw", and "Debug". The state says things like "ready" or "waitingCV" or "pageFault", followed by the name of the frame at which the process is frozen and the priority of the process. There is no way to look at non-frozen frames of the process. The state is wrapped in parentheses if the process's current frame is not frozen (that is, if the process is still executing. This can happen when you freeze a context.). If you click the "Debug" button, an Interpreter tool is created for that process, so you can look at the frozen parts of the callstack in more detail. If you click the "Thaw" button, the process is unfrozen and continues execution. If you click the "Adjust" button, the entire callstack of the process is refrozen (you might want to refreeze a process after thawing it). You can't thaw or adjust a process while it has an Interpreter; proceed or abort the Interpreter first (as described below).

E.3.1.1 Example: Freezing a Process

Say we are running the following program and we wish to stop it and examine its state. It doesn't matter whether we have run this program first or Sword first. This program is a small example of an infinite loop.

```
Foo: PROGRAM = {
    var: CARDINAL ← 0;
    DO Process.Pause[Process.SecondsToTicks[1]];
        var ← var + 1;
    ENDLOOP; }.
```

1. In the Sword tool, bug "Local" in the "World" item.
2. In the "Context" item, type "Foo".
3. In the "List" item, bug "Context". Sword prints the global frame handle for Foo.
4. In the "Freeze" item, bug "Context". Sword freezes the process running Foo and creates an entry in the Frozen Processes subwindow.
5. In the Frozen Processes entry, bug "Debug". An Interpreter is created (see section 24.3.2). List the value of the variable "var".
6. Type "Proceed" in the Interpreter.
7. In the Frozen Processes entry, bug "Thaw". The process continues running.
8. In the Frozen Processes entry, bug "Adjust". The process is frozen again.
9. In the Frozen Processes entry, bug "Debug". In the Interpreter, see that the value of the variable has changed.
10. Type "Quit" in the Interpreter.
11. In the Frozen Processes entry, bug "Thaw". The process is aborted.

E.3.2 Interpreter Tool

Sword registers the Executive command "Interpreter", which may be used to create an Interpreter tool. Interpreters are created automatically to handle certain local events. The four booleans *fault*, *uncaught*, *break*, and *calldebug* in the Options window control whether an event causes a world swap or is handled locally. If a boolean is **TRUE** and the corresponding event occurs, it is handled locally (that is, an Interpreter is created) instead of world-swapping. If a boolean is **FALSE** then the event causes a world-swap. When an event occurs, an existing Interpreter is used if possible. An existing Interpreter can be used if it is dormant (has no client), as indicated in the namestripe of the Interpreter. Dormant Interpreters are destroyed with the *destroy!* command. An Interpreter contains a form subwindow and a file subwindow.

Sessions

An Interpreter that is not dormant represents a debugging session. Debugging sessions are closed by making the Interpreter dormant. Debugging sessions are opened either from the Executive or from the client item in the Interpreter, described below. Sample debugging sessions are in sections E.3.2.2, E.3.2.3, and E.3.2.4.

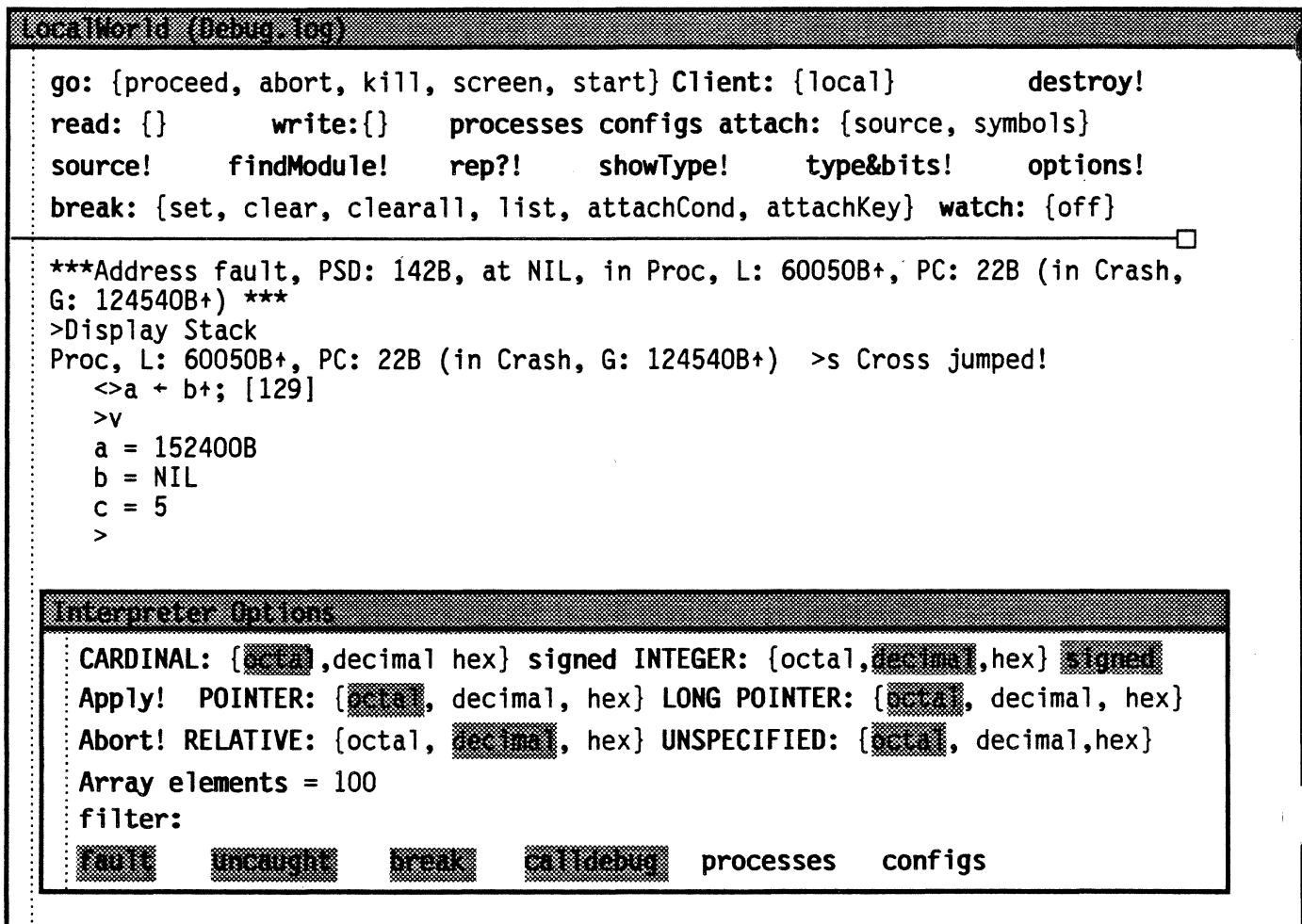


Figure E.2: LocalWorld with Interpreter Options

E.3.2.1 Interpreter form subwindow

The form subwindow has items for the interpreter commands used most often. Commands available in the form subwindow are described in this section; commands available in the file subwindow are described in later sections.

client

This enumerated item is used to open and close debugging sessions. If the Interpreter is dormant, selecting "Local" will create a local debugging session. Selecting "outload" or "remote" will create an outload or remote session. You will be prompted for the name of the remote host or outload file. To end an outload or local debugging session, change this enumerated to "dormant". To end a remote debugging session, change this enumerated to "closeRemote".

The debugger has a programmatic interface called "DebugUsefulDefs", which makes available some debugger functions. For instance, an application can read and write the memory of a remote host through DebugUsefulDefs, if there is a remote debugging session with that host. Because Sword can be debugging many clients at once, the DebugUsefulDefs interface needs to know which client it should use for its functions. The

user tells DebugUsefulDefs which client to use by selecting "setDUD" in client item. DebugUsefulDefs will use the client from that session until the user does another "setDUD". After the user ends the session, DebugUsefulDefs will return undefined results from its functions.

processes

Turning the processes boolean on creates the *process subwindow* in the Interpreter. The subwindow contains processes, callstacks, and local variables. You can zoom or close a particular line by selecting the cross (X) at the head of the line (the state is toggled). Zooming displays more detail; for instance, zooming a stack frame line displays the local variables of the stack frame. Closing the line erases the local variables. The window is automatically initialized to the process context of the crash. It takes about 10 seconds from the time of the crash until the processes subwindow is displayed.

configs

Turning the configs boolean on creates a *configs subwindow* in the Interpreter. The subwindow contains configurations, modules, and global variables. You can zoom or close a particular line by selecting the cross (X) at the head of the line (the state is toggled). Zooming displays more detail; for instance, zooming a configuration line displays the nested configurations and modules. Zooming a module line displays the global variables of the module. Closing the line erases the global variables. The window is automatically initialized to the module context of the crash.

source

If the current selection is on a local frame line in the process subwindow, then **source!** loads a filewindow with the source for that local frame. If the current selection is on a global frame line in the config subwindow, then **source!** loads a filewindow with the source for that global frame.

findModule

If the current selection is on a stack frame line in the process subwindow, then **findModule!** marks (with a black box) the configuration in the config subwindow that contains the module to which the stack frame points. If the current selection is on a module line in the config subwindow, then **findModule!** marks the processes in the process subwindow that contain stack frames which point to the module.

rep?

The value of the selected number is printed in several formats, including octal, decimal, and hexadecimal.

showType

The type of the selected expression is printed. The syntax of the expression must be either File.Type or File\$Type, where File is the name of an interface or program. If just a filename is selected, then all of the types in that file are printed.

type&bits

The type and bit layout of the selected expression is printed. This is particularly useful for finding the positions of fields within records. The syntax of the expression must be either File.Type or File\$Type, where File is the name of an interface or program.

watch

This command will stop execution of a program whenever the contents of a particular address changes. When this enumerated is set on, you are prompted for an address to watch. Afterwards, if the contents of the address changes, the debugger will be called with a swap reason of "TraceTrap". The enumerated should then be set off, to stop watching.

E.3.2.2 Example: A Local Debugging Session

1. Run Sword.
2. Type "Interpreter" in the Executive. An Interpreter tool is created, and its namestripe is "LocalWorld (Debug.log)".
3. Bug "Proceed" in the "go" item of the Interpreter form subwindow. The namestripe changes to "Dormant (Debug.log)".
4. Select "Local" in the "client" item of the Interpreter. The namestripe changes to "LocalWorld (Debug.log)".
5. Bug "Proceed". The namestripe changes to "Dormant (Debug.log)".
6. Bug "destroy". The Interpreter is destroyed.

E.3.2.3 Example: Two Outload Debugging Sessions

This example assumes that you are running Sword on a volume of type debugger, and that you have two volumes of type normal. Let us call the two volumes "NormalOne" and "NormalTwo". Let us assume you have enough room on your debugger volume for three outload files: "Debugger.outload", "NormalOne.outload", and "NormalTwo.outload".

1. In the [Debugger] section of your User.cm put
NormalOne: NormalOne.outload
NormalTwo: NormalTwo.outload
This tells Sword the names of outload files to use for particular volumes.
2. Run Sword.
3. Boot NormalOne from the HeraldWindow.
4. SHIFT-STOP. You will world-swap to the debugger volume.
5. Type "Interpreter NormalOne.outload/o". An Interpreter tool is created, and its namestripe is "Outload: NormalOne.outload, volume: NormalOne (Debug.log)". In the file subwindow is the swap reason, "Interrupt".
6. Bug "Proceed" in the "go" item of the Interpreter form subwindow. The namestripe changes to "Dormant (Debug.log)", and you world-swap to NormalOne.
7. SHIFT-STOP. You will world-swap to the debugger volume. In the file subwindow of the Interpreter is the swap reason, "Interrupt".
8. Boot NormalTwo from the HeraldWindow.
9. SHIFT-STOP. You will world-swap to the debugger volume.
10. Type "Interpreter NormalTwo.outload/o". Another Interpreter tool is created, and its

namestripe is "Outload: NormalTwo.outload, volume: NormalTwo (Debug.log1)". In the file subwindow is the swap reason, "Interrupt".

From this point, you can bug proceed in either of the Interpreters, and you will swap to the respective volume. To end the session in either Interpreter, select "dormant" in the client item of the form subwindow.

E.3.2.4 Example: A Remote Debugging Session

This example assumes that "Host" is the network name of a machine which is in 915, waiting to be remote debugged.

1. Login. Run Sword.
2. Type "Interpreter Host/r" in the Executive. Status messages are printed in the Executive as the debugger tries to open a connection to Host. If the connection is made, an Interpreter tool is created, and its namestripe is "Remote: Host (Debug.log)". The MP code of Host changes from 915 to 917. In the HeraldWindow, a pair of boxes appear, and they twiddle once for each page of data that is fetched from Host.
3. Bug "Proceed" in the "go" item of the Interpreter form subwindow. The namestripe changes to "Dormant (Debug.log)". The Host is now running and its MP code is 990.
4. Select "closeRemote" in the "client" item of the Interpreter. The twiddling boxes disappear, indicating that the connection to Host is closed.

E.3.2.5 Interpreter file subwindow

The file subwindow is recorded in a log file, which is named in the tool's namestripe. The file subwindow is a command processor, and the prompt character is >. The standard input editing characters (**BS** to delete a character and **BW** to delete a word) are allowed. Whenever a valid command is recognized, the Interpreter prompts for the parameters associated with that command (if any are required). Pressing **DELETE** terminates the command; ? gives a list of valid commands. Pressing **ABORT** at any point during command execution aborts the command. When receiving commands, the debugger extends each input character to the maximal unique string that it specifies. Whenever an invalid character is typed, a ? is displayed and you are returned to command level. Typing ? at any point during command selection prompts you with the collection of valid characters (in upper case) and their associated maximal strings (in lower case) and returns you to command level.

Current Context

Interpreting symbols (including displaying variables, setting breakpoints, and calling procedures) occurs in the *current context*; it consists of the current stack frame and its corresponding module, configuration, and process. The symbol lookup algorithm used by the debugger is to search the runtime stack of procedure frames in *Last-In-First-Out* order. First the local frame of the current procedure is examined, next its associated global frame. The search continues by following the return link to the next local frame. This continues until either the symbol is found or the root of the process is encountered.

When you first enter the debugger, the context is set to the frame of whatever process is currently running. Certain commands make it simple to enumerate contexts (**List Processes**, **List Configurations**), to change between contexts (**SEt Root**

configuration, **SEt Module context**), to display the current context (**CUrrent context**), and to examine the current dynamic state (**Display Stack**).

Looking up Symbols

Whenever the debugger needs symbols to display information, it first searches for symbols where they were last copied by the Binder, then for the original compiler-output object file. Types used, but not declared, within a module are looked up using the same algorithm as in the Compiler. If the interface module containing the original declaration is unavailable, the debugger uses whatever information has been copied into the symbol table of the module using that type.

Leaving the Debugger

In the debugger, you may execute any number of commands to examine (and change) the state of your program. When you are finished, you may decide either to continue execution of your program (**P**roceed), terminate execution of your program (**Q**uit), or end the debugging session completely and boot the physical volume (**K**ill).

E.3.2.6 Input conventions

String Input

Identifiers are sequences of characters beginning with an upper- or lower-case letter and terminating with a space (**SPACE**) or a carriage return (**RETURN**); *identifiers must be typed with correct capitalization*.

Numeric Input

A numeric parameter is a sequence of characters terminated by **SPACE** or **RETURN**. If the parameter is not a numeric constant, it is processed by the interpreter; any expression that evaluates to a number is legal. The default radix is octal for addresses (and input to octal commands) and decimal for everything else, unless otherwise specified with the Options window. The **D** or **d** suffix forces decimal interpretation; **B** or **b** forces octal; **H** or **h** forces hexadecimal. Numeric constants with a leading zero are considered **LONG**.

Default Values

The debugger saves the last used command parameters for each command; these values may be recalled by the **COMPLETE** key (aka →). The following parameters have default values that may be recalled with **COMPLETE**: *octal read address, octal write address, ascii read address, root configuration, configuration, module, procedure, condition, expression, process, address, and frame*. After the default parameter is displayed by the debugger, the standard input editing characters may be used to modify it. Striking the **COMPLETE** key to the command processor uses the last command as the default command.

E.3.2.7 Output conventions

A "?" in any variable display uniformly means that the value is out of range. An ellipsis ("...") indicates that there are additional fields present in a record that cannot be displayed due to lack of symbol table information. This can happen either in **OVERLaid**

records or because a **DEFINITIONS** file is not present on the disk. In display stack mode, variables declared in nested blocks are shown indented according to their nesting level.

The **Options** window allows you to change the default format the debugger uses in displaying values of variables. This window is created by selecting the **Options** item in the **form** subwindow, then bugging **Apply!** to keep the changes made, or **Abort!** to restore the previous options.

The **CARDINAL**, **INTEGER**, **POINTER**, **LONG POINTER**, and **RELATIVE (POINTER)** items are used to set the default output radix for that type. For **CARDINAL** and **INTEGER**, the default representation is signed or unsigned, depending on whether the boolean item **signed** is turned on or off. The **UNSPECIFIED** item is used to set the default type for displaying **UNSPECIFIED** variables. **Array elements** sets the number of **ARRAY** elements displayed to be the given value and **String length** sets the number of **STRING** characters displayed to the given value.

The debugger uses these default values along with the types of variables to decide on an appropriate output format. Listed below are the built-in types that the debugger distinguishes and the convention used to display instances of each type.

ARRAY

displays elements of an array; e.g., **a = (3)[[x: 0, y:0], [x: 1, y: 1], [x: 3, y:3]]**. The parenthesized value to the right of the "=" is the length of the array. Pressing **ABORT** will abort the display of long arrays. The default is to display the entire array; the **Array elements** item of the **Options** window may be used to change this.

ARRAY DESCRIPTOR

displays the descriptor followed by the contents of the array; e.g., **a = DESCRIPTOR[146013B↑,3](3)[[x: 0, y:0], [x: 1, y: 1], [x: 3, y:3]]**. For a **RELATIVE ARRAY DESCRIPTOR**, the word **RELATIVE** is displayed first. Pressing **ABORT** will abort the display of long array descriptors. The **Array elements** item in the **Options** window also controls this.

BOOLEAN

displays **TRUE** or **FALSE**. Since **BOOLEAN** is an enumerated type = {**FALSE**, **TRUE**}, values outside this range are indicated by a ? (probably an uninitialized variable).

CARDINAL

displays an octal number terminated by a "B" as the default. This may also be altered with the **Options** window. Cardinals may be displayed as decimal, octal, or hex; signed or unsigned.

CHARACTER

displays a printing character (**c**) as '**c**'. A control character (**x**) other than **BLANK**, **RUBOUT**, **NUL**, **TAB**, **LF**, **FF**, **CR**, or **ESC** is displayed as **↑x**. Values greater than 177B are displayed in octal.

CONDITION

displays a record containing an **UNSPECIFIED** and **timeout**, a **CARDINAL**.

ENUMERATED

displays the identifier constant used in the enumerated type declaration. For example, an instance **c** of the type **ChannelState: TYPE = {disconnected, busy, available}** might be displayed as **c=busy**. Values outside the range of the enumerated type are preceded by a question mark.

EXPORTED TYPES

displays the name of the type followed by an octal display of the contents if the length of the type is known. For example, an instance of the type **Handle: TYPE [1]** is displayed as **Handle(1) 1234B**.

INTEGER

always displays a decimal number. Uniformly, numeric output is decimal unless terminated by "B" (octal). Integer output may be changed with the **Options** window.

LONG

displays numbers following the same conventions as short numbers; i.e., **LONG CARDINAL** and **LONG UNSPECIFIED** are displayed in octal, **LONG INTEGER** in decimal.

MONITORLOCK

displays a record containing an **UNSPECIFIED**.

POINTER

displays an octal number, terminated with an "**↑**"; for instance **p=107362B↑**. **RELATIVE POINTERS** are decimal and are terminated with "**↑R**"; for instance **r=123↑R**. These defaults may be changed for **LONG POINTERS**, **RELATIVE POINTERS**, and **POINTERS** to either octal or decimal with the **Options** window.

PART

displays two octal numbers; for instance **p = PART [0, 172520B]**.

PROCEDURE, SIGNAL, ERROR

displays the name of the procedure (with its local frame) and the name of the program module in which it resides (with its global frame); for instance **GetMyChar, L: 165064B (in CollectParams, G: 166514B)**.

PROCESS

displays a **PROCESS** (pointer to a **ProcessStateBlock**); for instance **p = PROCESS [111B]**.

REAL

displays a floating-point number; for instance **-1.45**.

RECORD

displays a bracketed list of each field name and its value. For example, an instance **v** of the record **Vector: RECORD [x,y: INTEGER]** is displayed as **v=[x: 9, y: -1]**.

SEQUENCE

displays as an array. For example, an instance **s** of the record **Sequence: RECORD [length: UnsignedInt, text: PACKED SEQUENCE maxLength: UnsignedInt OF CHARACTER]** is displayed as **s=[length:3, text:(3)['a, 'b, 'c']]**.

STRING

displays the name of the string, followed by its current length, its maximum length, and the string body; for instance **s=(3,10)"foo"**. If the string is **NIL**, **s=NIL** is displayed. Pressing **ABORT** will abort the display of long strings. The default is to display the entire string; the **String length** item in the **Options** window can change this.

UNSPECIFIED

defaults to being displayed as if they were **CARDINALS**; this may be changed with the **Options** window.

ZONE

An **UNCOUNTED ZONE** displays as a **LONG POINTER**. An **MDSZone** displays as a **POINTER**.

Listed below are the conventions used to display context information throughout the debugger:

A local context is displayed as the procedure name with its local frame, followed by the module name and its global frame:

ProcedureName, L: nnnnB, PC: nnnB (in ModuleName, G: nnnnnB)

A global context is displayed as the module name and its global frame. If the global frame has not been started, it is followed by a **~**. If the global frame is followed by ***** (as **nnnnnB***) it is a copy created by the **NEW** construct.

ModuleName, G: nnnnnB

In response to an expression followed by a **?**, the interpreter will show:

```
Octal = Hexadecimal = Unsigned Decimal = Signed Decimal =
Byte,,Byte = Octal Byte,,Octal Byte = CHAR,,CHAR =
Nibble:Nibble,,Nibble:Nibble
```

If any of the values are 0 or out of range, they will not be shown. For **LONG** values the interpreter will show:

```
Octal = Hexadecimal = Decimal = OctalWord OctalWord =
Byte,,Byte Byte,,Byte
```

For example, in response to **61141B?** the debugger displays

```
61141B = 6261H = 25185 = 98,,97 = 142B,,141B = 'b,,'a = 6:2,,6:1
```

and for **1234567B?** it shows

```
1234567B = 53977H = 342391 = 34567B 5 = 57,,119 0,,5
```

E.4 Debugger commands

The command tree structure for the Interpreter appears at the end of this chapter. Capitalized letters are typed by the user (in either upper or lower case); commands are extended with lower-case strings by the command processor. Each command (and its parameters) is described below.

E.4.1 Breakpoints

All breakpoints may be conditional (see **ATTACH CONDITION**, below). An optional command string which is executed when the breakpoint is taken can be attached to each breakpoint (see **ATTACH KEYSTROKES**, below).

Breakpoints may be set at the following locations in a program: entry (to a procedure), exit (from a procedure), and at the closest statement boundary preceding a specific text location within a procedure or module body. Breaks on a specific text location can be set only with the **SET BREAK** command in the form **subwindow**. Note that breakpoints are set in all instances of a module. When the source line of the breakpoint is displayed, the indicator **<>** appears to the left of the source where the breakpoint has actually been set (for instance **IF foo THEN <> some statement;**). Before the debugger permits any breakpoints to be set from a **FileWindow**, the creation date of the source file is checked against the corresponding date recorded by the compiler in the **bcd**.

Fine point: Since there is only one exit from a procedure, the debugger shows the beginning of the procedure for exit breaks instead of indicating a potentially incorrect **RETURN** statement. Local variables may be invisible if this **RETURN** has a **PC** that is not in the block with their declarations; use **SOURCE BREAKS** on the **RETURN** statements instead of an exit break.

If you compile a module with the cross-jumping switch turned on (the default), be warned that when setting source breakpoints, the actual breakpoint may not end up where you expect (for instance you may break in the code of an **ELSE** clause when you really want the **THEN** clause if they share some common code). The message **Cross jumped!** will appear before the source of a cross-jumped module is displayed. Entry and exit breakpoints are not affected by cross jumping.

ATTACH SOURCE (in form subwindow)

tells the debugger to ignore the time stamp in the source file where the current selection is when setting breaks.

ATtach Condition [number, condition]

changes a normal breakpoint into a conditional one. Arguments are a breakpoint number and a condition, which is evaluated in the context of the breakpoint. The breakpoint number is displayed when the break/tracepoint is set, and may also be obtained using the **List Breaks** command. The two valid formats of a **Condition** are: *exp relation exp*, and *number*. A relations is in the set {<, >, =, #, <=, >=}. A *number* means "execute the break *number* times before invoking the debugger." The *exp* are interpreted expressions that are looked up in the context of the breakpoint. The *exp* may only evaluate to a value that is 32 bits long, 16 bits long, or less than 16 bits long. The expression can involve at most once dereference when the expression is evaluated at run time.

ATtach Keystrokes [number, command]

adds an arbitrary command string to breakpoints/tracepoints; the characters from this string are executed by the debugger when the breakpoint/tracepoint is taken. Arguments are a breakpoint number and a command string terminated with a **RETURN**. A **RETURN** can be embedded in the command string with \n.

Set Break (in form subwindow)

uses the current selection to set a breakpoint. If you select **PROCEDURE** or **PROC**, a breakpoint is set on the entry to the procedure; if you select **RETURN**, a breakpoint is set on the exit of the procedure; otherwise, a breakpoint is set at the closest statement to the beginning of the selection. **Note:** If the module was compiled with cross jumping, breaks may be set in unpredictable places. The debugger gives confirmation by moving the selection to the place at which the breakpoint is actually set.

For the following code fragments, a breakpoint set on **anyError** will invoke the debugger after the catch frame is entered. If a breakpoint is set on **MFile.Error**, the debugger is invoked for all signals and errors before any decision is made to catch the signal.

```
BEGIN ENABLE MFile.Error => {anyError ← TRUE; CONTINUE};
```

```
!MFile.Error => {anyError ← TRUE; CONTINUE};
```

Break All Entries [module/frame]

sets a break on the entry point to each procedure in *module* or *frame*, not including nested procedures and catch code.

Break All Xits [module/frame]

sets a break on the exit point of each procedure in *module* or *frame*.

Break Entry [proc]

inserts a breakpoint at the first instruction in the procedure *proc*. **Note:** You can place a breakpoint on the entry to the mainline code, by doing **Break Entry [module name]**.

Break Xit [proc]

inserts a breakpoint at the *last* instruction of the procedure body for *proc*. The breakpoint catches all **RETURN** statements in the procedure. Note: You can place a breakpoint on the exit from the mainline code, by doing **Break Xit [module name]**.

CLear All Breaks

removes all breakpoints/tracepoints.

CLear All Entries [module/frame]

removes all entry breakpoints/tracepoints in *module* or *frame*.

CLear All Xits [module/frame]

removes all exit breakpoints/tracepoints in *module* or *frame*.

CLear All Traces

removes all tracepoints.

CLear Break (in form subwindow)

clears the breakpoint or tracepoint at the location specified as above.

CLear Break [number]

removes a breakpoint by number. Pressing **RETURN** in place of a number clears the current breakpoint, the one that put you into the debugger.

CLear Condition [number]

changes a conditional breakpoint into an unconditional one. Pressing **RETURN** in place of a number clears the current breakpoint.

CLear Keystrokes [number]

clears any command string associated with the breakpoint. Pressing **RETURN** in place of a number clears the current breakpoint.

CLear Entry Break [proc]

converse of **Break Entry**.

CLear Entry Trace [proc]

converse of **Trace Entry**.

CLear Xit Break [proc]

converse of **Break Xit**.

Clear **Xit Trace [proc]**

converse of **Trace Xit**.

Display **Break [number]**

displays a breakpoint by number. Its type (entry, exit, source), and the procedure and/or module name in which it is found are displayed; for source breakpoints, the source text is also displayed; any attached conditions or keystrokes are also shown. Pressing **RETURN** in place of a number displays the current breakpoint.

List **Breaks [confirm]**

lists all breakpoints, displaying the same information as **Display Break**.

Trace **All Entries [module/frame]**

sets a trace on the entry point to each procedure in *module* or *frame*.

Trace **All Xits [module/frame]**

sets a trace on the exit point of each procedure in *module* or *frame*.

Trace **Entry [proc]**

sets a trace on the entry of the procedure *proc*. When an entry tracepoint is encountered, display stack mode is entered and the parameters are displayed.

Trace **Xit [proc]**

sets a trace on the exit of the procedure *proc*. When an exit tracepoint is encountered, display stack mode is entered and the return values are displayed.

E.4.2 Display runtime state

The scope of variable lookup is limited to the current context (unless otherwise specified below to be the current configuration). This means that if the current context is a local frame, the debugger examines the local frame of each procedure in the call stack (and its associated global frame) following return links until the root of the call stack is reached. If the current context is a module (global) context, just the global frame is searched. Global frames are searched in the order: declarations, imports, directory. If the variable you wish to examine is not within the current context, change contexts.

ASCII **Read [address, n]**

displays *n* (decimal) characters as text starting at *address* (octal).

Display **Configuration**

displays the name of the current configuration followed by the module name, corresponding global frame address, and instance name (if one exists) of each module in the current configuration.

Display Frame [address]

displays the contents of a frame, where *address* is its octal address (useful if you have several instances of the same module or examining a specific local frame). Display stack mode is entered.

Display GlobalFrameTable

displays the module name and corresponding global frame address, pc, codebase, and gfi of all entries in the global frame table.

Display Module [module]

displays the contents of a global frame, where *module* is the name of a program in the current configuration.

Display Process [process]

displays the frame and the state of *process*, which may be a variable of type **PROCESS** (returned as the result of a **FORK**) or an octal **PROCESS**. The state of *process* can be:

ready (ready to run and has a state vector)

waiting SV (ready to run but needs a state vector)

waiting ML (waiting on a monitor)

waiting CV (waiting on a condition variable)

frame fault, fsi: nn (needs a frame whose size index is *nn*)

page fault, address: nnnnnB (waiting for page whose address is *nnnnnB*; this is an address fault if location *nnnnnB* isn't mapped)

write fault, address: nnnnnB (waiting to write into location *nnnnnB*, which is write-protected)

faulted (unknown fault has occurred)

A * marks the current process. A process can be on one and only one queue (associated with a condition, monitor, ReadyList, etc.). After the process is displayed, you enter process subcommand mode. A response of **N** displays the next process; **S** displays the source text and loads and positions the source file in a window; **L** just displays the source text; **R** displays the root frame of the process; **P** displays the priority of the process; space (**SPACE**) enters the interpreter;-- starts a comment; and **Q** or **DELETE** terminates the display and returns you to the command processor.

Display Queue [id]

displays all the processes waiting on the queue associated with *id*. If *id* is simply an octal number, you are asked whether it is a condition variable **Condition? [Y or N]**. For each process, you enter process subcommand mode. The commands are the same as in **Display Process**, with the exception of **N**, which in this case follows the link in the

process. This command accepts either a condition variable, a monitor lock, a monitored record, a monitored program, or an octal pointer.

Display ReadyList

displays all the processes waiting on the queue associated with the **ReadyList**, the list of processes ready to run. For each process, you enter process subcommand mode; the commands are the same as in **Display Queue**.

Display Stack

displays the procedure call stack of the current process. At each frame, the corresponding procedure name and frame address are displayed. The commands are:

- V** displays all the frame's variables.
- G** displays the global variables of the module containing the current frame.
- P** displays the input parameters.
- R** displays the return values. (**anon**) appears as the name of unnamed return values.
- N** moves to the next frame on the call stack.
- J, n(10)** jumps down the stack n levels (if n is greater than the number of levels it can advance, the debugger tells you how far it was able to go).
- S** displays the source text and loads and positions the source file in a window.
- L** displays the source text.
- SPACE** enters the interpreter.
- starts a comment which ends with a **RETURN**.

Q or DELETE

terminates display stack mode and returns you to the command processor.

When the current context is a global frame, the **Display Stack** subcommands **G**, **J**, and **N** are disabled. When the debugger cannot find the symbol table for a frame on the call stack, only the **J**, **N**, **Q**, -- and **SPACE** subcommands are allowed.

Find variable [id]

displays the contents and module location of the variable named *id*, searching through the **GlobalFrames** of all the modules in the current configuration.

Find module [id]

displays the processes and local frames which are in the module id, searching through all of the processes. The information is printed out in a form that can be copied directly to the Interpreter. For instance:

```
Find Module: FooImpl
SEP130B
DSJ10
```

If the characters are copied directly into the Interpreter, context will be set to process 130B, at the tenth frame on the call stack.

E.4.3 Current context

The current context is used to determine the domain for symbol lookup. There are commands to display the current context, to display all the configurations and processes, to restore the starting context, and to change contexts.

Every time the debugger is entered, the current context is automatically set to (1) the process that caused the debugger to be called; (2) some significant frame in the calling process, not necessarily top of the call stack of the process (for example, for an uncaught signal, the significant frame is the one in which the signal was raised); and (3) the module and configuration of the local frame set in (2).

CURRENT CONTEXT

displays the name and corresponding global frame address (and instance name if one exists) of the current module, the name of the current configuration, and the **PROCESS** for the current process.

LIST CONFIGURATIONS

lists the name of each configuration that is loaded, beginning with the last configuration loaded. If you wish to see more information about a particular configuration, use the **DISPLAY CONFIGURATION** command.

LIST PROCESSES

lists all processes as in the **DISPLAY PROCESS** command.

RESET CONTEXT

restores the context that this instance of the debugger set upon entry (see the introduction to this section).

SET CONFIGURATION [config]

sets the current configuration to be *config*, where *config* is nested within the root configuration that is current. This command is useful for "jumping" further into the nested block structure of a configuration.

SEt Module context [module/frame]

changes the context to the program module whose name is *module* (within the current configuration). If there is more than one instance of *module*, the debugger lists the frame address of each instance and does *not* change the context. Using a *frame* address has the same effect as **SEt Octal context**.

SEt Octal context [address]

changes the current context to the frame at *address*. This is useful when there are several instances of the same module or in setting the current context to a specific local frame.

SEt Process context [process]

sets the current process context to be *process* and sets the corresponding frame context to be the top frame on the call stack of that *process*. Upon entering the debugger, the process context is set to the currently running process. The *process* may be either a variable of type **PROCESS** (returned as the result of a **FORK**) or an octal **PROCESS**.

SEt Root configuration [config]

sets the current configuration to be *config*, where *config* is at the outermost level (of its configuration). This command is sufficient for simple configurations of only one level. It is also useful in getting you to the outermost level of nested configurations, from which you may move "in" to more deeply nested configurations using **SEt Configuration**.

E.4.4 Program control

Kill session [confirm]

ends the debugging session, and executes **TemporaryBooting.BootButton** in the client.

Proceed [confirm]

continues execution of the program.

Quit [confirm]

raises the signal **ABORTED** in the process that entered the debugger. If the process was already processing an uncaught **ABORTED** signal (perhaps from a previous **Quit** command), this command passes the signal **UNWIND** to each frame of the process and then simulates a **RETURN** with no results by the root frame of the process, causing the process to be deleted. If this process is supposed to return any results, a stack error will result.

STart [address] [Confirm]

starts execution of the module whose frame is *address*. If the module has already been started, a **RESTART** will be done. Unlike the **START** statement in the Mesa language, no parameters may be passed.

Userscreen [confirm]

swaps to the user world for a look at the screen. Control is returned to the debugger automatically after 20 seconds or by typing the **ABORT** key earlier; it does not return until the **ABORT** key is let up.

E.4.5 Low-level facilities

ATtach Symbols [globalframe, filename]

attaches the *globalframe* to *filename*. **ATtach Symbols** is useful for allowing you to bring in additional symbols for debugging purposes when you do not have the correct object file. The default extension for *filename* is **.bcd**. Neither interfaces nor **.symbols** files can be attached.

Warning: This command overrides version checking of symbol tables and should be used with caution; it may cause the debugger to display incorrect values.

Display Eval-stack

displays the contents of the Mesa evaluation stack (in octal), which is useful for low-level debugging or for displaying the (un-named) return values of a procedure that has been broken at its exit point. This command is most useful at octal breakpoints because the eval stack is empty between most source level statements.

Octal Clear break [globalframe, bytepc]

is the converse of **Octal Set break**.

Octal Read [address, n]

displays the *n* (decimal) locations starting at *address*. An *address* in the first 64K is interpreted as an absolute virtual address if it has a leading zero; it is treated as MDS-relative otherwise.

Octal Set break [globalframe, bytepc]

sets a breakpoint at the byte offset *bytepc* in the code segment of the frame *globalframe*.

Octal Write [address, rhs]

stores *rhs* (octal) into the location *address*.

-- [comment]

starts a comment which ends with a **RETURN**.

E.5 The Debugger interpreter

The Mesa interpreter handles a subset of the Mesa language; it is useful for common operations such as assignments, dereferencing, procedure calls, indexing, field access of records, addressing, displaying variables and **TYPES**, and simple type conversion.

Only a specific subset of the Mesa language is acceptable to the interpreter (see the end of this chapter for grammar details). Several specialized notations (abbreviations) have been introduced in the interpreter grammar; these are valid only for debugging purposes and are not part of the Mesa language. The interpreter operates much like the Compiler in that strict type checking is performed on assignments and procedure calls.

E.5.1 Statement syntax

Typing **SPACE** to the command processor enables interpreter mode; the limited command processors of **Display Stack** and **Display Process** also permit interpreting. Multiple statements are separated by semicolons. If the statement is a simple expression (not an assignment), the result is displayed after evaluation.

E.5.2 Loopholes

A more concise **LOOPHOLE** notation has been introduced to make it easy to display arbitrary data in any format. The character **%** may be used instead of **LOOPHOLE[exp, type]**, with the expression on the left of the **%**, and the **type** on the right. However, **%** is not a valid **LeftSide**; all **type** expressions involving **%** must be enclosed in parentheses.

The following expressions are equivalent to the interpreter:

```
foo % (short red Foo) and LOOPHOLE[foo, short red Foo]  

(p % (LONG POINTER TO Object))↑ and LOOPHOLE[p, LONG POINTER TO Object]↑
```

The first pair of expressions loopholes the type of the variable **foo** to be a **short red Foo** and displays its value. The second pair loopholes **p** to be a **LONG POINTER TO Object** and dereferences it. **foo %** is a shorthand notation for **foo % UNSPECIFIED**.

A number may be loopholed into **PROCEDURE**, **SIGNAL**, or an **ERROR**. If it is valid, the debugger will display the procedure (or signal) name, module and global frame.

E.5.3 Subscripting

There are two types of interval notation acceptable to the interpreter; the closed, open, and half-open interval notation accepted by the Compiler and a shorthand version that uses **!**. The notation **[a . . . b]** means start at index **a** and end at index **b**. The notation **[a : b]** means start at index **a** and end at index **(a+b-1)**.

The following expressions all display the contents of MDS-relative memory locations **1104B** through **1107B**:

```
MEMORY[1104 . . 1107]  

MEMORY[1104 . . 1108)
```

```
MEMORY(1103 . . 1107]
MEMORY(1103 . . 1108)
MEMORY[1104 ! 4]
```

Note that the interval notation is only valid for display purposes and therefore is not allowed as a **LeftSide** or inside other expressions.

E.5.4 Explicit qualification vs qualification in the current context

The \$ notation has been introduced to distinguish between qualification in the current context and explicit qualification. The character \$ indicates that the name on the left is a module name or frame in which to look up the identifier or **TYPE** on the right. If a module cannot be found, it uses the name as a file (usually a definitions file).

For example, **FSP\$TheHeap** means look in the module **FSP** to find the value of the variable **TheHeap**. In dealing with variant records, be sure to specify the variant part of the record before the record name itself (e.g., **foo & (short red FooDefs\$FOO)**, not **foo & (FooDefs\$short red Foo)**).

E.5.5 Type expressions

The notation **@type** may be used as shorthand to construct a **POINTER TO type**. This notation is used for constructing types in **LOPHOLES** (ie., **@foo** will give you the type **POINTER TO foo**). There is no special shorthand to construct **LONG POINTER TO type**; however, **LONG @type** is legal.

E.5.6 Radix conversion

The notation **expression?** prints the value of the expression in several formats, including octal, decimal, and hex. Output radix may be controlled through the **Options** window.

E.5.7 Arithmetic expressions

Target typing is applied to some arithmetic expressions. In complex expressions, atoms that change the target type should occur first. For example:

```
(POINTER + offset)↑ -- correct
(offset + POINTER)↑ -- error message
```

E.5.8 Procedure calls

It is often useful to call procedures; this is generally done in the interpreter with the same syntax as in Mesa. The interpreter is able to invoke any procedure that is imported into the current module context; the \$ notation may be used to call procedures that are not imported.

The interpreter can only call procedures in modules for which it has complete symbols; this can be somewhat confusing since the interpreter "knows" a little about the procedures imported into a module it has symbols for. To determine whether the interpreter has symbols for a procedure and where it is implemented (a more useful feature), simply type

the procedure name to the interpreter. For example, typing either **Process.SetPriority** or **SetPriority** to the interpreter (while inside a module that imports it) will cause the debugger to display something like:

```
SetPriority = PROCEDURE [5461B] (in module Processes, G:11644B)
```

when symbols for **Processes** are not available. Reinterpreting **SetPriority** after retrieving the object file for **Processes** gives the following result:

```
SetPriority = PROCEDURE SetPriority (in module Processes, G:11644B)
```

The notation **Process.SetPriority** means the same to the interpreter as to the Mesa compiler; **SetPriority** is a procedure imported through the **Process** interface.

Since **SetPriority** is imported in this example, you could, for example, *call* it (nicknamed *interpret call* for historical reasons) by typing **SetPriority[1]**. To call **Process.Abort**, which is not imported, the notation **Processes\$Abort[processId]** or **nnnnnB\$Abort[processId]** (where **nnnnnB** is the global frame of **Processes**) works. If you are lacking a variable of type **PROCESS**, **Processes\$Abort[20B\$]** works; it loopholes the process ID number 20B into an **UNSPECIFIED**. (The trailing **\$** notation is a very easy method for constructing pointers; e.g., **123456B\$** is easier to type in a procedure call than **LOOPHOLE[123456B, POINTER]**.)

E.5.9 Sample expressions

Here are some sample expressions that combine several of the rules into useful combinations:

If you were interested in seeing which procedure is associated with the third keyword of the menu belonging to a particular window called **myWindow**, you would type:

```
> myWindow.menu.array[3].proc
```

which might produce the following output:

```
CreateWindow (PROCEDURE in WEWindows, G: 120134B).
```

The basic arithmetic operations are provided by the interpreter (with the same precedence rules as followed by the Mesa compiler).

```
> 3+4 MOD 2
```

would give the answer 3. A typical sequence of expressions one might use to initialize a record containing a pointer to an array of **Foos** and display some of them would be:

```
> rec.array ← FSP$AllocateHeapNode[n*SIZE[fooDefs$foo]];
> InitArray[rec.array]; rec.array[first..last]
```

The following command would display **rec** in octal:

```
>Octal Read: @rec, n: SIZE[Rec]
```

To find out what type a `HeapImpl.Handle` pointed to:

```
> HeapImpl$Handle  
Handle: PRIVATE TYPE = LONG POINTER TO Data
```

E.6 Signal and error messages

The following messages are generated by the debugger.

E.6.1 Entering the Debugger

The following messages from the debugger tell why the debugger was entered. If the situation permits, you may proceed execution of the program with a `Proceed` command. Proceeding from an `ERROR` causes a `ResumeError`. Programs often allow themselves to be aborted by the debugger's `Quit` command; it raises the `ERROR ABORTED` in the client process. If no client catches this error, the debugger will be called again.

***** Interrupt *****

An interrupt occurred, meaning `SHIFT-STOP` (aka `CALLDEBUG`) was typed.

***** uncaught SIGNAL SOS (in MayDay) *****

The program has raised a `SIGNAL` or `ERROR` which no one dynamically nested above the `SIGNAL` invocation was prepared to catch. At this point you might display the stack to see who raised the uncaught `SIGNAL`.

***** Address Fault at xxx (in MayDay) *****

The program has tried to access an unmapped address.

Eval stack not empty!

This warning is printed if the debugger is entered with values still on the evaluation stack; this indicates that the current value of some variables may not be in main memory, where the interpreter normally looks, and so incorrect values may be given. Exceptions to this are entry and exit breaks; the debugger has enough information to decode the argument records that are on the stack in this case.

***** Invalid Load State *****

The debugger has been entered without the client's load state available, probably because a client program smashed the load state. The load state is used by the debugger to translate numbers, such as global frames, into English for the user; without the load state only octal debugging features are available.

E.6.2 Symbol lookup

xxx cannot be acquired with read access!

The file named `xxx` exists, but cannot be read.

xxx not found!

The variable or file named **xxx** cannot be found.

!File: xxx

The file named **xxx** cannot be found.

nnnnnB not started!

The global frame **nnnnnB** has not yet been started. Any variables in the frame are uninitialized.

xxx not bound!

The imported variable **xxx** is not exported by anyone.

xxx has incorrect version!

The symbol file has an incorrect version stamp.

!Tree for xxx not in symbol table

A multiword constant in your code wasn't copied into the symbol table. Look in the source file to find the value.

Use Interface.importedVariable, not Interface\$importedVariable

The debugger cannot find imported variables from an interface file (the "\$" notation). The "." notation will tell it to use the interface record (if found) available in the current context.

E.6.3 Unrecognized structures

```
!Can't find links from frame: nnnnnB
!Invalid global frame
xxx not a frame!
xxx has a NULL returnlink!
xxx has a clobbered accesslink!
xxx is a clobbered frame!
xxx is an invalid PROCESS!
xxx is an invalid global frame!
xxx is an invalid image file!
xxx is not a valid frame!
```

The structure in question appears to be clobbered (invalid in some way).

E.6.4 Command execution errors

Can't use <module> of <time> instead of <time>

This message is printed if the creation date in the source, object, or symbols file on your disk is different than the corresponding date recorded by the Compiler or Binder. The requested version of the file should be retrieved.

!Number

An invalid number has been typed.

xxx is a definitions file!

You have tried to set a break in a definitions file.

xxx not a REAL!

xxx is not a valid representation of a real number.

!Invalid Address [nnnnB]

During the execution of a command, the debugger attempted to read or write location nnnnB, which was not mapped. I/O pages and pages belonging to the germ appear unmapped to the debugger.

!Write protected [nnnnB]

During the execution of a command, the debugger attempted to write location nnnnB, which was write-protected.

E.6.5 Breakpoints

Multiple instances; Use Display Stack, Source to load window.

You have tried to set a break when multiple instances of the module exist; explicitly setting the context for the source window will permit the break to be set.

too many conditional breaks!

You have tried to set more conditional breaks than the system allows.

invalid relation!

You have specified an illegal relation expression for a condition.

symboltable missing!

The debugger is trying to manipulate a breakpoint for which there is no symboltable and it is not prepared to handle the situation.

not allowed in INLINE!

You have tried to set a breakpoint in an **INLINE** procedure.

already set!

You have already set a breakpoint there.

! Patch table full

The maximum number of breakpoints (50) allowed by Pilot has been reached.

E.6.6 Displaying the stack

No previous frame!

The end of the call stack has been reached.

No symbol table for nnnnnnB

The symbol table file corresponding to the frame **nnnnnnB** is missing; any attempt to symbolically reference variables in this module will fail.

Cross jumped!

The bcd was compiled with the cross-jumping switch turned on. The source line displayed may not be what you expect.

Pc not in any procedure!

The debugger was unable to find a procedure or mainline code that matched the current pc. This is probably due to a clobber.

E.6.7 Interpreter

! x is an invalid character

The character **x** typed to the interpreter is illegal.

! Syntax error at [n]

There was a syntax error at location **n** in the expression given the interpreter.

! Parse error at [n]

There was an error at location **n** parsing the expression given the interpreter.

can't call an INLINE!

You tried to call a **INLINE PROCEDURE**.

can't lengthen!

The interpreter needed to lengthen a part of an expression while trying to evaluate it.

can't make a constructor!

Use field assignments. You gave the interpreter an expression using [] that looks like a constructor.

double word array index!

The index for an array must be a single word.

has an invalid address!

The expression to the right of the @ is not word-aligned.

is an invalid number!

This is probably a type mismatch.

is an invalid pointer!

This is probably a type mismatch.

invalid subrange!

This is probably a type mismatch.

pointer fault!

You tried to dereference NIL.

xxx is a constant array. Look at source code for value.

An operation on a constant array is too complicated to perform. The operation can be done by hand, however, by looking at the constant value in the source.

xxx is not an array!

You have tried to use xxx as an array.

is not a valid control link!

The procedure or signal in your expression has an illegal value.

is not a relative pointer!

In the expression base[rel], rel wasn't a RELATIVE POINTER.

is not a type!

The identifier used in a type expression was not a type.

is not a unique field selector!

The field selector occurs more than once in the computed or overlaid variant.

is not a valid field selector!

The identifier given for a field selector is not in the record. You may lack the symbols for the record declaration on your disk.

overflow!

Overflow occurred while doing arithmetic. Perhaps you need a **LONG** in the expression.

size mismatch!

You tried to assign or loophole two things of different sizes. Loopholing pointers is a useful trick for records of different sizes.

has incorrect type!

Type mismatch.

unknown variant!

The interpreter found a garbage tag field.

is the wrong base!

In the expression **base[rel]**, the type of **base** is not what **rel** expects.

has the wrong number of arguments!

The arguments to a procedure call are wrong.

used incorrectly with []!

You probably tried to use **[]** as a type constructor.

illegal indexing operation

You tried to index something that wasn't an array or sequence.

xxx\$ is ambiguous; use frame \$!

There is either more than one instance of **xxx** instantiated, or the code for **xxx** is packed with another module.

E.7 User.cm

The User.cm entries are read when Sword is loaded.

```
[Debugger]
uncaught: TRUE | FALSE -- handle uncaught signals locally, default true
fault: TRUE | FALSE -- handle faults locally, default true
break: TRUE | FALSE -- handle breakpoints locally, default true
calldebug: TRUE | FALSE -- handle calldebugs locally, default true
processes: TRUE | FALSE -- create a process subwindow in the Interpreter tool
configs: TRUE | FALSE -- create a config subwindow in the Interpreter tool
menu: TRUE | FALSE -- create a menu in the root window for creating Interpreters
cRadix: octal | decimal | hex -- radix for cardinals
cSigned: TRUE | FALSE -- print cardinals as signed
iRadix: octal | decimal | hex -- radix for integers
iSigned: TRUE | FALSE -- print integers as signed
pRadix: octal | decimal | hex -- radix for pointers
processRadix: octal | decimal | hex -- radix for processes
relRadix: octal | decimal | hex -- radix for relative pointers
unspec: octal | decimal | hex -- how to print UNSPECIFIED
elements: number -- number of array elements to display
chars: number -- number of characters of a string to display
volume1: outfile1 -- booting the volume will use the outfile
volume2: outfile2 -- the default outfile is "Debuggee.outload"
```



```
[Sword]
cardinalRadix: octal | decimal | hex -- radix for cardinals in Sword tool
processRadix: octal | decimal | hex -- radix for processes in Sword tool
```

E.8 Mesa Interpreter grammar

StatementList	::=	Statement StatementList; StatementList; Statement
Statement	::=	LeftSide Interval LeftSide ← Expression MEMORY Interval Expression Expression ?
LeftSide	::=	identifier (Expression) LeftSide Qualifier identifier \$ identifier number \$ identifier MEMORY [Expression] LOOPHOLE [Expression] LOOPHOLE [Expression , TypeExpression]
Qualifier	::=	.identifier [ExpressionList]
Interval	::=	[Bounds] [Bounds) (Bounds] (Bounds) [Expression ! Expression]
Bounds	::=	Expression .. Expression

Expression	:: = Sum
Sum	:: = Product Sum AddOp Product
AddOp	:: = + -
Product	:: = Factor Product MultOp Factor
MultOp	:: = * / MOD
Factor	:: = Primary Primary
Primary	:: = Literal LeftSide @ LeftSide BuiltinCall Primary % Primary % (TypeExpression)
Literal	:: = number character string
BuiltinCall	:: = NIL NIL [TypeExpression] PrefixOp [ExpressionList] TypeOp [TypeExpression]
PrefixOp	:: = ABS BASE LENGTH LONG MAX MIN
ExpressionList	:: = empty Expression ExpressionList, Expression
TypeOp	:: = SIZE
TypeExpression	:: = identifier TypelIdentifier TypeConstructor
TypelIdentifier	:: = BOOLEAN INTEGER CARDINAL WORD REAL CHARACTER STRING UNSPECIFIED PROC PROCEDURE SIGNAL ERROR identifier identifier identifier TypelIdentifier identifier . identifier identifier \$ identifier
TypeConstructor	:: = LONG TypeExpression @ TypeExpression POINTER TO TypeExpression

E.9 Commands summary

ASci

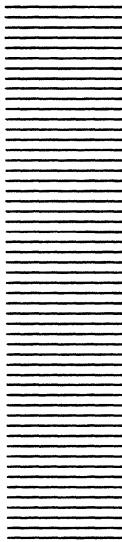
Read [address, count]
Display [address, count]

ATtach

Condition [number, condition]
Keystrokes [number, command]
Symbols [globalframe, filename]

BreakAllEntries [module/frame]Xits [module/frame]Entry [procedure]Xit [procedure]ClearAllBreaksEntries [module/frame]TracesXits [module/frame]Break [number]Condition [number]EntryBreak [procedure]Trace [procedure]Keystrokes [number]XitBreak [procedure]Trace [procedure]CUrrent contextDisplayBreak [number]ConfigurationEval-stackFrame [address] (g,j,l,n,p,q,r,s,v)GlobalFrameTableModule [module]Process [process] (l,n,p,q,r,s)Queue [identifier] (l,n,p,q,r,s)DisplayReadyList (l,n,p,q,r,s)Stack (g,j,l,n,p,q,r,s,v)Find variable [identifier]Kill session [confirm]ListBreaks [confirm]ConfigurationsProcessesOctalClear break [globalframe, bytepc]Read [address, number]Set break [globalframe, bytepc]Write [address, value]Proceed [confirm]Quit [confirm]

ReSet context [confirm]
SEt
 Configuration [*config*]
 Module context [*module/frame*]
 Octal context [*address*]
 Process context [*process*]
 Root configuration [*config*]
STart [*address*] [confirm]
Trace
 All
 Entries [*module/frame*]
 Xits [*module/frame*]
 Entry [*procedure*]
 Stack
 Xit [*procedure*]



DF Software Reference Manual

Outline

- 0. Introduction
- 1 Files
- 2. An overview of DF files and their use
- 3. User.cm
- 4. BringOver
- 5. Smodel
- 6. VerifyDF
- 7. DFDelete
- 8. DFSubstitute
- 9. DFDisk
- 10. DFTool
- 11. IncludeChecker and DF files
- 12. Dealing with Problems

Introduction

This document is based on Eric Schmidt's *DF Files Reference Manual*. It describes how to use the Klamath versions of the DF software. A companion document, *DF Release Tools Reference Manual*, describes the programs that are used by people who are responsible for doing software releases.

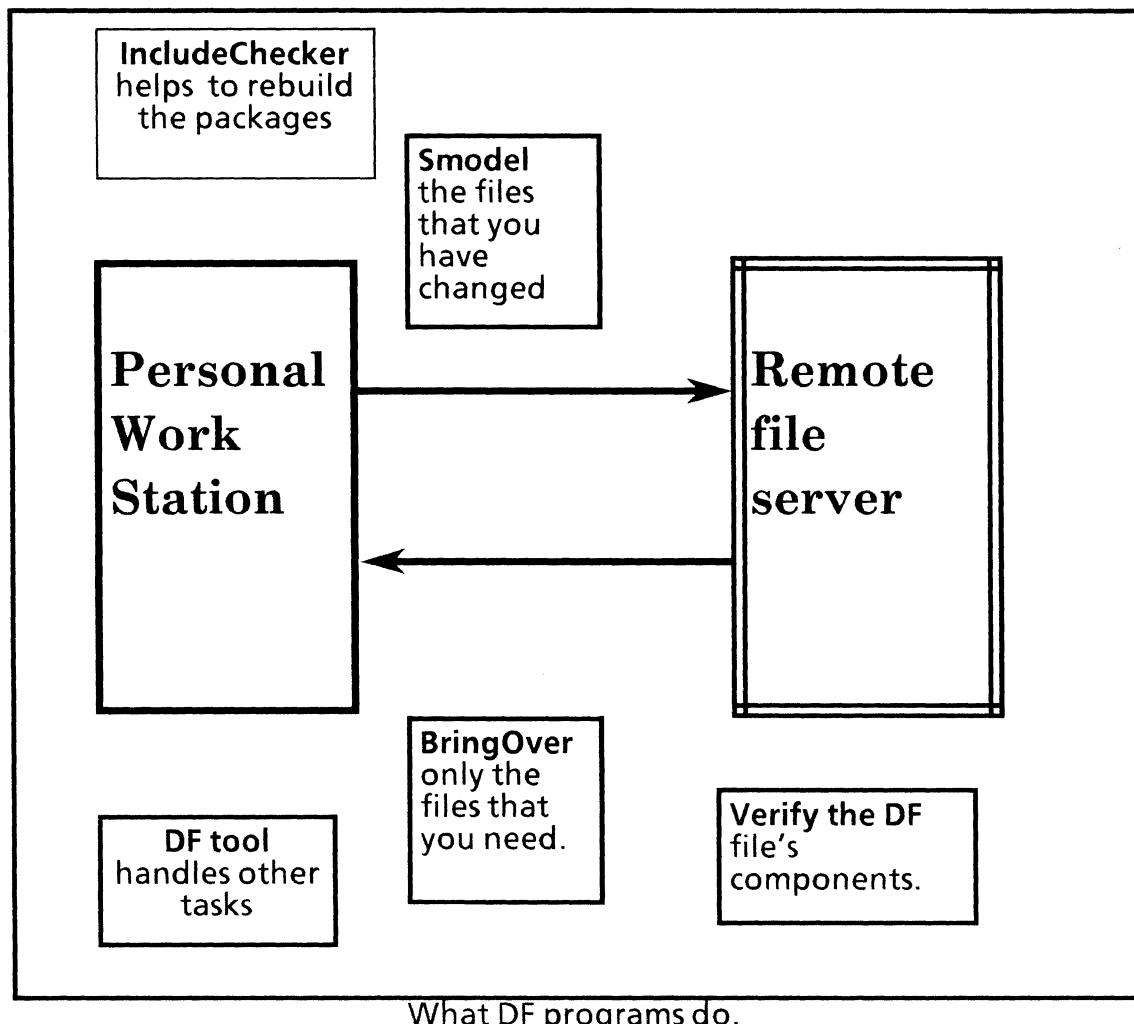
Why should one use the DF software?

The DF software helps the user keep track of files that you work on. These files may be program source and object files, or simple text files. Since it has the ability to describe the version and the location of files, the user needs to worry less about knowing where the files are located and knowing which versions of the files to use. A single DF file may describe all the required files in your program, thus you could simply use the name of the DF file to bring over all the files needed to work on a program from a remote file server to your local disk. After you modify and recompile some files, the DF program will store back only the files that were changed. This frees the user from remembering which files were changed. DF files may explicitly import and export files, in a manner similar to Mesa

programs or C/Mesa configurations. This allows careful sharing of programs between implementors. Having a software system described by a DF file allows one to use tools for managing files, verifying program consistency, and allowing programs be a part of a major software release. The DF software frees the user from bothering with the time consuming, yet important details of tracking program versions and locations. This manual introduces the user to the software and will also serve as a reference manual.

What are the DF programs?

The DF (Describe Files) programs comprise a general package for file management with explicit version control. These programs manipulate DF files, which are essentially lists of file names, fully qualified with remote location and create date. Each DF file typically corresponds to one software component.



Of the DF programs, these four are the most heavily used:

BringOver retrieves the files listed in a DF file from their remote file servers, possibly overwriting different versions already on the local disk. It

insures that all files for a component, and the correct versions of those files, are on the local disk.

SModel	stores changed versions of files back on remote file servers and produces a new DF file containing references to the newest versions. Normally, the new DF file is also stored remotely for use by clients of the component.
VerifyDF	checks that a DF file is complete and consistent. That is, that all files needed to build the top-level object files of a component are listed in the DF file and are consistent in the Mesa compiler and binder sense.
DFTool	provides a window interface to the other DF programs.

The Klamath IncludeChecker can also check DF files and generate command files to rebuild their packages. These capabilities, which are not described in the *Mesa User's Guide*, are discussed in section 11.

The DF files system was initially used by people running Mesa on shared Dorados, (a high performance personal computer) who wanted to guarantee they had the correct version of files they needed and as an easy way to save changed versions of file without unnecessary copying. It is now being used to partially automate the Klamath and Cedar release processes within Xerox.

1 Files

The DF programs are on the Klamath archive directory:

<APilot>11.0>DFFiles>Public>

and

The IncludeChecker is on the Klamath system test directory:

<AlphaMesa>11.0>.

If you are not in Xerox SDD, please refer to the release directories specific to your organization. If you using an earlier Mesa release such as Sierra (Mesa 10.0), please refer to an earlier version of this document.

2 An overview of DF files and their use

The DF file for a software component usually has three parts:

- A list of files exported by the component. These are interface or implementation files that are needed by clients; for example, `Space.bcd` and `Compiler.bcd`. DF file Exports (and Imports, described below) are analogous to Mesa module Exports (and Imports). A DF file normally exports itself; this self-reference causes SModel to store the DF file on a remote server whenever it changes.
- The component's implementation: the list of files that comprise the component but are of interest only to implementors (e.g., implementation modules).

- The imported files needed to build the component (e.g., **Environment.bcd** and **String.bcd** for many programs). These are usually public interfaces exported by another component.

2.1 An example DF file

Probably the easiest way to understand DF files is to consider an example. The following is a DF file for the Compare utility.

```
-- Compare.df  Last edited by Joe on 8-Feb-83 13:36:58
Exports [Igor]<Emerson>DF>  ReleaseAs [Idun]<APilot>DF>
    Compare.df                                22-Feb-83 13:59:40 PST

Exports [Igor]<Emerson>Compare>Public> ReleaseAs [Idun]<APilot>Compare>Public>
    + Compare.bcd!18                           22-Feb-83 13:53:16 PST
    Compare.symbols!7                          22-Feb-83 13:53:18 PST

Directory [Igor]<Emerson>Compare>Private> ReleaseAs [Idun]<APilot>Compare>Private>
    Compare.cm!2                             16-Nov-82 10:16:29 PST
    Compare.config!4                         16-Nov-82 10:21:06 PST
    CompareControl.bcd!13                    22-Feb-83 13:50:09 PST
    CompareControl.mesa!9                    22-Feb-83 13:49:53 PST
    CompareDefs.bcd!6                        16-Nov-83 11:48:21 PST
    CompareDefs.mesa!4                      16-Nov-83 10:16:47 PST
    CompareImpl.bcd!13                      18-Feb-83 14:55:08 PST
    CompareImpl.mesa!10                     18-Feb-83 14:55:00 PST
    CompareWindow.bcd!3                      23-Dec-83 13:52:41 PST
    CompareWindow.mesa!2                     23-Dec-83 13:51:53 PST

Imports [Igor]<Emerson>DF>ComSoftPublic.df Of #
    Using [Ascii.bcd, Format.bcd, Heap.bcd, String.bcd, Time.bcd]

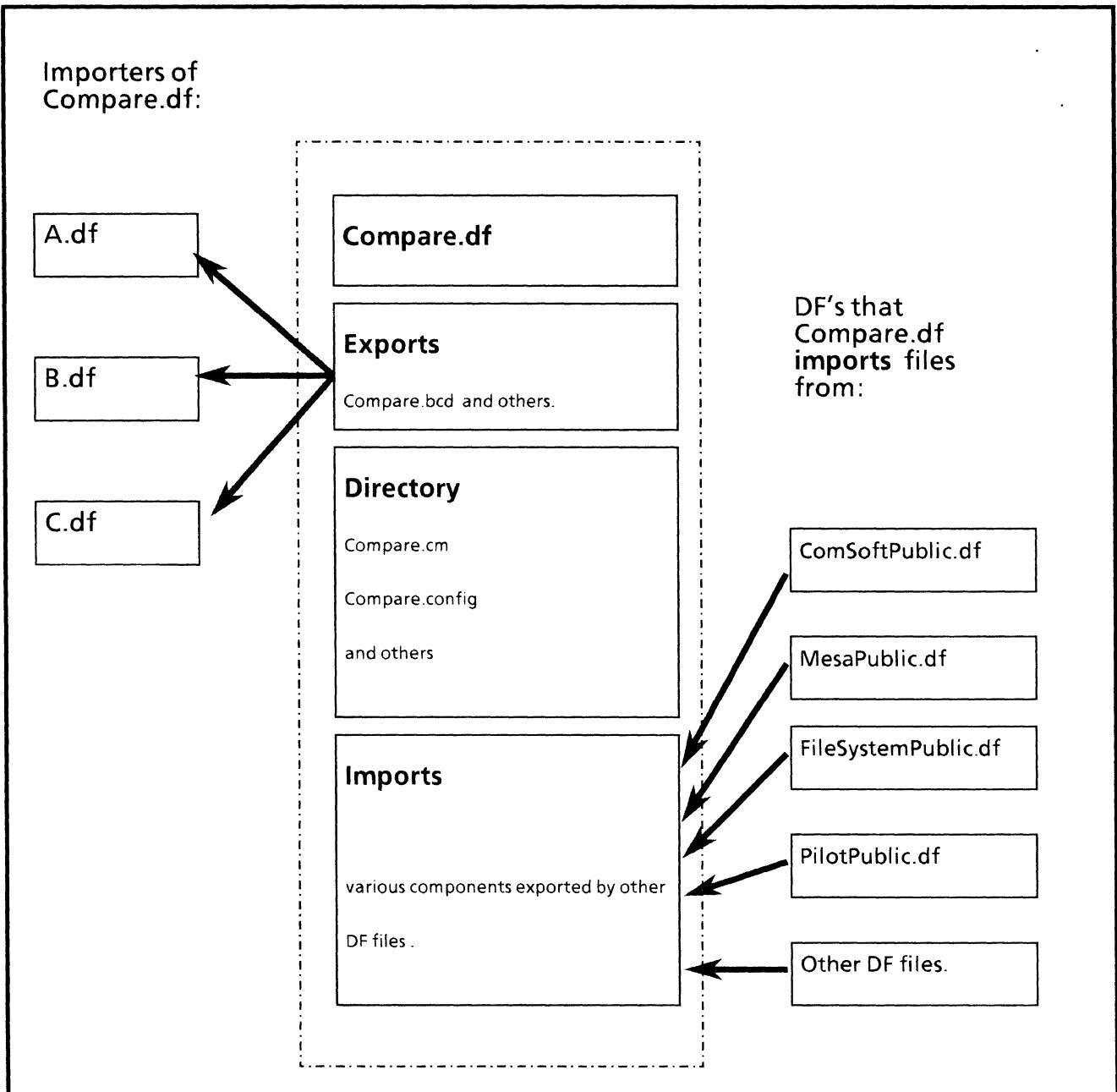
Imports [Igor]<Emerson>DF>MesaPublic.df Of #
    Using [Environment.bcd, Inline.bcd]

Imports [Igor]<Emerson>DF>FileSystemPublic.df Of #
    Using [MSegment.bcd, MStream.bcd]

Imports [Igor]<Emerson>DF>PilotPublic.df Of #
    Using [Process.bcd, Runtime.bcd, Stream.bcd, System.bcd, UserTerminal.bcd]

Imports [Igor]<Emerson>DF>TajoPublic.df Of #
    Using [Exec.bcd, FileName.bcd, FileTransfer.bcd, FormSW.bcd, Put.bcd,
          Tool.bcd, ToolWindow.bcd, UserInput.bcd, Version.bcd, Window.bcd]
```

The files exported by the Compare package (including the DF file itself) are marked with the keyword **Exports**, other files that are part of Compare are marked with **Directory**, and imported files have the keyword **Imports**. The ReleaseAs clauses are used to tell a program



A Df file with its contents and relationships with other DF files.

called the ReleaseTool where to store the files during a release. Only the files that are part of the component (i.e. not Imports) have ReleaseAs clauses.

Imported files, such as **Exec.bcd** and **MFile.bcd**, are retrieved when the DF files is brought over. However, since the DF file does not "own" them, they will not be stored by SModel. The Imports clauses in this DF file have explicit Using lists. If no Using list is given, all exported files in the Import DF file are assumed. Having a Using list is generally a good idea, since it documents which files are needed and speeds up **BringOver**. Note that imported files are gotten indirectly, by pointing to another DF file (the one for the component that exports those files). An importer doesn't need to know

anything about where the imported files are stored, or even their versions, since all that information is in the imported DF file.

A file is specified by a full path name (remote host and directory), an optional file server version number, and an optional creation date. The create date is used to uniquely determine the correct version of the file, while the version number is used as a hint to reduce the time needed to locate that correct version. If the create date is omitted, the highest remote version is assumed. In most cases, however, the create date has been filled in by the DF program.

An Extremely Important note: If you are storing your files on NS servers, please be sure to use the *fully qualified names*. For example, use:

[Tundra:OSBU North:Xerox] instead of
[Tundra:]

This is particularly important for files that are imported by other DF's, since the users may be dispersed across NS domains.

There are two special "create dates": ">" and "#". If the newer remote version of the file should always be brought over, ">" is used. The "#" specifies any remote version of the file that has a *different* create date than the version on the local disk. These two "create dates" support the *loose binding* of imports. If one imports FileSystemPublic.df of ">", Compare.df will always retrieve the MSegment.bcd and MStream.bcd described by the FileSystemPublic.df *most recently* Smodel'ed by its implementor. In general, files that are part of a component (i.e. Exports and Directory files) have explicit create dates.

We recommend the use of the "#" type of create dates for *imports* because it allows one to bringover old DF's for maintainance updates. If the ">" create date are used, you may accidentally use newer versions of imports which *happen to be* on your local disk. Furthermore, the "#" create will also bringover the correct version of files under normal development. One typically imports files from a release directory which has an unambiguous reference to a file, thus you would want the released version of the imported files, regardless of whether or not it is newer than the one on the local disk. The user should still consider the development practices in use and use the "right" mode. For example, you may not have an official release directory to import from, or perhaps you have some reason to avoid the use of older files.

The + in front of Compare.bcd indicates to the program VerifyDF that it is a top-level object file. A top level file may be of two types. First, it may be a .bcd or a .boot file. If so, VerifyDF will insure that all files needed to build Compare.bcd are listed in the DF file and are of the correct version. Otherwise it may be a file that is not a part of a component, such as documentation files. This prevents the program from doing unnecessary analysis.

Fine point:

some DF files also have files marked with "*". The * is ignored by all DF programs except for the ReleaseTool; it indicates files that must be copied onto [somehost]<archiveDirectory> or [anotherhost]<systemTestDirectory> after a release.

Blank lines in a DF file are ignored, and lines are treated as comments if they begin with "--" or "//".

2.2 A typical development scenario using DF files

DF files have little inherent structure or semantics. There is no requirement, for example, that the files they describe be consistent in the Mesa compiler sense (this allows DF files to be used to back up arbitrary files on personal workstations). However, DF files can provide considerable assistance for development if they are used in a stylized manner.

To illustrate the use of DF programs in program development, assume that you had to fix a problem with Compare. The steps you would take normally include the following:

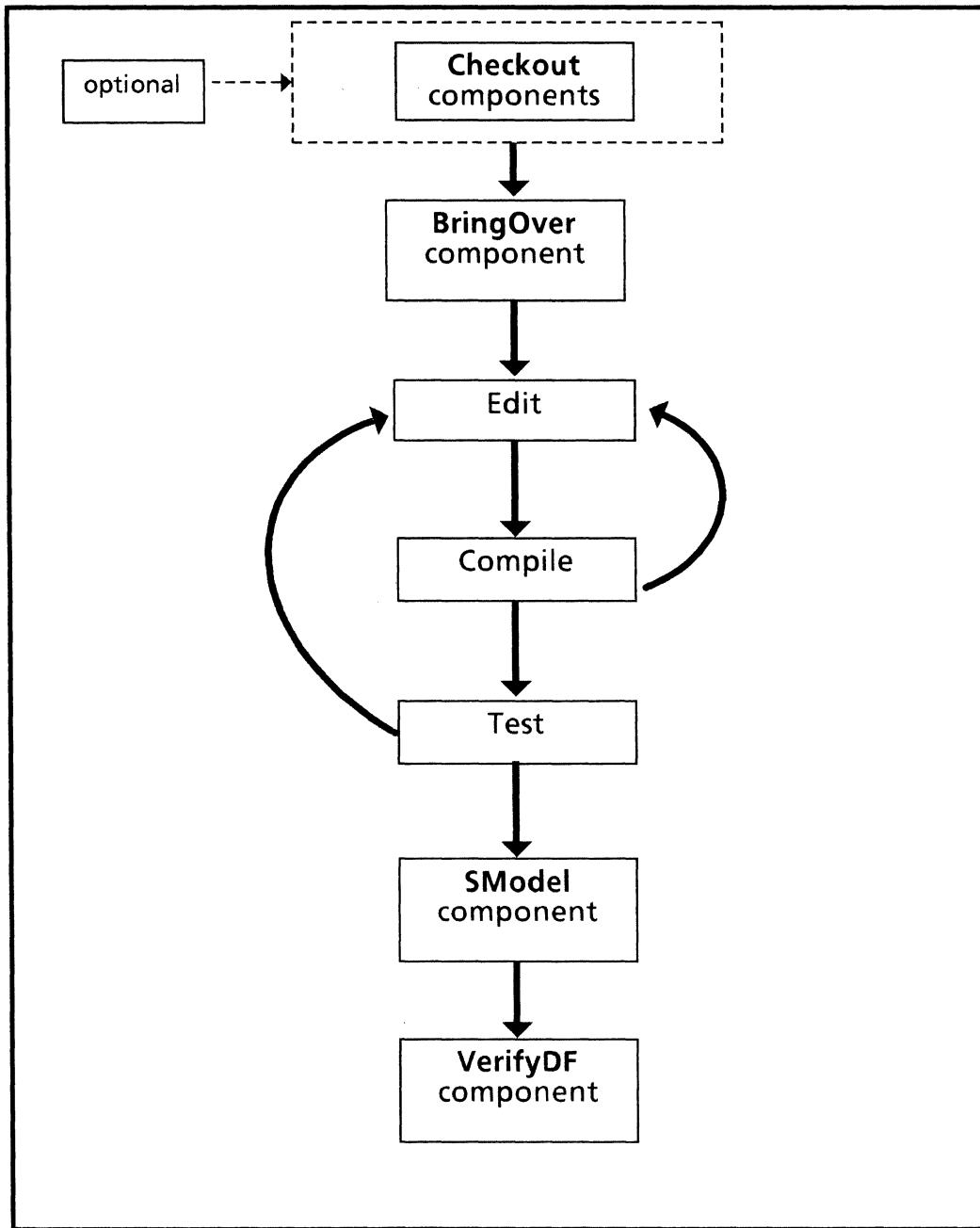
1. BringOver Compare.df. This insures that all files needed to build Compare, including imported files, are on the local disk in the correct version. BringOver will check to insure that you are using the most recent version of Compare.df.
2. Modify and test Compare. Since a DF file is just a text file with a fairly simple format, it can be edited whenever it is necessary, for example, to add new a module. Also, to assist in rebuilding its component, Compare.df, like many DF files, points to a command file (Compare.cm) that can be used to build the component from scratch; text can be selected from this command file and stuffed into the executive. It is also possible to run the IncludeChecker on a DF file to generate a command file for its reconstruction.
3. SModel Compare.df. This stores back changed files and updates Compare.df to reflect the new versions. In general, you do not have to think about what files have changed, you can just SModel the component.
4. VerifyDF Compare.df. This verifies that Compare is complete and consistent. At this point, you can let users know about the new version of Compare.

2.3 Releases and the use of file server directories

In using the DF software, three directories are of special importance. These are the *working*, *integration*, and *archive* directories.

Each group of software developers has a separate working directory that holds the latest versions of their software. For example, this directory is [Rasp:OSBU North:Xerox]<Emerson> for the Mesa group. The Directory and Exports clauses in the group's DF files point to the working directory, and that directory is the source and target of most BringOver and SModel runs. Experience has shown that it is useful to set aside a subdirectory of the working directory, e.g. [Rasp:OSBU North:Xerox]<Emerson>DF>, as the location for the group's "working" DF files. This simplifies finding DF files and allows BringOver (when the `DefaultDFLoc:` entry of the `User.cm` is set) to insure that only the most recent versions of DF files are used.

The integration directory is shared by development groups. A component is stored onto the integration directory when it has been tested and verified (using VerifyDF), and its developers believe that it is ready for use by other groups. Components are stored onto the integration directory by using SModel's prerelease mode. The integration directory is used to communicate software between groups. A development group should only obtain (Import) software from another group that has been stored onto the integration directory;



Typical development cycle.

it should never use software or DF files from the private working directory of another group. The integration directory for the Mesa group and all of System Software, for example, is [Idun]<Int>. The integration directory also serves as a staging area for a release.

A *release* is a set of compatible software components that have been saved in a safe location. The ReleaseTool verifies that a set of DFfiles is globally consistent and complete, copies the files to the release directory, and generates new DF files that describe the release. The new DF files are fully bound: all Imports (and Includes, which are discussed in section 2.7) are specified with explicit create dates (there are no > or #'s). Only the

ReleaseTool stores files onto the release directory. The release directory is named by ReleaseAs clauses in each DF file. For example, the release directory for Klamath is [Idun]<APilot>. Other directories may follow as required by the user organization as shown in the diagram below.

Fine points:

It is possible to override the release location with the ReleaseTool.

The new DF files generated by the ReleaseTool also have *CameFrom* clauses in place of the original ReleaseAs clauses. A CameFrom clause for a file documents the location on the prerelease directory from which the file was copied. For example, the ReleaseTool will change

```
Exports [Idun]<Int>Compiler>Public>    ReleaseAs [Idun]<APilot>Compiler>Public>
Compiler.bcd!27                                29-Feb-83 11:16:37 PST
```

to be

```
Exports [Idun]<APilot>Compiler>Public>    CameFrom [Idun]<Int>Compiler>Public>
Compiler.bcd!1                                 29-Feb-83 11:16:37 PST
```

2.4 Creating a new DF file from scratch

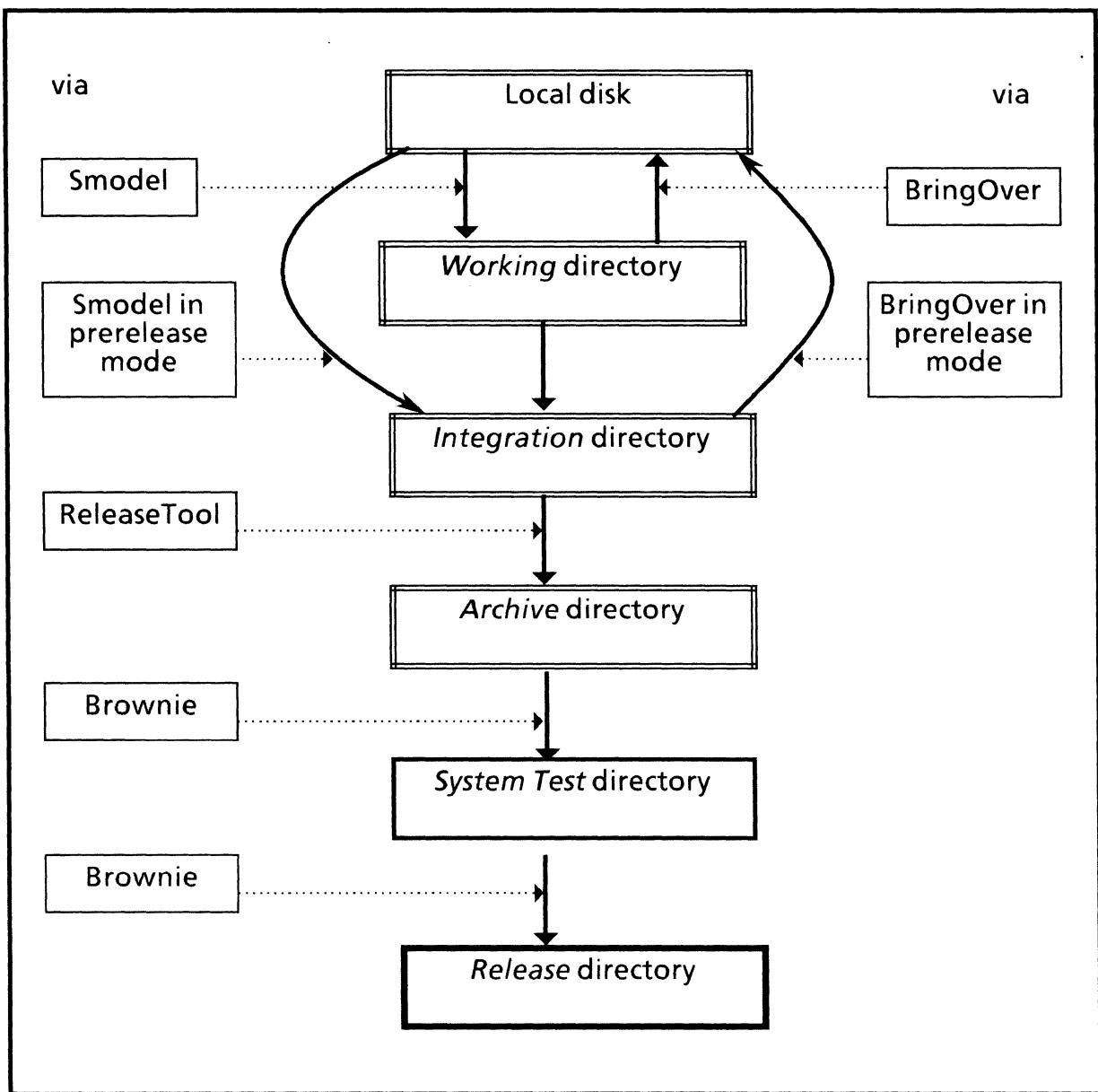
The easiest way to generate a DF file for a component is to do the following:

1. Get all of the component's files onto the local disk. Make sure that the local versions are the same as those on the remote file server (to keep from confusing yourself or the DF software).
2. Compose a skeleton DF file for the package that lists its files under the appropriate Directory and Exports lines. It is not necessary to fill in create dates since this will be done by SModel in step 3. Make sure that the remote locations and ReleaseAs locations are correct. Add any Imports that you can think of.
3. Run SModel on the skeleton DF file with the /n (don't store files remotely) switch. This will rewrite the DF file with the create dates filled in (and **will not store any files**).
4. Use VerifyDF to check the DF file. It will report any missing files, or files that have the wrong versions. Correct the DF file as necessary and repeat this step. The Find utility is often useful for locating the DF file that describes a needed import. The Find utility may be used to search over a number of DF files for the one that contains the files that you need. For example:

```
> find System.bcd [Host]<Directory>DF>*.df
```

will search over the DF files in the specified directory for the one that contains System.bcd. Ignore the DF files that Imports System.bcd, and look for the one that Exports it. The Exporting DF file would contain the create date, while the Importers would include System.bcd in the Using list.

fine point: If you are importing a file that is a part of a software release managed by DF software, you may be able to use a utility called DFfetch. This program is still not part of the general release. This program will query a database using a file name as a key and will return the name of the DF file that contains it.



Releases and the use of file server directories.

5. When VerifyDF no longer complains, run SModel to store the DF file itself and the files it describes remotely. Since you are running SModel for the first time, use the /v switch to make the program verify that your files exist in the destination and store it there as necessary.

2.5 DF files and libjects

If more than one person is responsible for a software component, it is important to prevent simultaneous modification of both the individual files of the component *and* its DF file. This is because the DF file points to specific versions of the component's files. To deal with

this problem, each DF file should have a program librarian libject. There are libjects for each Klamath DF file maintained by the Mesa group.

When a component is to be worked on, its DF file is first checked out, typically using the "Access" program. After the component has been changed and tested, it is SModel'ed, which will check in the libject for each changed file, including the DF file itself. If another person attempts to work on the package at the same time, he will be unable to since the libject is already checked out. Typically, one checks out a libject for only the DF file.

If this methodology appears to be too restrictive for some large component, there are two possibilities: 1) break up the component into smaller pieces, each with its own DF file and libject, or 2) adopt a more complicated checkout and checkin scheme. The Mesa group's experience has shown that it is much simpler and less error prone to break up the DF file. If the component shouldn't be broken up, and it is necessary for more than one person to be modifying (different portions of) it at the same time, the following methodology can be followed by each maintainer:

1. BringOver the component's DF file. Do not check out the DF file at this time. Do check out libjects for component files that you will be changing.
2. After modifying and testing the component, but just before SModeling it, check out the DF file. Successfully checking out the DF file means that you currently have the right to change the description of the "truth" on the shared remote directory (i.e. the DF file).
3. Now BringOver the DF file and its components *again*. This will use the most recent DF file, which might be newer than the one you originally brought over (if other people were modifying the component simultaneously). BringOver might retrieve newer versions of files that *others* changed. If so, rebuild and retest your version of the component. You do not have to run BringOver again since you "hold the lock" on storing new versions of the component's files.
4. SModel the DF file to store your changed files and the new DF file, and to release your "lock" on SModeling the DF file.

2.6 Use of IFSs and NS file servers

The DF software is able to retrieve and store files on both NS file servers and the PUP-based IFSs. To use a product file server, simply give its fully qualified clearinghouse name. For example,

```
Imports [Rasp:OSBU North:Xerox]<WComm>DF>RS232CPublic.df Of >
Using [RS232CIO.bcd]
```

2.7 Included DF files

Although a component is usually described by a single DF file, there are a few particularly large or complicated components that are more easily described by a set of DF files. An example is the Pilot kernel, which has so many files that it is convenient to have separate DF files for each major subconfiguration and for the public-, friends-, and private-level interfaces. Such a collection of DF files must have a "root" DF file that (directly or indirectly) *includes* the others. This is done with the **Includes** construct, which resembles

the Imports clause described above. For example, a fragment of the root DF file for the Pilot kernel, Pilot.df, is:

```
--Pilot.df Last edited by Jimmy on 3-Jan-83 21:25:59
Exports [Igor]<Emerson>DF> ReleaseAs [Idun]<APilot>DF>
Pilot.df 3-Jan-83 21:27:32 PST

-- Pilot kernel defs
Includes [Idun]<P>DF>PilotFriends.df Of > ReleaseAs [Idun]<APilot>DF>
Includes [Idun]<P>DF>PilotPrivate.df Of > ReleaseAs [Idun]<APilot>DF>
Includes [Idun]<P>DF>PilotPublic.df Of > ReleaseAs [Idun]<APilot>DF>

-- Pilot kernel subconfigurations
Includes [Idun]<P>DF>Control.df Of > ReleaseAs [Idun]<APilot>DF>
Includes [Idun]<P>DF>FileMgr.df Of > ReleaseAs [Idun]<APilot>DF>
Includes [Idun]<P>DF>Filer.df Of > ReleaseAs [Idun]<APilot>DF>
Includes [Idun]<P>DF>Swapper.df Of > ReleaseAs [Idun]<APilot>DF>
```

Includes is treated as macro substitution: the effect is to replace the Includes clause with the entire contents of the included DF file. Whenever one of the DF programs such as BringOver is run on the root DF file, it is applied recursively to the included DF files.

Note: There is a significant difference between Includes and Imports. The Imports clause is used when files are needed, but they are “owned” by another DF file. Although imported files are retrieved by BringOver, the DF programs do not otherwise recur on imported DF files. SModel, for example, will recursively store *included* DF files but not *imported* DF files.

2.8 ReadOnly files

If your component depends upon some files in a remote directory, but those files are not “owned” (described by) a DF file, you can’t just Import them. However, you can document your component’s dependence on those files, and have BringOver retrieve them when your DF file is brought over, by listing the files in your DF file and marking their directory ReadOnly. One needs this when you are importing components from implementors who are not using DF files. This practice should be discontinued once the implementors use DF files.

Here is an example,

```
ReadOnly Directory [Iris]<Smith>BTree>
BTree.bcd 12-Jan-83 10:17:22 PST
BTree.mesa 9-Jan-83 21:27:32 PST
BTreeImpl.bcd 17-Jan-83 15:56:19 PST
BTreeImpl.mesa 10-Jan-83 11:25:49 PST
```

(The keyword **Directory** after **ReadOnly** is optional). Readonly files are never stored by SModel. Since they are not owned by your DF file, they do not have a ReleaseAs clause to indicate where they are to be stored on a release.

The DF software `User.cm` section is called **[DFTool]**. The following is a list of the `User.cm` fields used by the DF programs:

- WorkingDFLoc:** the remote working directory, e.g. [Rasp:OSBU North:Xerox]<Emerson>DF>, that holds your group's "working" DF files. It is used by BringOver when you retrieve a component. If the component's DF file is not local, BringOver will retrieve it from the **WorkingDFLoc:**. If the DF file is on the local disk, BringOver will check that it is at least as new as the version on the **WorkingDFLoc:**. By default, the **WorkingDFLoc:** is empty, and BringOver does no checking.
- IntegrationLoc:** your group's prerelease location, e.g. [Idun]<Int>. By default, this entry is empty, and BringOver and SModel prerelease mode cannot be used.
- CheckLibrarian:** if **TRUE**, SModel will check, before storing each file, if it has a libject. If it doesn't, SModel simply stores the file. If it does have a libject, there are three possibilities: 1) If the file wasn't checked out, SModel won't store it. 2) If it was checked out, but not by you, SModel won't store it. 3) If it was checked out by you, SModel checks it back in and stores the file. Also, if **CheckLibrarian:** is **TRUE**, SModel's prerelease mode will warn you if any of the files that you are submitting to a release (storing onto the prerelease directory) have libjects checked out. The default value for **CheckLibrarian:** is **FALSE**.
- LocalDFDir:** the directory on your local disk, e.g. <>DF>, to which BringOver and VerifyDF will retrieve new DF files. Setting this entry helps to prevent DF files from being scattered all over your disk. The local DF directory should always be on your search path, since the DF software always looks files up (anywhere) on the search path. The default for **LocalDFDir:** is empty, and DF files are retrieved to the directory on the front of the search path.

fine point: The DF programs check that the LocalDFDir and LocalDir (see DFTool) are on the search path and a warning is given if they are not. For example, if they are not given in the search path, the **wrong** DF file may be used for a BringOver.

4 BringOver

BringOver runs in the Executive and takes commands from the command line. In the simplest case, to retrieve a DF file and its components, just type

```
>BringOver [Host]<Directory>DF>DFfile
```

BringOver works as follows: It reads the DF file one line at a time. It takes the remote file name listed in that line, strips off the directory information and looks to see if it is on the local disk. One of three things can happen:

- If the file is not on the local disk, BringOver will offer to retrieve it.
- If the file is on the local disk, BringOver looks at the version on the local disk. If the create date listed in the DF file differs from the create date of the local file, BringOver will try to retrieve the remote version. If this would retrieve an *older* version of the file over a newer version, BringOver will first ask for confirmation. This helps to support a “newer is usually better” file management methodology.
- If the create date is omitted from the DF file, BringOver will *always* try to retrieve the file. Again, if this would retrieve an *older* version of the file over a newer version, BringOver first asks for confirmation.

If you omit the file server version number (e.g. “!3”), BringOver will enumerate all the versions of that particular file looking for one with the correct create time. If there are no versions of the file you list in the DF file on the remote host in the directory you specify, BringOver will give you a warning message. If there are files with the same name and none of the create dates available match that listed in the DF file, BringOver will give you a warning and offer to retrieve the latest version.

After running BringOver you can be sure the files listed in the DF file are on your local disk, and that their create dates agree with the create dates listed in the DF file, or BringOver will have printed out error messages.

Normally BringOver will list each file to be retrieved and will ask for confirmation. (You may reply “y” or CR to confirm, “n” to skip retrieval of this file, “q” to stop BringOver altogether, and “a” to retrieve this file and subsequent files as if “y” were typed each time.) The /a switch can be given on BringOver’s command line to suppress (most) requests for confirmation:

```
>BringOver /a [Host]<Directory>DF>DFfile
```

If you use the /a switch or reply “a”, and an older version of a file would be retrieved over a newer one, BringOver will always stop and ask for explicit confirmation.

BringOver can read a local DF file as easily as a remote one:

```
>BringOver Compare.df
```

You will use a local copy of the DF file when you have done previous BringOver’s and Smodel’s. If so, you will have a local copy of the DF file that is identical to the remote one. That is because Smodel will modify the DF file and store it remotely, leaving a copy on your local disk.

As files are brought over, a property (called the *RemoteName* property) is added to their leader page recording of the retrieved file so that the Mesa Development Environment knows where the file came from. (FTP and the FileTool also set this property.) These remote locations can be printed out by the DFDisk program (described below in Section 9).

If the create date entry is a > rather than a normal date, BringOver will retrieve the file only if the version on the remote server is newer than the version on the local disk. If the file is not on the local disk, it will be retrieved. As an example,

```
BTree.mesa >
```

will retrieve BTree.mesa from the remote server only if there is a newer version on the server or no local copies exist.

Similarly, if the create date entry is a #, BringOver will retrieve the file only if the highest version on the remote server is different than the version on the local disk. If the file is not on the local disk, it will be retrieved. For example,

```
BTree.mesa #
```

An Includes clause, e.g.

```
Include [host]<path>Component.df Of <date>
```

will cause BringOver to invoke itself on Component.df at the point it encounters the Include statement. If the included file itself has an Include statement, BringOver will again invoke itself on the inner DF file, and so on, in a recursive fashion. Furthermore, the DF file itself is retrieved using the usual BringOver rules before the recursive call.

An Imports statement

```
Imports [host]<path>Package.df Of <date>
```

will cause BringOver to 1) retrieve Package.df to the local disk if necessary and 2) examine all exported files in Package.df and retrieve them if necessary. Of course Package.df may have Include or Imports statements, so this is a recursive algorithm.

Appending a Using clause to the Imports statement, analogous to the Mesa language construct, gives the user explicit control over the files to be retrieved. The Using list may be used to obtain files that are under both Exports and Directory headings; That is, files can be obtained with the Using clause whether they are exported or not. Although use of the Using clause is not required, it is strongly recommended.

```
Imports [host]<path>Package.df Of <date>
Using [list of files, separated by commas]
```

Examples of Imports:

```
Imports [Igor]<Emerson>DF>TajoFriends.df Of >
```

```
Imports [Idun]<APilot>DF>CoPilot.df Of 24-Feb-83 11:14:26 PDT
Using [CPSwapDefs.Bcd, CPSwap2.Bcd]
```

The files referred to by an Imports statement may themselves be exported by preceding the keyword **Imports** by **Exports**. This is useful when users of your package need to have files from some other package in order to, for example, compile their system.

4.1 BringOver modes

There are three special modes in which you can run BringOver.

4.1.1 BringOver only specified files mode

The switch /o will instruct BringOver to retrieve only the files listed on the command line after the /o. The DF file to be used is given last. This mode is often used when you are not working on a package, but you simply need some files that are described by its DF file. For example,

```
>BringOver /o MFile.bcd MStream.bcd MSegment.bcd FileSystemPublic.df
```

will examine and potentially retrieve only **MFile.bcd**, **MStream.bcd** and **MSegment.bcd** in **FileSystemPublic.df**.

4.1.2 BringOver "verify files exist" mode

The switch /v will cause BringOver to run in *verify files exist* mode, where it will check that the files listed in the DF file actually exist on the remote servers or the local disk. No files are retrieved in this mode. BringOver will inform the user if newer versions were found, and if so will write a new DF file listing the newer versions. Also, if any files were listed in the DF file without their file server version numbers (e.g. !5), BringOver will write a new DF file with those version numbers filled in. This mode is often used to "flesh out" a skeleton DF file with the correct create dates and file server version numbers. If the file is listed correctly and a local copy exists, BringOver will add the **RemoteName** property to its leader page. Note for large DF files the verify option takes a few minutes.

4.1.3 BringOver prerelease mode

BringOver's prerelease mode is useful for fixing an old version of a package that was submitted to a release. This mode is entered with the /z switch. It brings over the DF file for the package that is on the remote integration directory (named by the **IntegrationLoc:** entry in User.cm). Since this might overwrite newer versions on the local disk, BringOver asks for confirmation before doing any retrievals.

4.2 BringOver's command line

In general, the command line for BringOver has the form

```
>BringOver [/<global switches>] DFfile1[/<local switches>] ... DFfilen[/<local switches>]
```

The optional global switches control the retrieval of the following DF files. You can also set global switches by giving an empty DF file name.

BringOver's "only file" mode (/o) has a slightly different format: the files to be retrieved are listed after the global switch /o, and the DF file to be used is named last.

BringOver also recognizes commands, **localDir/c**, **localDFDir/c**, **WorkingDFLoc/c**, and **IntegrationLoc/c** that specify subdirectories for file retrieval and storage.

The command **localDir/c** gives the directory for looking up and retrieving files. For example, the command line

```
>BringOver localDir/c <>MyPackage> MyPackage.df
```

will retrieve MyPackage's files to the directory `<>MyPackage>` on the system volume.

The command **localDFDir/c** names the directory to which DF files themselves (not their contained files) should be retrieved. It overrides any **LocalDFDir:** entry in User.cm. If both **localDFDir/c** and **localDir/c** are specified, the local DF directory is used for DF files; all other files use the localDir/c directory. For those rare occasions when you don't want a package's DF files to go to the User.cm-specified local DF directory (e.g. if you're fixing an old version of a package, perhaps one submitted to a release), you can use the **localDFDir/c** command to force the DF files to go to another directory. For example:

```
>BringOver localDir/c <>Old> localDFDir/c <>Old> [Idun]<Int>Stuff.df
```

The command **WorkingDFLoc /c** and **IntegrationLoc/c** overrides any WorkingDFLoc: and IntegrationLoc : entry in User.cm. For example:

```
>BringOver WorkingDFLoc/c [Idun]<P> IntegrationLoc/c [Idun]<Int> MyStuff.df/z
```

It helps developers that have more than one working directory; e.g. those doing both microcode and Pilot development.

4.2.1 BringOver switches

A switch specification is a letter identifying the switch, optionally preceded by a '`-`' or '`~`' to reverse the sense of that switch.

fine point: If you are using TTYTajo , please use the '`-`' rather than the '`~`'.

The valid switches are:

- a** always retrieve without confirmation (unless an existing local file is newer)
- b** get just "bcd" (derived) files: .bcd, .symbols, .boot, .signals, .press files
- f** force retrieval of all files, disregarding any newer local files.
- o** get only specified files: e.g. BringOver /o Exec.bcd Put.bcd Tajo.df
- p** get just public (exported) files
- r** get just readonly files: Imported (and ReadOnly) files
- s** get just "source" files (inverse of /b)
- u** only update existing local files (never get new files)
- v** verify files exist in the right place and version, and fill in DF dates
- w** get just "writable" (Exports or Directory) files (inverse of /r)
- x** rename ".bcd" to ".archiveBcd" if an archiveBcd already exists
- z** prerelease mode
- <** suppress confirmation request if an *older* remote version is retrieved

The default setting for all switches is *off*. You can also change the default setting of any switch by using a global switch. Any switch given with no file name (i.e., just a slash and

switches) establishes the default setting for that switch. Unless overridden or reset, that default applies to all subsequent commands.

4.3 BringOver limitations

Each DF file read by BringOver must contain no more than 450 files. This applies to each Imported and Included DF file as well.

5 SModel

SModel (the name stands for "Simple Modeller") is used to store back new versions of files you've changed since the last time you ran BringOver on a DF file. For example, if you are working on the Compare program and you've already run BringOver on Compare.df, then

```
>SModel Compare.df
```

does the following: The files listed in Compare.df are checked on the local disk. If any have different create dates SModel will offer to store them on the remote servers specified in Compare.df. SModel then produces a new Compare.df file with the new create dates and remote file system version numbers (e.g. !4). The old DF file is saved by copying it to a "\$" file, e.g. Compare.df\$. Files listed under an "Imports" or a "Readonly" clause will be ignored.

If a file on the local disk is listed without any create date in the DF file, SModel will fill in the create date from the version on the local disk and then offer to store the file remotely. If the file listed in the DF file is followed by a > or #, it is ignored and will not be transferred.

If the DF file contains a reference to itself (e.g. Compare.df lists Compare.df), SModel will also store a new version of the DF file on the remote server. Since SModel has to write out a new DF file before it can store the DF file, SModel cannot put the file server version number (e.g. !5) on the DF file self-reference. However, BringOver will always get the correct version of the DF file since it will use the create date of the DF file instead.

Before storing a file, SModel first checks to see if that file is already on the remote directory (as the highest version); if so, it won't actually store the file. Also, if storing a file would write a version with a create date that is *older* than that of the current highest version, SModel will always ask for confirmation. This helps to support the "newer is usually better" methodology.

If the **CheckLibrarian:** entry in the User.cm is **TRUE**, then before storing a file, SModel will check if the file has a program librarian libject. If so, SModel won't store the file if it wasn't checked out, or if it wasn't checked out by you. (The old create date is left in the DF file, so that in most cases, you can simply check out the file [without retrieving the source] and rerun SModel.) One must be careful when doing this since someone else may have checked out the DF file and Smodel'ed already.

SModel invokes itself recursively on Included DF files. It does *not* invoke itself on Imported or ReadOnly DF files. If the Includes or Imports statement is not followed by an "Of <date>" clause, SModel will insert such a clause in the new DF file with <date> replaced by the create date of the file on the local disk.

5.1 SModel modes

There are three special modes in which you can run SModel.

5.1.1 SModel "verify files are remote" mode

New DF software users are often confused about the relationship of the entries in a DF file to the local and remote directories, and what SModel will do in certain cases. The easiest way to understand it is that SModel assumes 1) that the DF file was an accurate description of the remote directory at some point in the past, and 2) files with different create dates that it finds on the local disk are the "truth" and should be transferred. **This is one of the most important things to know. If you are having trouble with your DF files, always remember that the DF file describes the state of a remote directory.** However, assumption 1) allows SModel to assume that files with the same create date in the DF file and on the local disk *also* exist on the remote server. For this reason, and because remote enumerations are relatively slow, SModel does not check the remote server to see if in fact the files described by a DF file are actually there (unless it has already decided to store a file). So, SModel may not detect that certain files listed in a DF file are not present on a remote server, unless you use a special switch described below.

A common mistake is to assume that if you run SModel on a DF file successfully, and then simply change a Directory in the DF file, then all the files will be copied (again) to the new directory. This is *wrong!* After SModel has been run the first time, the create dates in the DF file are the same as those on the local disk. Since SModel just checks the create dates in the DF file against the local files, the second SModel invocation will not detect that any files need be transferred even though the Directory was changed.

To resolve these problems, SModel has a *verify files are remote* mode which is entered with the /v switch. In this mode, SModel not only applies the algorithm described above to store files, but if it decides a file doesn't need to be stored, it will look on the remote file server and check that in fact the file does not need to be stored. If the file is not on the remote directory, or the version listed in the DF file is not on the remote directory, then SModel will offer to store the file. In this way SModel /v will try to force the remote directory to agree with the DF file.

For example, the following will insure that all files listed in Compare.df are actually on the remote servers:

```
>SModel Compare.df/v
```

5.1.2 SModel "don't store files" mode

The /n switch has SModel do everything it normally does, except for storing files. The DF file is rewritten if there are different versions of files on the local disk, but those files, and the DF file itself, are not stored. This mode is useful for "fleshing out" a DF file with the versions of files that are on the local disk. **This is a dangerous thing to do. The DF file itself simply describes the version and locations of the files listed. If you use the "don't store files" mode, the Df file will be changed to include the create dates of the files on your local disk. Subsequent uses of the DF file will look for those files on the remote server, but they will not exist. Make sure you Smodel again using the "verify files are remote" mode to actually store the files remotely.**

5.1.3 SModel prerelease mode

In this mode, SModel stores a component on your prerelease directory (which is named by the **IntegrationLoc:** entry in User.cm). The actual remote directory for each file is gotten by concatenating the **IntegrationLoc:** directory with the *ReleaseAs subdirectory* for the file (Although this sounds strange, that is the correct location. The Exports or Directory subdirectory, for example, might point off to a temporary or personal directory.)

For example: the Integration directory of: [Idun]<Int> and the ReleaseAs directory of [Idun]<APilot>MyProg>Public will yield: [Idun]<Int>MyProg>Public.

SModel recurs on Included DF files and stores them out as well. Imports clauses are changed to point to the prerelease directory and SModel checks that the imported DF files already exist there. If the User.cm entry **CheckLibrarian:** is **TRUE**, SModel also checks to see if each file has a libject that is checked out; if so, it gives a warning.

5.2 SModel's command line

The SModel command line has the form

```
>SModel [/<global switches>] DFfile1[/<local switches>] ... DFfilen/[<local switches>]
```

Global switches are optional and control the store of subsequent DF files. You can also set global switches by giving an empty DF file name.

The subcommands **WorkingDFLoc/c** and **IntegrationLoc/c** work identically to the same commands in BringOver.

Secondary connect credentials can be given on the command line; e.g.

```
>SModel Conn/c Dir Passwd MyComponent.df/z
```

5.2.1 SModel switches

A switch specification is a letter, optionally preceded by a '-' or '~' to reverse the sense of that switch. The switches recognized by SModel are:

- a store always: without confirmation
- f flip CameFrom clause to be ReleaseAs (*default*)
- l check with program librarian (*default*) It overrides User.cm
- n do not store files remotely
- r ignore ReadOnly or Imports designation and store files if different versions
- t process only top (outermost) DF file, not Included DF files
- v verify that files are really on the remote server and store if necessary
- z prerelease mode

The default setting for the /f and /l switches is *on*; all other switches are *off*. You can change the default setting of any switch by using a global switch. Any switch given with no file name (i.e., just a slash and switches) establishes the default setting for that switch. Unless overridden, that default applies to following commands.

Fine point: When the /f (flip CameFrom) switch is on, SModel will convert a CameFrom clause back to a ReleaseAs clause. This makes it easier to use a DF file that was generated

by the ReleaseTool as a starting point for a new “working” DF file after a release. This switch generally has only a minor effect on the use of DF files.

5.2 SModel limitations

Each DF file processed by SModel must contain no more than 450 files. This applies to each Included DF file as well. This number may change. Contact the implementors if there is a question.

6 VerifyDF

VerifyDF attempts to answer the question: Does this DF file have entries for all the files I need to rebuild my program, and are these files consistent? VerifyDF scans a DF file looking for “end result” bcd and boot files. These are the files marked with a “+” before their names (a DF file can have more than one “end result”). VerifyDF will analyze **each** of these files to determine what files were needed to build it and will compare the needed files against entries in the DF file. If a needed file is not in the DF file, VerifyDF will give an error message. Also, VerifyDF will give an error message if a needed file is listed in a different version.

After checking the “end result” files, VerifyDF recursively analyzes the files they need. This process continues until all files in the closure of dependencies, except for imported (and missing) files, have been analyzed.

For example, to verify Compare.df, type

```
>VerifyDF Compare.df
```

Any files that are missing from the DF file are listed with the create dates and remote location (gotten from the RemoteName leader page property) of files on the local disk. This can help, for example, to identify DF files from which some of those files should be imported. VerifyDF also prints out files listed in the DF file that appear to be unnecessary. These might include such files as command files and signals listings, but they might also include imports that are no longer necessary. If those files are actually necessary, such as the command files, you can suppress these warnings by marking these files with a ‘+’. for example:

```
Directory [MyHost]<MyDir>Private          ReleaseAs [ReleseHost]<RelDir>Private>
+ Source.MyStuff!1    14-Dec-60 16:23:27 PDT
+ ReBuildMyStuff.cm!1 14-Dec-60 16:25:53 PDT
```

VerifyDF also checks for certain common mistakes, such as files on a directory that are released onto a directory with a different >Public, >Friends, or >Private suffix. For example,

```
Directory [Igor]<Emerson>Compare>Public ReleaseAs [Idun]<Apilot>Compare>Private
```

is probably a mistake, since Public is not the same as Private.

VerifyDF will look on remote file servers for the correct versions of files if they are not local. So, the files described by a DF file do not have to be on the local disk for VerifyDF to do its job. However, since this remote checking must currently be done with a pseudo page-level access protocol, it can be relatively slow. The DF file itself also does not have to be on the local disk. For example, the following can be used to check a remote version of Compare.df:

```
>VerifyDF [Igor]<Emerson>DF>Compare.df
```

When processing a DF file, VerifyDF may have to retrieve imported or included DF files from a remote server. Unless the /t (fetch to temporary files) switch is off, these DF files will be retrieved to temporary files. This avoids cluttering your disk with DF files you may not want.

6.1 VerifyDF's command line

The VerifyDF command line has the form

```
>VerifyDF [/<global switches>] DFfile1[/<local switches>] ... DFfilen[/<local switches>]
```

Global switches are optional and control the verification of subsequent DF files. You can also set global switches by giving an empty DF file name.

6.1.1 VerifyDF switches

A switch specification is a letter, optionally preceded by a '-' or '~' to reverse the sense of that switch. The valid switches are:

- f print "flattened" DF file (all Imports and Includes structure removed)
- n check that all files seem necessary (*default*) You would probably want this when using the /f switch
- t retrieve DF files to temporary files (*default*)

The default setting for the /n and /t switches is *on*, while the /f switch is *off*. You can change the default setting of any switch by using a global switch. Any switch given with no file name (i.e., just a slash and switches) establishes the default setting for that switch. Unless overridden, that default applies to following commands.

6.2 VerifyDF limitations

The total number of files that VerifyDF can check, *including* those from imported and included DF files, is 1000.

7 DFDelete

When you have finished working on a DF file and have SModel'ed its files out to their remote locations, you can free up space on your local disk by running DFDelete on the DF file. This program scans a DF file (and the ones it Includes), and generates a command in Line.cm that can be used to delete the files described by the DF file. Deleting these files is safe because you can be certain, after running VerifyDF and SModel on a DF file, that all needed files have been stored remotely.

DFDelete will *not* add to the delete command any file on the local disk that has a create date *different* than that listed for it in the DF file.

7.1 DFDelete's command line

The DFDelete command line has the form

```
>DFDelete [/<global switches>] DFfile1[/<local switches>] ... DFfilen[/<local switches>]
```

As usual, global switches are optional and control the deletion of following DF files. You can also set global switches by giving an empty DF file name.

7.1.1 DFDelete switches

DFDelete has only one switch which can be preceded by a '-' or '~' to reverse its sense:

r also delete Imported and ReadOnly files

7.2 DFDelete limitations

Each DF file processed by DFDelete must contain no more than 450 files. This applies to each Included DF file as well.

8 DFSubstitute

Although a DF file is just a text file that can be edited by the user, it is still awkward to make simple repetitive changes to large numbers of DF files. The program DFSubstitute can be used to simplify this task. It can:

- change hosts or directories,
- move an Imported file (e.g. Heap.bcd) from one DF file (PilotPublic.df) to another (ComSoftPublic.df), and
- insert or delete Imported files.

DFSubstitute makes changes to a set of DF files according to commands in a substitution script file. The commands are executed in order from first to last for each line in a DF file. This means that later commands can take advantage of the substitutions made by previous commands. Included DF files are processed in the usual bottom-up recursive fashion. The rewritten DF files are not stored remotely by DFSubstitute; you must use SModel to do that.

The four DFSubstitute commands are:

- Rename [RHS] <loc₁> To <loc₂>

Rename changes the remote location on the left hand side (Directory, Imports, Includes, or ReadOnly) of matching DF file lines. If RHS is specified, matching right hand sides (ReleaseAs or CameFrom) are changed. Each <loc> can be a host (e.g. [Igor]), a directory

(e.g. <Emerson>Tajo>), a file name (e.g. **MFile.bcd**), or a combination of all three. For example, to rename [Igor] to [Idun] in all left hand sides, use the following:

```
Rename [Igor] To [Idun]
```

To change all r.h.s. references to [Igor]<Ramona> to be [Idun]<Int>, use

```
Rename RHS [Igor]<Ramona> To [Idun]<Int>
```

It is also possible to change just a subdirectory, e.g.

```
Rename CWF>Public To Other>Public
```

To change the location of just one file, use a command like the following:

```
Rename [Igor]<Emerson>Tajo>Private>NSFileTransfersA.bcd  
To [Igor]<Emerson>NSFileTransfer>Friends>NSFileTransfersA.bcd
```

- Move Import <name> From <DF file₁> To <DF file₂>

This moves an import from the Using list of one DF file to another. If no Using list files remain the first Imports line is entirely deleted. For example,

```
Move Import Heap.bcd From Pilot.df To ComSoftPublic.df
```

- Delete Import <name> From <DF file>

This just removes the specified import from the Using list of an imported DF file. If no Using list files remain the entire Imports line is deleted. For example, to remove all importations of Space.bcd from PilotPublic.df, use

```
Delete Import Space.bcd From PilotPublic.df
```

- Insert Import <name> From <DF file>

This simply adds an import to the DF file's Using list, e.g.

```
Insert Import Environment.bcd From MesaPublic.df
```

8.1 DFSubstitute's command line

The DFSubstitute command line has the form

```
>DFSubstitute ScriptFile DFfile1 ... DFfilen
```

The first file is a substitution script (default extension ".script") that specifies the changes to be made to the following DF files. DFSubstitute has no switches.

8.2 DFSubstitute limitations

Each DF file processed by DFSubstitute must contain no more than 600 files. This applies to each Included DF file as well.

Important!

If there are *spaces* embedded in a token, please quote them. For example:

Rename "[Walter:Very Nice:Music]<Carlos>" To "[Wendy:Very Nice:Music]<Carlos>"

9 DFDisk

DFDisk produces a file "Disk.df" that describes the current search path. With the exception of "\$" files and a few kinds of log files, it lists all files on the search path with the create date and remote location found on the local disk. The remote location is taken from the RemoteName property in each file's leader page. DFDisk is most useful when you are trying to find the remote location for files, or when you are trying to save all your files before reformatting the volume. It can also tell you about new files that should be recorded in a DF file, since the RemoteName property for these files will not have been set and they will be listed under the remote "location" [Unknown]<Unknown>.

9.1 DFDisk's command line

DFDisk has no switches, and its command line is simply

```
>DFDisk
```

9.2 DFDisk limitations

The maximum number of files that DFDisk can process is 1000.

10 DFTool

The DFTool provides a window interface to the other DF programs. It supports BringOver, SModel, VerifyDF, DFDelete, DFDisk, DFSubstitute, as well as program librarian CheckOut and Query. Since the different commands share several DF implementation modules, fewer resources are used by this tool than by the separate DF programs. The price you pay for this is that only one command can be run at a time as opposed to having multiple Executive windows the run the DF programs from. There is currently no command line interface to the different commands.

The DFTool communicates through four subwindows: a message, form, command, and TTY subwindow. The TTY subwindow is used to log the progress of each command, and for interaction with the user (e.g. for passwords and for file transfer confirmations). There is also an Options window which is used to set infrequently modified parameters.

fine point: A picture of the tool will be supplied here eventually.

10.1 Form subwindow

The fields that can be used as arguments to a command are listed in the form subwindow. The first row has a five Booleans that correspond to the most widely used DF program command line switches. The next four rows are string items that provide parameters for the DF commands.

Booleans:

Confirm	means ask for confirmation before retrieving or storing a file. It has the same effect as /a on the BringOver and SModel command lines. The default is TRUE .
GetOnlyExports	means retrieve only files marked as exports when bringing over a DF file. This is the same as the BringOver /p switch. The default is FALSE .
VerifyFilesExist	means check that files exist in the right place and version. It is the same as the BringOver /v switch. The default is FALSE .
DontStoreRemotely	means never store files remotely. This is equivalent to SModel's /n switch. The default is FALSE .
VerifyRemoteOnStore	when doing an SModel! , means verify that files are really on the remote server and store if necessary. This is the same as the SModel /v switch. The default is FALSE .

Fill-ins:

DF Files:	are the names of the DF files to be used for any commands. If a DF file's name contains spaces (e.g. a fully qualified DF file name on an NS server), the name must be enclosed in double-quotes ("").
Files:	is a list of files (separated by spaces) for the next command to act upon. These might be, for example, the specific files that BringOver! should retrieve from a DF file (BringOver only files mode).
LocalDir:	means that BringOver! should do all retrievals to this directory. This is equivalent to BringOver's localDir/c command. If the directory is not a complete path name, i.e. it does not begin with <, it is assumed to have a < > prepended.
Checkout Reason:	is given to the program librarian when files are checked out.

10.2 DFTool command subwindow

The fields in the command subwindow are the following:

CheckOut!	checks out the libjects for the DF files listed in the DF Files: line and the files listed in the Files: line. The CheckOut Reason: is passed to the program librarian.
Query!	displays information regarding the libjects for the DF files listed in the DF Files: line and the files listed in the Files: line.
BringOver!	invokes the BringOver algorithm on the DF files listed in the DF Files: entry. If any files are also listed in the Files: line, BringOver will enter only files mode and retrieve just those files.
SModel!	stores back the DF files and its components within of the DF files listed in the DF Files: entry.
VerifyDF!	verifies the completeness and consistency of each DF file named in the DF Files: entry.
DFDelete!	invokes the DFDelete algorithm on each DF file listed in the DF Files: line.
DFDisk!	generates a file "Disk.df" that describes all the files on the current search path.
DFSubstitute!	modifies DF files listed in the DF Files: entry according to the substitution script file named in the Files: entry..
Options!	creates an options window for the DFTool if one does not already exist.

10.3 DFTool Options window

The Options window is created by the **Options!** command. It contains a string item and Booleans that govern the DFTool's operation, but which are typically changed only infrequently. The string item, **LocalDFDir:**, is initialized from the **User.cm LocalDFDir:** entry. The Booleans correspond directly to the command line switches for each DF program. After changing the options, invoke **Apply!** to invoke those changes. The **Abort** command will restore the options to what they were before the **Options!** command was invoked. Both **Apply!** and **Abort!** perform the appropriate actions and then destroy the Options window.

10.4 DFTool User.cm fields

The DFTool uses the same [DFTool] section in the User.cm as the other DF programs. Besides the fields described above in Section 3, the standard **InitialState:**, **TinyPlace:**, and **WindowBox:** entries can be set.

11 The IncludeChecker and DF files

If you are a DF software user (i.e. if you have a [DFTool] section in your User.cm), the released Klamath IncludeChecker can process a DF file as well as a lists of files. The

IncludeChecker has a **df/c** command that is used to specify the DF file to check. For example,

```
>IncludeChecker MyPackage.list/cio df/c MyPackage.df
```

will analyze the files described by MyPackage.df and generate an includes and includedBy listing in MyPackage.list and a rebuild command in Line.cm.

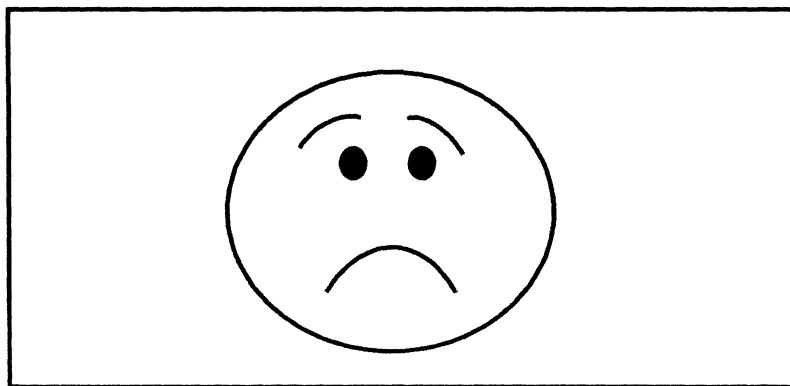
Each file listed in the DF file is looked up on the local disk. If there, that version is analyzed regardless of its create date. If the file is not local, the remote version is checked (the remote path is gotten from the DF file). Because the IncludeChecker believes that the local versions of files are the “truth” (the assumption is that the DF file was brought over and some changes were made to its files), it can be used to verify a component that has not yet been SModel’ed.

Note: *VerifyDF operates differently:* VerifyDF checks the particular snapshot of a package described by a DF file. Local versions of files with different create dates are ignored and the remote versions are used instead. This means, in general, that a DF file has to be SModel’ed before VerifyDF can be run on it.

The IncludeChecker also has a **/r** (examine DF imports) command line switch that may be specified when DF files are processed. When set, the IncludeChecker will also analyze imported files. If you believe that no imported file has changed since you brought over the DF file, you can use **/-r** to reduce the IncludeChecker’s running time. The initial default for the **/r** switch is *on*. In general, you should not use **/-r**.

If you have a [DFTool] section in your User.cm, some additional parameters and an additional command appear in the IncludeChecker window. The **DF File:** entry names the DF file to be processed when the **Check DF!** command is invoked. The Boolean **Examine DF Imports** appears in the IncludeChecker Options window and has the same effect as the **/r** command line switch; it is initially **TRUE**.

12 Dealing with Problems



As much as we try to avoid them, problems still crop up. Here are some common problems and ways you could deal with them.

Network problems

Problem: You did a BringOver, modified files, and do a Smodel. Unfortunately, you lost a connection in the middle of a Smodel, so only some of the files were stored.

Solution: When the network is up, do a Smodel with the "verify files exists" mode to make sure **all** the files are stored back.

You forgot to check out the Libject.

Problem: Shame on you. Smodel probably gave you a warning. Using the "don't check librarian" switch is probably dangerous since someone else may have checked out the DF. and has worked on the program.

Solution: You should try to check out the libject. There are two cases:

If you are denied access, go talk to the person who checked out the libject and try to coordinate the modifications. If you changes do not overlap, you are lucky.

If you are given access to the libject, you may still be in trouble since someone else may have done a checkout and a checkin during the period you were working on the component. Find out the checkin date of the libject and see if that occurred during your work after you did a BringOver.

The Librarian is down.

Problem: Smodel fails since the libject cannot be checked in.

Solution: You are safe since no one else can modify the files you have checked out. Frequently, you will only do a checkout of the DF file itself. In that case, the files listed in the DF file may have been stored back already, leaving only the DF file "un-stored". However, the DF file is already modified with the new create dates, including that of the DF file itself. Thus a subsequent Smodel (done when the librarian is up) will use the local DF file and will believe that all the required files are remotely stored. When the librarian is up, do a Smodel with the "verify files exists" mode to store the DF file back.

The programs tell me that it can't parse dates in my DF

Problem: Your time zone requires you to specify time relative to GMT. Some parts of the world require you to specify the time in the format of hh:mm:ss + N GMT. The released parser is not able to parse that correctly. This is a limitation in the Klamath version .

Solution: Please wait for the announcement of the newer, better, and more worldly DF software. The new parser should be able to handle various time formats.

I want to move the files pointed to by a DF from one location to another.

Problem: You want to change the "Directory" statements and move the files to their new destinations..

Solution: do a BringOver, run a DFSubstitute, and do a Smodel.

How do I stop the DF Tool?

Problem: You made filled in the wrong parameters, or perhaps the made some other mistakes, and you want to stop the operations.

Solution: Press the **STOP** key, and keep on trying until it stops.



Index

#, 4-2
' , 4-2, 10-2
--, 4-3
;, 4-2
? , 4-2
@ , 4-2, 10-3, A-2
\ , 4-3
abbreviation-expansion pair, 2-1
accelerator

- Pilot, B-6
bootable floppy, 22-1
Booting other volumes from CoPilot, B-6
Booting other volumes from Othello, B-4
bootmesa, 21-2
bounds checking, 19-4
Break, 15-1
breakpoint, 24-2, 24-8
breakpoint commands, III-13
breakpoints, 15-1
 conditional, 28-2, 28-8
Brownie, 8-1
 command line, 8-1
 commands, 8-2
 example, 8-3
 parameters, 8-2
 script file, 8-1
BS, 4-1
BW, 4-1
C/Mesa, 17-1
CALDEBUG, 24-2, III-7
CATCH CODE, 23-8
catch code, 23-4, 23-8
ChangeCommandName, 4-4, 4-9
changing user information, 6-1
character class, 3-5
character patterns
 finding, 14-1
Chat
 form subwindow, 32-2
 special keys, 32-3
 TTY subwindow, 32-2
 user interface, 32-1
Chat User.cm, 32-4
Check Drive, A-3
chording, I-12
Clear, 15-1
Clearinghouse, 4-4
Clearinghouse, 30-1
client, I-3
ClientRun, 4-3, 4-7
Close, A-7
CloseVolume, 4-3
closure, 3-6
CoCoPilot, 24-2
CODE, 21-3
code links, 19-2, 21-1, 23-3
CodePack, 21-3, 23-5, 23-7, 23-8, 23-11
code pack, 23-1, 23-2, 23-3, 23-5, 23-10
code packing, 23-2
code segment, 23-1, 23-2, 23-5, 23-10
codelinks, 4-5, 4-7, 4-10
command files
 passwords in, 9-3
command item, I-9
command line
 expansion, 4-2
 interpretation, 4-3
CommandCentral, 4-3, 4-6, 4-7, 18-1
 command subwindow, 18-1
 User.cm, 18-2
comment, 4-3
Compare, 13-1
 command line, 13-3
 examples, 13-4
 file pair switches, 13-3
 form subwindow, 13-2
 via a window interface, 13-1
 via the Executive window, 13-3
Compiler, 4-6, 19-1, 20-1, 23-2, 27-1
 command line, 19-2
 error messages, 19-6
 examples, 19-3, 19-6
 failures, 19-8
 limitations, 19-8
 switches, 4-10, 19-3
compiler error log, III-2
compiler switch defaults, III-2
Compiler switches, III-2
Compiling, III-2
COMPLETE, 4-1
concatenation, 4-4
configuration, I-1
configuration description file, III-3
configuration description language, 17-1
context, III-8
CONTROL statement, III-3
control transfer, 28-1, 28-2
CONTROL-C, 4-1
CONTROL-P, A-2
CONTROL-X, 4-1
converting object files to boot files, III-6
CoPilot, I-3
CoPilot, 24-1
CoPilotDLion.boot, 24-1
Copy
 Executive command, 4-4
CountPackage, 28-1, 28-1
 getting started, 28-6
 limitations, 28-5
 operation, 28-4
 sample session, 28-6
 user interface, 28-2
Create, 15-2
Create Physical Volume, A-3
CreateDir, 4-4
Creating a source file, III-1
creation date, 9-4, 9-5

cross jumping, 24-10
cross reference, 27-1, 27-3
 by callee, 27-3
 by caller, 27-3
cross-jumping, 19-4
current selection, I-10
CWD, 4-4
Dandelion, B-1, I-2
Debug Ops menu, 15-1, 24-9
 Attach, 15-1, 24-9
 Break, 15-1, 24-10
 Clear, 15-1, 24-11
 Trace, 15-1, 24-12
Debug.log, 24-2, 24-3, 24-32
Debuggee.outload, 24-1
Debugger, 11-3, 15-1
 (also see Sword Debugger)
debugger
 breakpoint, 24-8
 breakpoint commands, 24-9
 commands, 24-8
 commands summary, 24-34
 CoPilot, 24-1
 core image, 24-1
 cross jumping, 24-10
 current context, 24-4, 24-15
 Debug Ops menu, 24-9
 error messages, 24-23
 input conventions, 24-4
 installation, 24-32
 interpreter, 24-19
 interpreter grammar, 24-33
 kill debugger session, 24-17
 loaded configurations, 24-15
 logical volume, 24-1
 low level facilities, octal break, 24-18
 low level facilities, octal read, 24-18
 low level facilities, octal write, 24-18
 low-level facilities, 24-17
 Mesa data types, 24-5
 new session, 24-2
 options window, 24-5
 output conventions, 24-5
 procedure calls, 24-21
 proceed from debugger, 24-17
 process display, 24-12, 24-16
 quit from debugger, 24-17
 remote debug, 24-18
 runtime state, 24-12
 stack display, 24-12
 symbols, 24-4
 teledebug, 24-18
 tracepoint, 24-8
 user interface, III-8
 User.cm, 24-32
 Userscreen, 24-17
worry mode breakpoints, 24-19
Debugger Pointer, 24-2, E-2
Debugger.outload, 24-1, E-2
debuggerDebugger, 24-2, A-5, A-11
debugging
 Profile Tool option, 6-1
 storage leaks, 25-1
DebugHeap, 25-1
 client words, 25-1, 25-3
 example, 25-4
 heap OwnerChecking switch, 25-4,
 25-4
 node storage usage, 25-1
 nodes, examining, 25-2
 private heaps, 25-2
 storage leaks, 25-1
 system heaps, 25-2
 zone, 25-2
DEFINITION, 2-1, 27-2
Definition of terms, I-3
DELETE, 4-1
Delete
 Executive command, 4-4
Delete Boot File, A-9
Delete Temporary Files, A-6
Describe Physical Volumes, A-3, A-4
description modules, III-1
DestDir, 4-8
Destroy, 15-2
diagnostic boot, B-2
diagnostic microcode, A-6
Diagnostic Microcode Fetch, A-9
dictionary, 2-1
Dictionary Tool, 2-1
 commands, 2-2
 Dictionary Tool, 2-1
 EXPAND, 2-1
 file format, 2-2
 User.cm, 2-2
DIRECTORY, 17-7
Directory
 Othello command, A-8
directory statement, III-3
DISCARD CODE PACK, 23-8
disk
 label check, A-12
 Shugart SA1000, A-2
 Trident 300, A-2
 Trident 80, A-2
Disk booting, B-2
Disk Label Check, 24-2
display screen
 inverting, 1-1
 preservation, 1-1
DMT.bcd, 1-1

domain, 6-1
 setting, 6-1
Edit, 15-2
Edit Dictionary, 2-1
Edit Ops menu, 3-3
editable window, 15-2
editing characters, 4-1
EditOps menu, 3-4
Editor property sheet, 3-3
Editor property sheet accelerator, 3-4
Editor Symbiote, I-16
editor symbiote
 use, 3-1
empty window, 15-2
ENABLE, 23-4
Ending a session, B-16
ENTRY VECTOR, 23-8
entry vector, 23-3, 23-4
enumerated item, I-9
Erase, A-5
error recovery, B-11
errors, I-3
escaped character, 3-5
Ethernet, A-11
Ethernet, I-2
Ethernet booting, B-3
Examining and changing the state, III-12
example volume configurations, B-5
EXCEPT, 23-6, 23-9
Exec Ops menu, 4-10
 CoPilot, 4-10
 File Window, 4-10
 Load, 4-10
 New Exec, 4-10
 Power Off, 4-10
 Quit, 4-10
 Run, 4-10
 Start, 4-10
ExecOps menu
 File Window, 15-1
Executive, 4-1, 9-4, 20-2
 built-in commands, 4-3
 command line expansion, 4-2
 command line interpretation, 4-3
 editing functions, 4-1
 Exec Ops menu, 4-10
 loading programs, 4-5, 4-7
 pattern matching, 4-2
 running programs, 4-7, 4-9
 User.cm, 4-10
EXPAND, 2-1
expansion, 4-2
EXPORTS, 17-6
extension
 .brownie, 8-1
 .list, 23-3
.map, 23-3
.pack, 23-2
.scratch\$, 16-1
.tds, 7-2
External, 32-4
Fetch, A-7, A-9
Fetch Boot File, A-7
file
 ArpaFileServer, 38-1
ArpaFileTool, 37-1
code, 17-1
comparing, 9-6
copy, 4-4, 4-8
copying local, 10-4
creation date, 9-2, 9-4, 9-5
dates, 4-5
deleting local, 10-3
deleting remote, 9-6
ID, 4-5
listing local, 10-3
listing remote, 9-5
local, 9-2
name completion, 4-1
object, 17-1, 19-1, 27-1
object, version stamp, 27-2, 27-3
options for listing local, 10-4
partial, 11-3
protection, 4-5
read date, 9-5
remote, 9-2
renaming, 9-6
retrieving, 4-9, 9-1, 9-4, 10-3
size, 4-5
storing, 9-1, 9-4, 10-3
symbols, 17-1
text, 15-1
times, 4-5
transfer, 10-2
write date, 9-5
File Name, 5-1
File Tool, 9-1, 10-1
 command subwindow, 10-3
 form subwindow, 10-2
operational notes, 10-5
options window, 10-4
User.cm, 10-4
file transfer, 9-1, 10-2
 ArpaFileServer, 38-1
 ArpaFileTool, 37-1
File window, 4-10, 15-1
 Create, 15-2
Debug Ops menu, 15-1
Destroy, 15-2
Edit, 15-2
editable, 15-2
empty, 15-2

Exec Ops menu, 15-1
Load, 15-2
menu, 15-2
non-editable, 15-2
Reset, 15-2
Save, 15-2
Store, 15-2
Time, 15-3
User.cm, 15-3
file-related tools, II-2
filename
 fully-qualified, II-1
 simple, II-1
Filestat, 4-5
FileTool, 31-1, 31-3
Find, 14-1
 command line, 14-1
 examples, 14-3
 switches, 14-1
floppy, 11-1
 bootable, 22-1
 disk drive, 11-1
Floppy
 Executive command, 4-5
Floppy booting, B-3
Floppy commands, 11-1
 command line, 11-1
 error messages, 11-4
 examples, 11-3
 partial files, 11-3
 switches, 11-2
font
 face, 16-3
 family, 16-3
 names, 16-3
 point size, 16-3
form subwindow commands, I-8
form subwindows, I-1
Formatter, 20-1
 command line, 20-1
 examples, 20-5
 failures, 20-6
 rules, 20-3
 switches, 20-2
 User.cm, 20-2, 20-5
FRAME, 21-3
FRAME PACK, 21-3, 23-9, 23-11
frame pack, 23-1, 23-3, 23-11
FRAME PACK MERGES, 23-10
frequency statistics, 28-1, 28-8
FTP (File Transfer Protocol), 9-1, 31-1,
 31-3
 ArpaMailServer, 38-1
 ArpaFileTool, 37-1
 command abbreviation, 9-1
 command line, 9-1
examples, 9-7
switches, 9-1
FTP protocol, 9-1
functions
 global, I-20
 keyboard, I-19
General Tools, I-1
germ, 21-1, A-6, III-6
Germ Fetch, A-9, A-10
GLOBAL FRAME, 21-3
global frame, 21-3, 23-1, 23-3, 23-4
 debugger display, 24-12, E-17
 packaged, 21-3
 unpackaged, 21-3
global replace, 3-2
heap debugging, 25-1
HeraldWindow, 5-1
 Boot from: menu commands, 5-1
 User.cm, 5-2
IMPORTS, 17-6
IMPORTS statement, III-3
improving swapping performance,
 III-6
Inactive menu, I-15
IncludeChecker, 26-1
 command line, 26-4
 examples, 26-5
 form subwindow, 26-2
 option window, 26-3
 switches, 26-4
 User.cm, 26-7
initial microcode, A-6
initialization code, 23-1, 23-4
initializing debugger volumes, B-9
input focus, I-6
Installing boot files, B-9
Installing the development
 environment, B-10
integration machine, 32-13
Interactive Terminal Service, 32-1
internal scavenger, A-5
Interpress, 16-1
Interpreter Tool, E-5
 file subwindow, E-9
 form subwindow, E-6, E-14
 sessions, E-5, E-8
Interpreter form subwindow commands,
 E-6
 client, E-6
 processes, E-7
 configs, E-7
 source, E-7
 findModule, E-7
 rep?, E-7
 showType, E-7
 type&bits, E-8

Index

watch, E-8
Interpreting signals, III-14
Interrupt, 24-2
invoking the Binder, III-3
invoking the compiler, III-2
invoking the debugger, III-7
kill
 debugger session, 24-17, E-21
Lexicon, III-1
LexiconClient, III-1
libject
 setting prefix, 6-2
 setting suffix, 6-2
Librarian, 6-2
 setting, 6-2
links, 23-3
List Bad Pages, A-4
List Drives, A-2, A-3
List Logical Volumes, A-5
List Physical Volumes, A-3
List Remote Files, A-7
Lister, 27-1
 command line, 27-1
 switches, 27-4, 27-5
ListRemoteHosts, 32-12
Load, 4-5, 4-10, 15-2
load handle, 4-5, 4-9
Loader, 23-1
loader
 MakeBoot, 21-1
loading programs, 4-5, 4-7
loadmap, 21-2, 21-3
local file, 9-2
local file system, II-1
local frame
 debugger display, 24-12, E-17
logical volume, A-1
 debugger, 24-1, A-5, E-1
 debuggerDebugger, 24-2, A-5, E-2
 foreign, A-5
 normal, A-5
 Othello commands, A-4
 types, A-5
logical volumes, B-4
Login, 4-5, A-7
login name, 6-1
 setting, 6-1
login password, 6-1
 setting, 6-1
logout, 1-1
mail
 answering, 30-1
 ArpaMailTool, 39-1
 changing mail files, 30-4
 deleting, 30-1
 forwarding, 30-1
moving, 30-1
reading, 30-1
retrieving, 30-1
saving, 30-1
sending, 30-1
mail registry, 6-1
 setting, 6-1
MailFileScavenger, 30-1
MailTool, 30-1
 Abort!, 30-6
 Active.nsMail, 30-2
 Append!, 30-4
 Apply!, 30-6
 AutoDisplay, 30-6
 attachments, 30-2
 current mail file, 30-2
 current messages, 30-2
 Delete!, 30-4
 Display!, 30-3
 DisplayOnNewMail, 30-3, 30-6
 ExpandPvtDLs:, 30-5
 Expunge!, 30-4
 File:, 30-5
 Flush Remote, 30-4
 Forward!, 30-5
 Hardcopy!, 30-3
 Landscape Font:, 30-7
 Mail File:, 30-6
 Move!, 30-5
 New Form!, 30-5
 New Mail!, 30-3
 One Per Page, 30-6
 Options!, 30-5
 Orientation:, 30-6
 Output To File, 30-6
 Portrait Font:, 30-7
 Printer:, 30-7
 Sides:, 30-6
 Sort!, 30-5
 table of contents, 30-2
 To:, 30-5
 Undelete!, 30-4
 User.cm, 30-2, 30-11
 via the Executive, 30-7
MAIN, 23-4, 23-8
mainline code, 23-4
Maintain, 30-1
 Add!, 30-18
 Add! Remove! Mailbox:, 30-18
 Add: Self!, 30-15
 Alias:, 30-17, 30-18
 Aliases!, 30-15, 30-18
 Another!, 30-18
 Anyentry, 30-18
 Argument:, 30-16
 CheckNames, 30-18

Create!, 30-17, 30-18
Delete!, 30-17, 30-18
Destroy!, 30-18
Details!, 30-17, 30-18
friends of a group, 30-19
Group:, 30-15
Individual:, 30-16, 3017
Level, 30-18
Mat ches!, 30-18
Members!, 30-15
NameList:, 30-17
Normal Level, 30-15
Owner Level, 30-16
owners of a group, 30-19
Password:, 30-16
Remove!, 30-17, 30-18
Remove: Self!, 30-15
Set! Password, 30-16, 30-18
Set! Remark:, 30-17, 30-18
Summary!, 30-15, 30-16, 30-17
UseBackground, 30-19
Which:, 30-17
maintenance panel, **B-2**
maintenance panel error codes, B-12
maintenance panel initialization
 codes, B-3
MakeBoot, **21-1**, III-6
Makeboot, 23-1
MakeBoot
 commands, 21-2
 examples, 21-5
 loader, 21-1
 parameter files, 21-1, 21-2, 21-3,
 21-4
 switches, 21-3
MakeDLionBootFloppyTool, **22-1**
 command subwindow, 22-2
 form subwindow, 22-1
Making boot files, III-6
Map Log, 24-2
menu
 Boot from:, 5-1
 current search path directories, 12-2
 Debug Ops, 15-1
 Exec Ops, 4-10, 15-1
 existing search path directories, 12-2
 File Window, 15-2
MENU key, I-12
menu prompts, I-10
menus, I-1
MFileServer, **31-1**
 executive commands, 31-2
 form subwindow, 31-2
 User.cm, 31-2
microcode
 diagnostic, A-6
initial, A-6
Pilot, A-6
ModuleMaker, C-1
modulename.bcd, III-3
modules, 23-2, 23-4
mouse, I-2
moving files, 10-2
MP codes, E-2
multilingual debugger, E-1
multiword read-only constants, 23-4
name
 login, 6-1
 setting, 6-1
 user, 6-1
name frame, I-14
name frame operations, I-14
naming conventions, II-1
New Exec, 4-10
nil checking, 19-5
non-diagnostic boot, B-2
non-editable window, 15-2
NS, 30-1
NSTerminal
 terminal types, 32-7
NSTerminal user.cm, 32-10
numeric item, I-10
object file, 19-1, 27-1, C-1, III-2
 version stamp, 27-2, 27-3
Offline, A-4
Online, A-3
Open, A-7
OpenVolume, 4-6
organization, 6-1
 setting, 6-1
Othello, I-3
Othello, A-1
 accessible disk drives, A-2
 booting, A-1
 checking a pack, A-3
 command file, A-2
 command line, A-2
 commands, A-2
 diagnostic microcode, A-6
 exiting, A-12
 fetch commands, A-7
 initial microcode, A-6
 logical volume, A-4
 physical volume, A-3
 Pilot microcode, A-6
 routing tables, A-11
 time, A-10
Packager, **23-1**, 27-1, III-6
 command line, 23-2
 example, 23-11
 information about modules, 23-4
 operation, 23-12

- packaging description language, 23-5
- switches, 23-2
- packaging, 28-1
- page fault
 - tracing, 28-21
- password, 4-5, 6-1, 30-18
 - setting, 6-1
- Performance Measurement Tool, 28-1, 28-2
 - concepts, 28-9
 - getting started, 28-15
 - limitations, 28-14
 - operation, 28-13
 - sample session, 28-15
 - terms, 28-9
 - user interface, 28-10
- performance monitoring, 28-10, 28-17
- Performance Tools, 28-1
 - Ben, 28-1, 28-21
 - CountPackage, 28-1
 - Measurement Tool, 28-1, 28-8
 - PerfPackage, 28-1, 28-8, 28-12
 - Spy, 28-1, 28-17
- PerfPackage, 28-1, 28-8
 - concepts, 28-9
 - getting started, 28-15
 - limitations, 28-14
 - operation, 28-13
 - sample session, 28-15
 - terms, 28-9
 - user interface, 28-10
- physical volume, A-1
- Physical Volume Scavenge**, A-6
- Physical Volume Scavenger, A-6
- Pilot, I-2
 - Pilot
 - internal scavenger, A-5
 - microcode, A-6
 - Pilot error messages, B-14
 - Pilot file backing cache, B-7
 - Pilot Microcode Fetch, A-9
- PopWD, 4-6
- pound sign, 4-2
- Power Off**, 4-10, A-7, A-12
- Print, 16-1
 - command line, 16-1
 - defaults, 16-3
 - examples, 16-2
 - font names, 16-3
 - formatting, 16-4
 - switches, 16-2
 - User.cm, 16-4
- private heap
 - debugging, 25-2
- proceed
- from debugger, 24-17, E-21
- process
 - debugger display, 24-12, E-17
- ProcessInBackground**, 4-6
- ProcessInNormalPriority**, 4-6
- Profile Tool, 6-1, 9-3
 - form subwindow, 6-1
- PushWD**, 4-6
- Quantum 2040
 - 2080, A-2
- question mark, 4-2
- Quit**, 4-10, A-7, A-12
- quit
 - from debugger, 24-17, E-21
- read date, 9-5
- recompile, 26-1
- referencing environment, III-8
- registered commands, 4-3
- registry, 6-1
 - setting, 6-1
- remote connection, 9-3
- remote debuggee, 24-18
- Remote Executive
 - additional commands, 32-11, 32-14
 - character codes, 32-13
- Remote executive
 - user interface, 32-11
- Remote Executive User.cm, 32-12
- remote filename conventions, II-1
- Remote System Administration, 32-1
- RemoteExec**, 32-12
- Rename**, 4-6
- repetitive tool execution, 7-1
- replace field, 3-2
- replacement expression, 3-6
- Reset**, 15-2
- Reset Priority**, 5-2
- Reset Switches**, 5-2
- RET**, 4-2
- root window, I-6
- RS 232C, 32-8, 32-9
- Run**, 4-7, 4-10
 - run!
 - Command Central command, III-5
 - Running a program, III-5
 - running programs, 4-7, 4-9, -10
- sample session, III-8
- Save**, 15-2
- Scavenge**, A-5
- scavenger, A-6
- script file
 - Tool Driver, 7-1, 7-3
- scrollbars, I-7
- search and pattern matching-facilities, 3-2
- search context, 3-3

search expression, 3-5
search field, 3-2
search path, 4-6, 4-8, 12-1
Search Path Tool, 12-1
 commands, 12-1
 current directories menu, 12-2
 existing directories menu, 12-2
 form subwindow, 12-1
searching
 character patterns, 14-1
SEGMENT MERGES, 23-9
semicolon, 4-2
SendTool, 30-5, 30-7
 Answer!, 30-7
 cc:, 30-8
 Deliver!, 30-7
 Destroy!, 30-7
 Get!, 30-8
 If Need Reply-To, 30-8
 Invalid OK, 30-8
 MailNote, 30-8
 MailNote with attachment, 30-8
 New Form!, 30-7
 private distribution lists, 30-10
 public distribution lists, 30-9
 Put!, 30-7
 recipients, 30-9
 Reply-To:, 30-10
 Reset!, 30-7
 SendAs:, 30-8
 SendTool via the Executive, 30-10
 Subject, 30-9
 Text, 30-8
 User.cm, 30-12
session
 ending, 1-1
Set Boot File Default Switches, A-9
Set Debugger Pointers, A-11
Set Hardware Clock Upper Limit, A-10
Set Physical Volume Boot Files, A-10
Set Priority Up, 5-2
Set Switches, 5-1
SetClientVolume, 4-7
SetErrorLevel, 4-7
SetPriority, 4-7
SetSearchPath, 4-8
setting breakpoints, III-10
Setting debugger pointers, B-10
setting user information, 6-1
ShowAccessList, 32-12
ShowSearchPath, 4-8
single quote, 4-2, 10-3
SMTP (Simple Mail Transport Protocol)
 ArpaMailTool, 39-1
 ArpaSendTool, 39-5, 39-6
 MailFilScavenger, 39-10

Snarf, 4-8
snarf
 Executive command, III-5
Snarf command, III-5
SourceDir, 4-8
sourcename.bcd, III-2
sourcename.errlog, III-2
SPACE, 21-3
Spy, 28-1, 28-17
 error messages, 28-20
 getting started, 28-19
 limitations, 28-21
 operation, 28-19
 user interface, 28-17
stack
 debugger display, 24-14, E-19
Start, 4-9, 4-10
Statistics, 29-1
 command line, 29-1
 example, 29-2
statistics
 frequency, 28-1, 28-8
Statistics
 switches, 29-1
statistics
 timing, 28-1, 28-8
Statistics
 types, 29-2
storage
 debugging leaks, 25-1
Store, 15-2
StringCompactor, C-1
subwindow boundaries, I-8
swap units, 23-1
swapping, 23-1
Sword Debugger, E-1
 breakpoint commands, E-14
 breakpoints, E-14
 client, E-1
 commands, E-14
 commands summary, E-33
 core image, E-2
 cross jumping, E-14
 current context, E-9, E-17, E-20
 Debuggee.outload, E-2
 error messages, E-26
 events, E-1
 freeze, E-4
 input conventions, E-10
 interpreter, E-22
 Interpreter, E-1, E-2, E-3
 interpreter grammar, E-32
 Interpreter Tool, E-5
 kill debugger session, E-21
 local debugging, E-1, E-8
 log file, E-9

- logical volume, E-1
low-level facilities, E-22
low-level facilities, octal read, E-22
low-level facilities, octal write, E-22
low-level facilities, octal set break, E-22
Mesa, E-1
outload debugging, E-2, E-8
output conventions, E-10
procedure calls, E-24
proceed from debugger, E-21
quit from debugger, E-21
remote debugging, E-2, E-9
runtime state, E-17
stack display, E-17, E-19
styles of debugging, E-1
Sword Tool, E-3, E-4
symbols, E-10
thaw, E-4
user interface, E-3
user.cm, E-32
userscreen, E-22
Sword Tool, E-3
 Frozen process, E-4
 Loadstate facilities, E-3
 Process facilities, E-3
symbiote, I-16
Symbiote menu, I-16
symbol table, 17-1, 19-1, 27-2
system heap
 debugging, 25-2
System Overview, I-1
TAB, 4-2
table-compiled, 23-10
TableCompiler, C-1
 command line, C-2, C-5
 Examples, C-5
 Switches, C-6
tag item, I-10
tail recursion, 19-4
Tajo, I-2, E-2
TCP/IP Related Tools and Applications, V-1
 Arpa Getting Started, 33-1
 ArpaCacheAddress, 34-1
 ArpaChat, 35-1
 ArpaFileServer, 38-1
 ArpaFileTool, 37-1
 ArpaMailTool, 39-1
 ArpaRemoteExec, 36-1
TDE.log, 7-3
teledebug, 24-2, 24-18
Telnet Protocol, 35-1
text item, I-10
Text Ops menu, I-15
text subwindow commands, I-21
text subwindows, I-10
TFTP (Trivial File Transfer Protocol)
 ArpaFileTool, 37-1
 ArpaMailServer, 38-1
thrashing, 23-1
thumbing, I-8
Time, 15-3, A-10
timing statistics, 28-1, 28-8
token, 4-1
Tool Driver, 7-1
 BNF for script files, 7-7
 example script, 7-6
 file requirements, 7-1
 form subwindow, 7-2
 operation, 7-9
 script file, 7-1, 7-3
 subwindows file, 7-9
tool execution
 automated, 7-1
Tool.sws, 7-1
tools, B-10, I-3
Trace, 15-1
 tracepoint, 15-1
 tracepoints, III-13
trash bin, I-11
Trident 315, A-2
TTY-emulation capability, 32-1
TTYTajo, 32-13
 interfaces exported, 32-14
 program interface, 32-14
 user interface, 32-13
Type
 Executive command, 4-9
Uncaught Signal, 24-2
uninitialized variable checking, 19-5
Unload, 4-9
upArrow, 4-2, 10-3
user, I-2
user command file, I-21
user information, 6-1
user name, 6-1, 9-3
 setting, 6-1
user password, 9-3
 in command files, 9-3
user profile, 6-1, 9-3
User.cm
 AccessGroups entry, 32-12
 ArpaChat, 35-3
 ArpaFileServer, 38-2
 ArpaFileTool, 37-5
 ArpaMailTool, 39-8
 ArpaRemoteExec, 36-3
 CommandCentral, 18-2
 debugger, 24-2, 24-32, E-32
 Dictionary Tool, 2-2
 Executive, 4-10
 File Tool, 10-4

File Window, 15-3
Formatter, 20-2, 20-5
Hardcopy, 16-4
HeraldWindow, 5-2
IncludeChecker, 26-7
MFileServer, 31-2
Print, 16-4
 user profile, 6-1
User.cm entry, 3-9
Userscreen, 24-17, E-22
userscreen command, III-13
USING, 27-2
Utility Pilot client, 22-1
version stamp, 27-2, 27-3
Virtual Terminal
 ArpaChat, 35-1
 ArpaRemoteExec, 36-1
volume, A-1
 logical, A-1, A-4
 physical, A-1
window
 editable, 15-2
 empty, 15-1, 15-2
 non-editable, 15-2
Window Manager menu, I-12
windows, I-1
windowstates, I-6
word, 4-1
working directory, 4-4, 4-6, 12-2
world-swap, 24-1, E-2, E-5
write date, 9-5
Write Protect Fault, 24-2
write-protect fault, III-15
write-protected directories, II-2
XDE boot switches, B-8
XFER, 28-1, 28-2
XNS Filing protocol, 31-1
Zap, 4-9
zone
 debugging, 25-1, 25-2

Index

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 _____