Mesa User's Handbook

Version 4.0
June 1978

This handbook contains documentation for using all of the standard Mesa services intended for Mesa programmers as well as operational procedures for the Alto. In general, the sections are short and to the point, serving as a how-to guide rather than a reference document containing all of the details. This handbook assumes prior familiarity with the Mesa language as well as the Alto.

XEROX
SYSTEMS DEVELOPMENT DEPARTMENT
3408 Hillview Avenue / Palo Alto / California 94304

Table of Contents

Preface

June 1978

This handbook contains documentation for using all of the standard Mesa services intended for Mesa programmers as well as operational procedures for the Alto. In general, the sections are short and to the point, serving as a how-to guide rather than a reference document containing all of the details. This handbook assumes prior familiarity with the Mesa language as well as the Alto. All suggestions as to the form, correctness, and understandability of this document should be sent to your support group.

This documentation is divided into 4 parts. **Section 1** tells you the basics needed to get started, **section 2** lists the directories that are of interest to Mesa users, **section 3** lists various resources you should be aware of, and **appendices** A through E give further details on the compiler, binder, system, debugger, and utilities.

The style of this handbook is similar to that used in the *Mesa Language Manual.* All fine points are in this font, any word or phrase which needs to be stressed is *italicized*, user input/debugger output is `in this font`, file names are IN THIS FONT, and references to other documents are *in this font.*

Section 1: Getting Started

This section tells you all that you need to know for getting started and running a Mesa program. See the appendices for further details on the various subsystems and a sample debugging session.

1.1. Setting up your Alto disk

If you are setting up an Alto disk from scratch, either copy the standard Mesa disk maintained by your support group or obtain the command file NEWMESADISK.CM, which transfers the basic runtime files, as well as Bravo (and a Mesa USER.CM file), to your Alto disk. You also need to install the Alto Operating System Version 15/5, Executive 8, using erase option, before executing the command file; this should leave your disk with about 4000 free pages. If you just wish to get a new Mesa system on an already initialized disk, obtain the command file MESA.CM.

In either case, the basic Mesa runtime files that are transferred are: (1) RUNMESA.RUN, a BCPL program which loads the ram with the Mesa emulator, loads main memory with the kernel Mesa system, and starts execution, (2) MESA.IMAGE, the Mesa system, (3) COMPILER.IMAGE, the compiler, and (4) BINDER.IMAGE, the binder (5) XDEBUG.IMAGE, the debugger, (6) WINDEX.BCD, the window manager for (optional) use with the debugger, and (7) the system definitions files. Note that you need approximately 1400 pages for all of the Mesa files plus about 850 pages for Bravo and related files. These command files also install the debugger (and Bravo).

If the file MESAFONT.AL exists, Mesa will use it for the system display; otherwise SYSFONT.AL is used.

1.2. Installing the debugger

In order to establish the communication link between the debugger and the Mesa Executive, you must *install* the debugger. This installation is similar to installing the Swat debugger, for those familiar with that operation. Make sure your Alto disk contains the debugger, XDEBUG.IMAGE, and the window manager for (optional) use with the debugger, WINDEX.BCD.

The `Install` command may be invoked from the Alto Executive by typing `XDebug Windex/LI`, which loads the window manager (with code links) and installs the debugger. Alternatively, you may type `XDebug` to the Alto Executive, which leaves you talking to the *debugger nub* whose prompt character is "/". If you would like to include the window manager in the debugger, execute a `New - Start` sequence on the file WINDEX.BCD. If you want to load some of your own programs into the debugger, see the *Mesa Debugger Documentation* for more complete details on how this is done. When you are satisfied with the status of your debugger, issue the `Install` command; this saves the current core image of the debugger and exits to the Alto Executive.

1.3. Preparing your source file

Mesa accepts both unformatted ASCII and formatted Bravo source text files. Since the debugger uses source files to print source-text descriptions of the locus of the pc in frames and for setting breakpoints, be sure that the source files on your Alto disk are consistent with the object files.

1.4. Compiling your program

Type `Compiler` to the Alto Executive to invoke the compiler. It prompts for the source file name; when it finishes, it prompts again; a null filename (CR) returns you to the Alto Executive. Alternately, you may type `Compiler source1 source2 . . .` directly to the Alto Executive, making use of its filename completer if you wish. The compiler assumes a "`.mesa`" filename extension if it is not supplied. Compiled versions of all DEFINITIONS modules that your program uses must be on your disk.

If a syntactic error occurs, the compiler attempts to recover by deleting and/or inserting text (not in the file), displays the change(s), and tries to plow on. Semantic errors result in a symbolic print-out of the location of the error (in the form: `procedure[character-position]`) and an indication of the type of error. The semantic passes try very hard to muddle through with a complete diagnosis. The compiler puts all error messages in the file `sourcename.errlog`. When compiled successfully, the resulting object file is found on `sourcename.bcd`.

1.5. Binding your configuration

Typing `Binder` to the Alto Executive invokes the binder. It prompts for the source file name; when it finishes, it prompts again; a null filename (CR) returns you to the Alto Executive. Alternately, you may type `Binder source1 source2 . . .` directly to the Alto Executive, making use of its filename completer if you wish. The binder assumes a "`.config`" filename extension if it is not supplied.

Compiled versions of all modules in your configuration must be on your disk. The binder goes through your configuration description, `sourcename.config`, and attempts to bind the IMPORTS/EXPORTS. All error messages are put in the MESA.TYPESCRIPT file. When successfully bound, your `sourcename.bcd` file is ready to run.

1.6. Running your program

Type `Mesa` to the Alto Executive and you will find yourself talking to the Mesa Executive. At system start-up the Mesa Executive is given control in a context from which all the various system utilities are visible. At this point, you are well advised to browse through the *Mesa System Documentation* for complete details on what you can do. Basically, you must: (1) load your program -- `New` command, and (2) execute its initialization code and start execution -- `Start` command. If this fails, try putting in some breakpoints or enabling some tracing before executing step (2).

1.7. Debugging your program

In order to set some breakpoints in your program, trace program execution, display the runtime state, or interpret simple Mesa statements, you must first invoke the Mesa debugger. There are several ways of doing this. The straightforward method is to issue the Debug command to the Mesa Executive; this brings you into the debugger, ready to execute a command. If you wish to enter the debugger at any time (i.e., while your program is running), ↑SWAT interrupts your program. Once you are inside the debugger, typing "?" to the command processor gives you a list of the valid commands. The *Mesa Debugger Documentation* contains details on other ways of entering the debugger and complete documentation on all the available commands.

1.8. Reporting problems

Any requests or problems with the Mesa system should be sent to <SDSUPPORT>. Bug reports and messages that cannot be answered immediately are assigned a number and a state (open, closed, rejected, or superseded) and filed in <SDSUPPORT>CR.LOG. Whenever a change request is moved from one state to another, the originator is notified. Information about any request can always be found in the log.

Section 2: Directories

These directories are maintained on [IRIS]. Users without access to [IRIS] should consult their support group to find another host.

<ALPHAMESA>

Contains the new version of the Mesa system during the alpha test period. When the system is ready to be released, the contents of the <MESA> directory moves (temporarily) to <OLDMESA> and the contents of <ALPHAMESA> moves to <MESA>.

<MESA>

Contains the IMAGE files of interest to users of the Mesa system. It contains MESA.IMAGE (the Alto/Mesa system), COMPILER.IMAGE (compiler), BINDER.IMAGE (binder), XDEBUG.IMAGE (debugger), WINDEX.BCD (window manager for optional use with the debugger), and RUNMESA.RUN (Alto/Bcpl program which "boots" the Mesa system on the Alto).

<MESA>SYSTEM>

Contains the source and object files for the system definitions and program modules. Several packages constructed from standard Mesa system modules are also stored here.

<MESA>COMPILER>
<MESA>BINDER>
<MESA>XDEBUG>
<MESA>LISTER>
<MESA>UTILITIES>

Contains the source and object files for the the compiler, binder, and debugger, lister, and utility programs respectively.

<MESA>DOC>

Contains the documentation for the Mesa system. Both .BRAVO and .PRESS versions are maintained here.

<MESALIB>

An informal directory containing packages and independent subsystems along with corresponding documentaion. The file SUMMARY.PRESS contains a list of these packages and a short description of each.

[MAXC]<SDSUPPORT>

Contains CR.LOG, the log of change requests for the Mesa system (as explained in **Section 1.8**). Any problems with the Mesa system should be reported to <SDSUPPORT>.

Section 3: Resources

The following list enumerates resources that may be of interest to Mesa programmers.

Documents

Mesa Language Manual

Complete reference on the language, syntax, and use.

Elements of Mesa Style

Describes some of the novel features of Mesa using a number of examples oriented towards the systems programmer. It concentrates on compile-time checking, interfaces, and modularity.

Early Experience with Mesa

Discusses issues involved in using Mesa for systems programming (written by the designers of Mesa). It is recommended for those interested in the philosophy behind the language (not for the beginner).

Mesa System Documentation

Describes configurations of the Mesa system software and components which comprise them.

OIS Mesa Functional Specification

Describes the implementation of the runtime support necessary to execute Mesa programs. It assumes a Dstar machine, rather than an Alto, and is quite detailed.

OIS Processor Principles of Operation

Describes the interior architecture of the OIS System Element Digital Processor. It includes a description of the virtual storage system, the instruction set, and the input-output facilities.

Mesa Debugger Documentation

Describes the current release of the Mesa debugger.

Debugger - Extended Features

Describes some extended features of the Mesa debugger: FTP command, user invoked procedures, and the window manager (WINDEX).

Files

<MESA>MESA.signals
<MESA>BASICMESA.signals
<MESA>BINDER.signals
<MESA>COMPILER.signals
<MESA>XDEBUG.signals

Lists the uncaught signal names (and global frame addresses) for various Mesa components.

<MESA>USER.CM

A USER.CM file set up with the Mesa Bravo macros, GACHA10 for the editing font, and minimal printing fonts.

<MESA>NEWMESADISK.CM
<MESA>MESA.CM

The command files used for setting up a basic Mesa disk (as described in **section 1.1**).

[MAXC1]<SECRETARY>MESAUSERS.dl

Distribution list for messages to the Mesa user community. If you wish to get on this list, talk to your secretary.

Other materials

There have been a series of videotapes prepared which describe various features of the language and runtime environment. See a member of your support group for further details on the tapes that are currently available and where to get them.

Appendix A: Compiler

The Mesa compiler translates Mesa source files into corresponding object files. An object file contains the executable code for the module (if any) plus 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 appendix describes the operation of the compiler, including the compile-time options and messages.

Preparing Source Files

The compiler accepts ASCII text files. In a source file, any sequence of characters that begins with a ↑Z is skipped up to (but excluding) the next carriage return (or end of file). This convention accomodates Bravo formatting codes. You may use such formatting in your source files as you see fit. Note, however, that Mesa *does not* interpret any information about fonts, position, etc., attached to source text that it displays (in, e.g., identifying the location of an error or breakpoint).

The recommended extension for naming any Mesa source file is ".Mesa".

Standard Bravo macros useful during the editing and compilation cycle are described later.

Running the Compiler

The compiler takes commands either from the command line or interactively from the keyboard.

> To enter interactive mode, type just "compiler" to the Alto Executive. The compiler will prompt you for commands. You can correct a command during typein by using the usual set of editing characters. To exit from the compiler, respond to the prompt with just a carriage return.
>
> To invoke the compiler specifying command line input, follow "compiler" with a list of commands, separated by spaces. In this mode, you can use the Executive's file completion facilities to build the command list, and *all* input is taken from the command line.

The simplest form of command is just the name of a source file to be compiled. If you supply the command `sourcefile` with no period and no extension, the compiler assumes you mean `sourcefile.Mesa`.

During compilation, the display is turned off and a die is displayed in the cursor. The number on the die identifies the pass of the compiler that is running. This allows you to check the progress of the compilation and also provides useful feedback to the maintainers of the compiler when something goes drastically wrong.

Fine point:

Don't confuse the compiler's display with DMT's.

The compiler reports the result of each command with a message having one of the following forms (each * is replaced by an appropriate number; bracketed items appear only when relevant):

file.mesa -- source chars: *, time: *
[code bytes: *, links: *, frame size: *]
[* warnings on *file*.errlog]

Compilation was successful. The object file is *file*.BCD. For a DEFINITIONS module, the middle line is not meaningful and is 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. The 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 remains valid). Otherwise, the object file was invalidated and will be rejected by the binder and loader.

File error

The compiler could not find the specified file.

If you are providing commands interactively, these messages appear on the Alto screen after each command is completed. Otherwise, they are written into the file MESA.TYPESCRIPT. In the latter case, the compiler will process the entire command line; then, if any error or warning messages were issued, it brings this to your attention with a message of the following form:

Errors [and Warnings] logged; type any character to finish.

The compiler will not return to the executive or run another subsystem until you acknowledge the message. (You can change this behavior by using switches, which are described next.)

Compiler 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

a	compile code for an Alto (default)
p	pause after compiling file if there are errors
r	terminate compilation and run the program contained in file
s	sort global variables and entry indices (default)

w	log warning messages (default)
x	prepare a cross-reference file

Each switch has a default setting, The command `sourcefile` is equivalent to `sourcefile/a~psw~x` if you use the standard defaults, i.e., the compiler generates code for an Alto (not a Dstar), does not pause after compiling `file`, sorts variables, logs warning messages, and does not produce a cross-reference file. Note that the "r" switch changes the interpretation of `file`, which should name a subsystem when used with this switch.

You can also change the default setting of any switch by using the "c" switch. The text preceding a "/c" is interpreted as a switch specification (designating a single switch only) and it establishes the default setting for that switch. Unless overridden or reset, that default applies to all subsequent commands.

Here is some more information about the options:

`a[lto]`

Generate code for Alto (`a`) or DStar (`-a`) hardware. This switch primarily affects the treatment of long pointers in the object code.

`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.)

`w[arnings]`

Log (`w`) or ignore (`-w`) certain legal but suspicious usage that can be detected by the compiler.

`x[ref]`

Generate (`x`) a file `sourcename.XRJ` containing cross-reference information for the file being compiled (`sourcename.Mesa`). The file requires post-processing by separate utility programs before it is useful.

Examples:

`foo`

Compile `foo` using all the default switch settings (standard or established by preceding "/c" switches).

`foo/-wx`

As above, but suppress warning messages and generate `foo.XRJ`.

The `p[ause]` switch requires special comment. You can use it to control progress through a sequence of files specified on the command line. As a global switch (set using "/c"), it specifies pausing (`p`) or not pausing (`-p`) just before exiting from the compiler. 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* contained errors; moreover, any remaining commands are ignored. The local default is not to pause but to continue with the next command.

Examples:

`compile -p/c 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.

`compile 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.

Context Switching and Bravo Macros

If you are a Bravo user, you might find the following macros useful for switching between Bravo and the Mesa compiler. They are included in <MESA>USER.CM.

`bravo/m filename`

This invokes Bravo with two windows, gets `filename.mesa` in the top window and gets `filename.errlog` in a smaller, bottom window. (Be sure not to use `filename.mesa` on the command line.)

`bravo/j filename` *octalNumber*

This invokes Bravo and gets `filename.mesa`. It also selects the character position corresponding to the octal number and normalizes the selection. This is useful when the source text printed with an error message does not supply enough context to locate the error; each error message also includes the octal number needed by this macro.

`q[uit]/m`

This Bravo command writes out the file in the selected window (say `filename.mesa`) and terminates Bravo. It then specifies the following sequence of (Executive) commands:

```
delete filename.errlog
compile filename
bravo/m filename
```

The command line switch "/r" (run) causes the compiler to terminate by running some other program instead of returning to the Alto Executive. You may specify either a ".image" or a ".run" file. If you omit the extension, ".image" is assumed. Any switches after the "r" and any other text remaining in the command line after the command specifying this switch are copied to the file COM.CM for inspection by the new program. The facility is primarily intended for use in (program generated) command files.

Examples:

```
Compiler sourcefile Mesa/r sourcefile
```

Compile `sourcefile`; then invoke `mesa.image` to load and start `sourcefile.bcd`. Note that "`Compiler sourcefile; Mesa sourcefile`" has the same effect but is slower, because it returns to the Alto Executive before invoking Mesa. (There are overheads of several seconds associated with both restarting the Executive and reestablishing the Mesa environment.)

```
Compiler sourcefile Ftp.run/r Iris store sourcefile.bcd
```

Compile `sourcefile`, then store the object file on Iris. Note that you must supply the "`.run`" and ".bcd" to invoke Ftp in this way.

Fine point:

You can run Bravo using the "/r" switch, but the current version (7.1) will not correctly find switches or arguments on the command line.

Error Messages

The compiler writes error and warning messages for `sourcefile.mesa` on `sourcefile.errlog`. 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. With the exception of "correctable" syntactic 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.

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;
```

```
?+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 octal 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.

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 by character index (the octal number) and 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. 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.

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. If you get such a message (or encounter other compiler problems), you should submit a change request (CR) as described in **Section 1.8**. Be sure to preserve the relevant files and to mention the octal codes identifying the pass (*n*), signal (*s*) and message (*m*) in your CR.

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 required by a procedure or program cannot exceed 4096 words.

The compiler allocates its internal tables dynamically and tries to adjust their relative sizes to accomodate the program being compiled. When it is unsuccessful, it reports failure with a message of the form:

`Storage Overflow in Pass` *n*

You must split your program into two or more smaller modules.

Appendix B: Binder

The Mesa binder combines modules and previously bound configurations to produce a new configuration. The *Mesa Language Manual* documents the syntax of a *configuration description* which describes the desired configuration to the binder. The output of the binder is a *binary configuration description* (BCD) which may be loaded into a running system or processed by a later invocation of the binder. This section will discuss the operation of the binder including the binding time options and switches.

File Organization

In order to understand the options described below, it is necessary to understand something about how configurations exist in files. The BCD file produced by the binder normally contains only the compiled description of the configuration. It does not contain any code or symbols. For each module instance in the configuration, the BCD specifies the location of the code and symbols by file name (and time stamp), starting page, and number of pages. Thus the code and symbols for a configuration may be scattered over a large number of files. It is possible to put the BCD, the code, and the symbols in the same file (this is the way BCDs are generated by the Mesa compiler).

While debugging, the "normal" mode of operation is not to copy code or symbol segments to another file (the default; no switches), but to leave them in the files generated by the compiler. This saves disk space and requires the least binding time.

For distribution, code and/or symbols can be copied into the output file by using the corresponding switch on the source file name (not on the output file name). Alternately, they can be copied into different code or symbols files by giving the file name and switch following the source file name.

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. (The FETCH program and the debugger's `ATtach Symbols` command can be used to get symbols for individual modules during debugging.)

There is also an option for compressing the symbol tables as they are copied. In this mode, only public symbols declared in the global frame (plus all procedures and signals and their parameters and results) are included. Private symbols and variables local to procedures are not copied. This option allows limited but usually adequate debugging, and will substantially reduce the size of the symbols file (typically by 50%).

Fine point:

Copying code into a file other than the BCD file is supported, but probably not useful.

Running the Binder

The binder takes commands either from the command line or interactively from the keyboard. Commands are of the form

```
source[/s file/s file/s]
```

where [] indicates optional parts. The valid switches are

```
/d  - enter debugging mode
/c  - copy code segments to this file
/o  - give this name to the output BCD file
/s  - copy symbol segments to this file
/x  - copy compressed symbols to this file
/p  - pause before proceeding if there are errors
/r  - run the specified program
/g  - (go) begin processing the preceeding files
```

A switch specified with a null file name is a global switch. A switch letter may be preceeded by "-" to negate its effect. The only switch with either of these properties is the /p switch. The binder will pause after completing all commands if any errors were reported. Applying the /p switch to an individual source may cause a pause earlier as well.

Normally a command to the binder is terminated with an end-of-line. In order to specify more than one command using command line input, the /g switch (for "go") may be used to replace the end-of-line. Simply add the /g switch to the last file name of each command. (This option is not available when input is from the keyboard.)

The first file name is always the source configuration description. The last occurance of a /c, /o or /s file will prevail, and extra filenames are ignored. Default extensions are "config" for source, "bcd" for output, "code" for code and "symbols" for symbols. Default output is to source.bcd. Examples:

foo

Read foo.config and write the resulting BCD on foo.bcd. This is the "normal" debugging mode since it is the fastest and requires the least disk space.

foo/c

Read foo.config, write foo.bcd. Copy all code segments into foo.bcd. Leave all symbol segments as they were in the input files. This is a possible "distribution" mode.

foo/cs

Read foo.config, write foo.bcd. Copy all code and symbol segments into foo.bcd. This is also a possible distribution mode, if debugging will be required.

foo/c foo/x

Read foo.config, write foo.bcd. Copy all code segments into foo.bcd; compress all symbol segments into foo.symbols. By packaging all of the symbols in a single file, you minimize the risk of getting an incorrect version of some symbol table.

foo.cd/c foo.sym/s foo.bound/o

Read foo.cd, write foo.bound. Copy all code segments into foo.bound and all symbol segments into foo.sym.

foo.cd/c foo.sym/sg bar/c

Read foo.cd, write foo.bcd. Copy all code segments into foo.bcd and all symbol segments into foo.sym. Then read bar.config and write bar.bcd. Copy all its code into bar.bcd.

/-p foo/g bar/cg dum

Bind foo, bar, and dum and will not pause even if there are errors.

foo/g bar/cpg dum

Bind foo, bar, and dum as usual and, in addition, stop after bar if it contains errors.

Because of the large number of options available, it is doubly important to maintain file consistency. Appropriate version checks are included in the binder, the loader, and the debugger.

Context Switching

The command line switch /r (run) is used to specify that the Binder should run some other program rather than returning to the Alto Executive. Both ".image" and ".run" files may be specified. If there is no explicit extension, ".image" is assumed. Any switches after the r and any other text remaining in the command line after the file with the /r switch will be copied to the file COM.CM for inspection by the new program.

Examples:

Binder SomeConfig/g Mesa/r SomeConfig

will bind SomeConfig and then run Mesa.image as if you had typed Mesa SomeConfig.

Binder SomeConfig/g Mesa/rd OtherConfig/-s SomeConfig

will bind SomeConfig and then run Mesa.image as if you had typed Mesa/d OtherConfig/-s SomeConfig.

Binder SomeConfig/cg Ftp.run/r Store SomeConfig.bcd

will bind SomeConfig copying the code and then run Ftp.run as if you had typed Ftp.run Store SomeConfig.bcd.

Fine points:

The last specification before the file with the /r switch must have the /g switch to indicate the end of the previous command.

You can run Bravo using the /r switch, but the current version (7.1) will not find switches (or arguments) on the command line.

Error Messages

The binder reports error and warning messages on the display and in the file MESA.TYPESCRIPT. If possible, the binder will indicate the offending source line and configuration name with each error. Some of the common error messages are:

`foo is undeclared (in baz)`

> The module `baz` is trying to import the interface (or program) `foo` but `foo` is neither imported from a higher lever configuration nor exported by any module or configuration at the same level.

`foo does not name a module or configuration`

> 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.

`item nnn in interface foo is unbindable`

> (Warning) Item number `nnn` in the interface `foo` has no implementation. You can count (from 0) the interface items in `foo` or use the lister's `Interface` command to get more information.

`foo referenced in different versions`

> (Warning) Two different versions of the named file are referenced by the modules being bound. This will produce another error message if you attempt to match the two versions as import and export.

`foo cannot be imported as baz`

> `foo` is the interface (file name and version) which is available for import (or being passed as a parameter), but the importer is asking for `baz`. The source line shows the importer.

`foo cannot be exported as baz`

> The source line shows an exporter of `foo` who trying to assign the interface (implicitly or explicitly) to `baz`. This may be a version problem (if the names are the same) or an error in an assignment.

`foo is not imported by any modules`
`foo 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.

`Errors detected, BCD not written`

> The binder has produced no output.

`Errors detected, BCD is invalid`

Errors were discovered after the binder had started writing the output file. The file has been made invalid so that neither the binder nor the loader will accept it as input.

`Type any character to exit`

The binder will normally pause before returning to the Alto Executive (or running another program) if there were any errors detected. To turn this global pause flag off, use the switch `/-p` with a null file name.

`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.

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 BCD file can be renamed; the code and symbols files cannot (since the BCD contains the names of these files in its internal tables).

Copying code and symbols into the same file (other than the BCD file) is not implemented.

Multiple instantiations of nested configurations are not implemented. You can get around this by binding the nested configuration in a separate step.

Estimated running time: five seconds for initialization plus one-half second per included file (module or configuration). Add one second per module to copy code and one second per module to copy symbols.

Appendix C: System

Mesa systems are available in both *standard* and *basic* configurations. The basic configuration's only interface is the command line. BCDs may be loaded and started by specifying them on the command line. The standard configuration contains the *Mesa Executive* which serves as the user interface. See the *Mesa System Documentation* for details. The standard configuration also allows command line loading.

Command line loading

Both the standard configuration and the basic configurations allow clients to load their BCDs by specifying them on the command line. The general form of the command line is:

```
>Mesa[/d] file1[/sw] file2[/sw] . . .
```

The valid switches are listed below. A '- preceeding the switch inverts the meaning.

> `/d` -- go to the debugger after loading this BCD but before starting it. This is the only switch applicable to the image file.
>
> `/s` -- start the BCD (default if non-null control module).
>
> `/l` -- load the BCD with code links. The /l switch is also applicable to the New command of the Mesa Executive. The modules will only have code links if there is room for the links in the code and the modules specify that they want code links.

The default extension is ".bcd". There are no global switches. All switches only apply to the file to which they are attached. If BasicMesa runs out of things to load from the command line, it returns to the *Alto Executive*. If Mesa runs out, the *Mesa Executive* is given control.

Examples:

```
>BasicMesa WindowPackage/l-s SomeConfig/d
```

> Start `BasicMesa` and load the `WindowPackage` with code links but don't start it. Then load `SomeConfig` and go to the debugger before starting it.

```
>Mesa AConfig/-s StrangeConfig.foo/-s
```

> Start `Mesa` and load `AConfig.bcd` and `StrangeConfig.foo` without starting either and then enter the *Mesa Executive*.

Error Messages

Errors generated during loading or interaction with the *Mesa Executive* are reported by displayed messages and by uncaught signals in BasicMesa. The following error messages are given by the *Mesa Executive*:

`!File: file`

When attempting to load a BCD, *file* cannot be found or is an invalid BCD. If *file* is not the BCD being loaded, then it is a code file for the BCD. A BCD may be invalid because it is was invalidated by either the Compiler or Binder due to errors in its construction, or because it was produced by a previous version of the system.

`!Number`

An invalid number was typed.

`!String too long`

A string was typed that was too long.

`!File name referenced in different versions`

When loading a BCD, the interface or program *name* was referenced in different versions. Loading is continued but there may be unbound external references.

`External Debugger not installed, type DEL to abort`

An attempt was made to invoke the debugger but it has not been installed.

The signals that may be generated by BasicMesa are listed below. See BASICMESA.SIGNALS for the corresponding signal values.

`BadFile[name]`

When attempting to load the BCD *name*, either it cannot be found, it is an invalid BCD, or a code file in the BCD is not available.

`VersionMismatch[name]`

When loading a BCD, the interface or program *name* was referenced in different versions.

Appendix D: Debugger

The common facilities available in the Mesa debugger include setting breakpoints, tracing program execution, displaying the runtime state, and interpreting Mesa statements. It will be easiest to understand how to access these facilities by going through a simple example using many of the common commands.

Command line installing

To install the debugger from the command line of the Alto Executive, use the "I" switch; use the "L" switch to load programs with code links (to save space).

For example, typing `XDebug WindEx/i` installs the debugger with the window manager (WINDEX.BCD); typing `XDebug WindEx/il` installs WINDEX with code links.

Files

The debugger itself is contained in the file XDEBUG.IMAGE; the window manager is WINDEX.BCD. There are several other files that are used by the debugger and *should not be deleted from your disk*. These are the swapping files used by the debugger: SWATEE (to hold the user's core image) and MESADEBUGGER (to hold the debugger's core image), and the DEBUG.TYPESCRIPT file, used as a log of your debugging session.

Signals and errors

See the *Mesa Debugger Documentation* for details on the common signal and error messages you might receive and suggestions for recovery.

Sample program

The configuration we are going to use as an example is taken from the *Mesa Language Manual* (chapter 7). These files may be found on [IRIS]MESA>DOC.

The simple configuration **Config2** consists of two modules, **Lexicon** and **LexiconClient**. After the modules have been compiled and the configuration has been successfully bound, you are ready to load and debug the program.

Entering the debugger

Let us assume that the configuration has been loaded (but not started) and you have entered the debugger for the first time (via the `Debug` command or the "D" switch). You get a herald that indicates when your version of the debugger was built followed by the current date and time and a prompt for the first command:

```
Alto/Mesa Debugger 4.0 of 25-May-78 16:31
 6-June-78 17:27

>
```

Setting the context

In order to get to a context from which you can set breakpoints in one of the modules in **Config2**, let's look to see which configurations have been loaded by saying:

```
>List Configurations [confirm]
```

which responds with:

```
Config2
Mesa.
```

If we check the current context at this point, you can see that the current configuration is **Mesa**,

```
>CUrrent context --
  Module: NubControl, G: 172234B, L: 167230B, PSB: 2770B
  Configuration: Mesa.
```

We need to set the current configuration to be **Config2**,

```
>Set Root configuration: Config2
```

and find out which modules are in this configuration,

```
>Display Configuration Config2
Lexicon, G: 150404B
LexiconClient, G: 150420B.
```

Now we can set the context to be **Lexicon**, so that we can set some breakpoints,

```
>Set Module context: Lexicon.
```

Using windows

Let us assume that you have loaded the window manager (WINDEX) with your debugger; this allows you to position and change the size of the windows as well as to set breakpoints and to select text to be used as type-in. The left margin of a window is used for scrolling; use the red mouse button to scroll up; the yellow button to thumb; and the blue button to scroll down. The rest of the window is a text area; you can make character selections by clicking the red mouse button, word selections with the yellow button, and display the menu with the blue button. See the *Extended Features Memo* for complete details on WINDEX.

Setting breakpoints by selections

Let's load the source text for **Lexicon** into a window so we can set breakpoints using selections. This may be done in one of two ways: either display the stack and ask to see the source (this loads the source file for the current module into the sourcefile window of the debugger),

```
>Display Stack
Lexicon G: 150404B  >s
 Source:  <>--Lexicon.mesa  June 6, 1978 5:20 PM
  >q.
```

or you can create a new scratch window and type-in the name of the file followed by escape (ESC) (this loads the source file into the window you have just created).

Now move into the window containing the sourcefile, and click a mouse button to make it the current window. Suppose we want 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 this procedure (by using the yellow button for word select mode). Hold down the menu button and choose the `SetBr` command. This sets a breakpoint on the exit of the procedure (similarly selecting the word PROCEDURE sets a breakpoint on the entry to the procedure).

Suppose you want to set a breakpoint in the end of one of the conditional IF-THEN-ELSE statements in the procedure **InsertString.** Select any character in the statement ELSE **n.llink ← NewNode[];**. Confirmation that the breakpoint has been set is given by the moving the selection to ELSE **<>n.llink ← NewNode[];** (Note that in all cases, the closest enclosing statement is the place at which the breakpoint is actually set).

Setting breakpoints by type-in

You may also set breakpoints in the program by means of typing-in the command. This gives you the added capability of specifying a condition that must be satisfied for the breakpoint to be taken. If, for instance, you want to set a breakpoint on the entry to the procedure **FindString**, and invoke the debugger only if **root** is not NIL, you can do this by saying:

```
>Break Entry Procedure: FindString, condition: root # NIL.
```

Inserting comments

Saving some comments along with the commands is a good idea so that it is easier to remember what happened when looking back at the typescript file. For instance you might now say,

```
>--This breakpoint was set to skip checking for a lexicon if we
>--know the tree is empty.
```

Proceeding

It is now time to proceed and start the program. This is done by executing the following command:

```
>Proceed [confirm]
```

and Starting the program.

If we try to add the lexicon "xxxxx" to the tree, we will then reach our breakpoints.

Examine and change the state

You next enter the debugger at one of the breakpoints with the herald:

```
>Break at exit from NewNode, L: 165034B (in Lexicon, G:150404B)
```

to indicate where you are. At this point you might display the stack and look at some variables,

```
>Display Stack
NewNode, L: 165034B (in Lexiocn, G:150404B)  >v
   n=164333b↑
        >q
```

or look at the several levels of the stack,

```
>Display Stack
NewNode, L: 165034B (in Lexicon, G:150404B)  >n
InsertString, L: 165044B (in Lexicon, G:150404B)  >n
AddString, L: 165054B (in Lexicon, G:150404B)  >n
LexiconClient, L: 171664B (in LexiconClient, G:150420B)  >n
No symbols for NubControl, G: 172234B, L:172060B, PC: 314B, E
  >q
```

or ask to see what the node n (in NewNode) looks like (by using the interpreter),

```
> n↑
Node[llink:NIL, rlink:NIL, string:(5,5)"xxxxx"].
```

Let's say we wanted to set both the left link and right link of n to point to itself and then check the values. This may be done by saying,

```
> n.llink ← n ; n.rlink ← n ; n ; n↑
```

which responds with,

```
164333B↑
Node[llink:164333B↑, rlink:164333B↑, string:(5,5)"xxxxx"].
```

If at this point we want to see the value of the variable **ch** in the module **LexiconClient** (a variable in the current configuration but not in the current context), this may be done by saying,

```
>Find variable: ch
```

which responds with the first character of the last lexicon that was typed,

```
'x (in LexiconClient)
```

More breakpoint commands

The following command lists all of the breakpoints that have been set:

```
List Breaks [confirm]
```

If you decide that you are no longer interested in any of these breakpoints you can

```
Clear All Breaks [confirm]
```

which removes all breakpoints and restores the instructions.

Look at the user world

If you are interested in seeing the state of the user display, you can look at the user world by saying

```
Userscreen [confirm]
```

When you are done, hitting the **SWAT** key returns you to the debugger, ready to execute more commands.

Setting tracepoints

Suppose next you want to set a trace on the entry to the procedure **LexicalCompare** so that you can simply see the two strings being compared and go on. You may do so by saying,

```
>Trace Entry Procedure: LexicalCompare.
```

Now we should proceed and try to add a new lexicon, say "yyy".

When the tracepoint is reached, you get a dump of the input parameters of **LexicalCompare** along with the herald:

```
Trace at entry to LexicalCompare, L: 171674B (in Lexicon, G:150404B)
  s1=(3,80)"yyy"
  s2=(5,5)"xxxxx" >
```

at which point you may continue executing your program (respond with Q) or enter the debugger command processor (respond with B).

This represents a brief introduction to the use of most of the debugger's commands that you will commonly need. See the *Mesa Debugger Documentation* for further details; the best teachers are experienced Mesa programmers and lots of practice!!!

Appendix E: Utilities

Described below are several utility packages that have proved useful in building Mesa systems. The *Lister* produces human readable listings of various Mesa file formats. The *IncludeChecker* checks for object file consistency. The *Statistics* package generates source and object statistics. *Version* lists creation dates for source and object files. The *SignalLister* produces a mapping of signals and signal values.

Lister

The Lister produces listings of code, symbols, bcds, etc. from object and source files. To use it, retrieve <MESA>LISTER.IMAGE. It operates in either command line or keyboard mode. Commands look like procedure calls with constant (string, numeric, character, boolean) arguments. Arguments are type checked by the command interpreter. In command line mode type to the Alto Executive:

```
>Lister command1[arg1, arg2, ...] command2[arg1, ...]
```

You actually type the square brackets, as in a Mesa procedure call.

In keyboard mode you just type the command with arguments. Typing the ESC key will extend the command name if a unique command exists. The Lister will prompt for arguments if the command name is terminated with CR. Typing ? in keyboard mode will produce a list of available commands and their arguments. The current commands are:

`Code["Filename"]`

Given a bcd file produced by the compiler, this command will produce a listing (on `Filename.cl`) of the object code. If the source file is available on your disk, the source for each statement will be listed just before the object code.

Warning: This command produces a large amount of output.

`OctalCode["Filename"]`

Same as the `Code` command, except that opcodes are given in octal as well as by name.

Warning: This command produces a large amount of output.

`OpcodeList["Filename"]`

Generates on `Filename.list` a one page (Gacha8) listing of the Mesa opcodes.

`Bcd["Filename"]`

This command produces (on `Filename.bl`) a listing of the internal tables of the binary configuration description. Output of either the compiler or binder is acceptable.

`BcdLinks["Filename"]`

Same as the `Bcd` command, expect that the control links of imported and exported items are included.

`BcdSegment["Filename",Base,Pages,Links]`

The most general form of the `Bcd` command allowing you to specify the location of the bcd by filename, starting page number, and number of pages. Specify TRUE or FALSE for `Links`.

`Interface["Filename"]`

Given the bcd file for an interface (DEFINITIONS file), this command will produce (on `Filename.il`) a list of the interface items and numbers. These numbers are the ones reported by the Binder for unbindable items.

`Symbols["Filename"]`

Given a compiler output bcd, this command will list the internal symbol table.

`SymbolSegment["Filename",Base,Pages]`

A more general form allowing complete specification of the location of the symbols (e.g. in a `.symbols` file).

`Load["Filename"]`

An escape hatch allowing other programs to run in the Lister environment. NEWS and STARTS the module `Filename`.

`Debug[]`

Invokes the Debugger.

`Quit[]`

Returns to the Alto Executive.

Include Checker

The IncludeChecker is a program that checks the include relationships in bcds for consistency. Its output is in three sections. The first is a compilation order for those files checked that is alphabetic as much as possible while satisfying the include relationships. The second is a list of include relationships, listing those files each module includes and the time stamps of the files. Any inconsistancies are flagged with an asterisk. The final list is the included relationships, listing those files which include a particular file. To use it retrieve <MESA>UTILITIES>INCLUDECHECKER.BCD. It gets all of its parameters from the command line and is started by typing to the Alto Executive:

```
>Mesa IncludeChecker outputfile [filename1 filename2 . . .]
```

where

`outputfile` is the name of the file the output is written on. If no extension is given, "`.list`" is assumed.

the list of filenames specifies those bcds that are to be checked. If no files are specified, all bcds on the disk are examined.

For example, the following command line will produce the output shown below on file

Foo.list.

```
>Mesa IncludeChecker Foo Allocator AltoDefs FspDefs InlineDefs MopCodes
SystemDefs TableDefs

Compilation Order:
 MopCodes InlineDefs AltoDefs FspDefs SystemDefs TableDefs Allocator

Allocator (12-May-78 17:19:32 #5 #20) includes
  InlineDefs (11-Apr-78 18:21:55 #5 #143)
  SystemDefs (11-Apr-78 18:30:25 #5 #143)
  TableDefs (11-Apr-78 18:22:24 #5 #143)

AltoDefs (11-Apr-78 18:20:32 #5 #143) includes nothing

FspDefs (11-Apr-78 18:26:53 #5 #143) includes
  AltoDefs (11-Apr-78 18:20:32 #5 #143)

InlineDefs (11-Apr-78 18:21:55 #5 #143) includes
  MopCodes (11-Apr-78 18:21:02 #5 #143)

MopCodes (11-Apr-78 18:21:02 #5 #143) includes nothing

SystemDefs (11-Apr-78 18:30:25 #5 #143) includes
  FspDefs (11-Apr-78 18:26:53 #5 #143)

TableDefs (11-Apr-78 18:22:24 #5 #143) includes
  AltoDefs (11-Apr-78 18:20:32 #5 #143)

Allocator is included by nothing

AltoDefs is included by
  FspDefs
  TableDefs

FspDefs is included by
  SystemDefs

InlineDefs is included by
  Allocator

MopCodes is included by
  InlineDefs

SystemDefs is included by
  Allocator

TableDefs is included by
  Allocator
```

Statistics Package

This package gathers statistics about Mesa source and object files and writes them on MESA.TYPESCRIPT. It may be invoked either interactively of from the command line. It may be invoked from the command line by typing the following to the Alto Executive:

```
>Mesa Statistics filename[/switches] . . .
```

Output is to the display and to MESA.TYPESCRIPT. If no filenames are specified on the command line, Statistics enters the interactive mode. Type ? to get full documentation. The follow switches are used:

- /b -- bcd statistics (default).
- /c -- command: use filename as switch.
- /d -- invoke debugger.
- /h -- print heading.
- /m -- source statistics (default).
- /s -- print subtotal.
- /t -- print total.
- /x -- "Management" statistics.

The following command line will generate the output shown below:

```
>Mesa Statistics AlFont DisplayControl StreamIO SystemDisplay t/c

Alto/Mesa 4.0 of 26-May-78 12:31
 8-Jun-78  8:58
>Statistics -- 170330B


Alto/Mesa Statistics Package

Statistics as of  8-Jun-78  8:59:01

                chars lines codebytes framesize ngfi nlinks codepages sympages
                -------  ------  -------  ------  ---  ------  -----   ----

AlFont           6136     212      552       4     1       6       2      11
DisplayControl   6754     197      684      50     1      26       2      14
StreamIO         6404     263      876       6     1       7       2      10
SystemDisplay   15834     526     1778      63     1       8       4      20
                -------  ------  -------  ------  ---  ------  -----   ----

TOTAL:          35128    1198     3890     123     4      47      10      55
```

Version

Version is a program which displays time stamp information for files with any of the following extensions: mesa, config, bcd, image, symbols and code. For source files (mesa and config extensions) it reads the first page of the file and trys to find a valid date. For binary files (bcd, image, symbols and code) it knows where to find the time stamp that Mesa uses for version checking. When given a file name root, Version searches the disk for files with that root and the above extensions, displaying the time stamp for each file found.

Version may be run either from the command line or interactively. To run Version from the command line type the following to the Alto Executive

```
>Mesa Version [filename1 filename2 . . .]
```

To run Version interactively omit the list of filenames in the above line. Version will then prompt for input.

Sample output is shown below.

```
Alto/Mesa 4.0 of 26-May-78 12:31
 8-Jun-78  8:35
>Version -- 170324B

Mesa
 config: 30-Mar-78 16:47:00
 symbols: 26-May-78 12:29:56 5#20B#
 image: 26-May-78 12:31:24 5#20B#

AltoDefs
 mesa: 25-Jan-78 17:49:00
 bcd: 11-Apr-78 18:20:32 5#143B#
```

SignalLister

SignalLister is a program which will produce a signal listing for an image file, like MESA.SIGNALS. To produce a signal listing for FOO.IMAGE, type to the Alto Executive:

```
>Mesa SignalLister Foo
```

The signal listing will be produced on file FOO.SIGNALS. If the symbols for a module are not available, no signals for that module are listed. For example, if FOO.IMAGE was made by loading FOO.BCD on top of MESA.IMAGE, a complete signal listing for FOO.IMAGE will require that MESA.SYMBOLS and all the symbols for FOO.BCD be on the disk. If MESA.SYMBOLS is not present, only those modules from FOO.BCD will have their signals listed.