Inter-Office Memorandum

To	Mesa Users	Date	October 27, 1980
From	Brian Lewis, Jim Sandman, Dick Sweet	Location	Palo Alto
Subject	**Mesa 6.0 Utilities Update**	Organization	SDD/SS/Mesa

XEROX

Filed on: [Iris]<Mesa>Doc>Utilities60.bravo (and .press)

This memo outlines the changes made in the utility packages since the last release (Mesa 5.0, April 9, 1979). More complete information can be found in the *Mesa User's Handbook.*

Major changes include some new commands in the Lister and an extensively reworked IncludeChecker. In addition, there is a version of the SignalLister that reads `.bcd` files.

Lister

The lister is now available only as a `.bcd` file. The user interface has been changed slightly: commands that take string parameters no longer need string quotes. The command scanner takes all characters up to the next comma or right bracket as the parameter. Thus

`Code[ListerRoutines]` and `Code["ListerRoutines"]`

are equivalent. The new commands (and "improved" old ones) are listed below. Note that several of the new commands (and some of the old ones) are useful only for internal (Compiler) debugging.

`CodeInConfig[`**config, module**`]`
`OctalCodeInConfig[`**config, module**`]`

> **Config** names a bound configuration; **module** is a module within that configuration. A code listing is produced for the module (see the `Code` and `OctalCode` commands). This is of particular interest for packaged configurations where the code has been rearranged among segments and code packs.

`CompressUsing[`**file**`]`

> The named file should contain a list of BCD file names. 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 (see below). The output is written to `file.ul`.

`Hexify[], Octify[]`

> The code lister normally prints addresses (and opcodes for "octal" listings) in base eight. For microcode debugging, base sixteen is sometimes preferable. `Hexify`[] puts the code lister into hexadecimal mode; `Octify[]` reverts to octal mode.

`Implementors[`**file**`]`

The named file should contain a list of compiler output BCDs (interfaces and program modules). This command creates a file, `File.iml`, showing for each interface exported by any program in the list, where the various interface items are implemented. If the list also includes the BCD for a particular interface, the interface items not implemented by any program are also shown. In order to run this command, one needs not only the BCDs in the list, but also the BCDs for the interfaces exported by the programs therein. Missing BCDs are reported and the command attempts to forge on.

`Stamps[`**file**`]`

File names a compiler or binder output BCD. This command generates a file, `File.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[`**file**`]`

File names a compiler or binder output BCD. This command examines all of the exported interfaces and enumerates interface items in those interfaces that are not exported by this module or configuration.

`Using[`**file**`]`

Given a compiler output BCD, this command generates a directory statement with its included identifier lists (on `file.ul`). Since there is not enough information in the BCD to tell which symbols were implicitly included, the USING clauses will contain a superset of those items actually needed. The Mesa 6 Lister does a much better job of weeding out extraneous names in the USING clauses.

`Version[`**file**`]`

File names a compiler or binder output BCD, or an IMAGE file. This command shows, on `Mesa.typescript`, the object, source, and creator version stamps of the file.

`XrefByCallee[`**file**`], XrefByCaller[`**file**`]`

File names a file that contains a list of BCD file names. For each module in the list, a scan is made of the code to find all procedure calls. The <caller, callee> pairs are then sorted by either caller or callee. These commands produce output on `file.xle` and `file.xlr`, respectively.

There are three kinds of procedure calls: local, external, and stack. The program can figure out which procedure is being called for local and external calls. Stack function calls are used for procedure variables (e.g., **stream.get[...]**) and for nested procedure calls. The program ignores nested calls and indicates a callee of `*` for procedure variables.

Include Checker

The most significant differences between this version of the IncludeChecker and the one released with Mesa 5.0 are the following:

1. It handles more files, and requires less processing time for large numbers of files.

2. It executes either from the command line or interactively.

3. It obtains the creation dates for source files from their leader page, rather than from the first few lines of the source text (however, see the description of the new switch `/t` below).

There have been other minor changes to command syntax. The entire section of the *Mesa User's Handbook* on the IncludeChecker is included below.

The IncludeChecker is a program that examines a collection of Mesa source and BCDs for consistency. It produces an output listing that gives a compilation order for the files, and for each BCD, a list of all the BCDs that it includes, and a list of the BCDs which include it. Any inconsistencies (which are described below) are flagged in this listing by an asterisk. As an option, the IncludeChecker will also generate a compilation command on `Line.cm` that can be executed to make the files consistent.

The IncludeChecker determines that an inconsistency exists among the input files if either:

1. A BCD includes another BCD with a version different from the one currently on the disk. This might happen, for example, if the included BCD had been recompiled.

2. A source file is "newer" than the corresponding BCD. This could happen if the source had been edited, or if the source had been retrieved from a remote file server. The IncludeChecker compares the creation date of the source file against the creation date recorded in the BCD of the source file from which the BCD was derived.

The IncludeChecker operates in either command line or interactive mode. To use it in command line mode, type to the Alto executive:

```
>IncludeChecker [outputfile][/switches] [filename1 filename2 ...]
```

where

`outputfile` is the name of the file written. If no extension or switches are given, `.list` is assumed. If no file name is specified, the file `Includes.list` is assumed.

`filename1 filename2 . . .` is the list of file names specifying the source and `.bcd` files to be checked. It is not necessary to give an extension, since the IncludeChecker will look for any `.mesa` or `.bcd` file with the specified name. If no input files are specified, all `.mesa` and `.bcd` files on the disk are examined.

To use the IncludeChecker interactively just type:

```
>IncludeChecker
```

It will then prompt for the output file name and switches, and then a list of the files to check. These are typed one at a time, and the list of file names is terminated by a `CR`. Typing ? `CR` in interactive mode displays a short summary of the IncludeChecker's parameters and use.

Each switch can be preceded by a `-` or `~` to turn it off. The switches are:

`/o` Print a compilation order in the output file (this is the default); `-o` suppresses this listing.

`/i` Print both the includes and included by relationships in the output file (default).

`/t` Obtain the creation date of source files from their leader page (default); `-t` will attempt to get the creation date from the first few lines of the source text.

`/c` Write a consistent compilation command in `Line.cm` (`-c` is the default). In addition, list as comments any BCDs and source files not on the disk which are needed to do the compilation.

`/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. This switch is useful if the output would otherwise be too large to fit into Bravo.

`/n` Do not compile source files that do not currently have corresponding `.bcd`s on the disk (`-n` is default).

`/p` Place a `/p` after every change of inclusion depth (see below) in the compilation command (`-p` is default). This will cause the Compiler to pause if errors are found while compiling that or any previous module.

`/s` Same as `/c-i-o`. This is used when only a consistent compilation command is needed.

The default switches are `/oit-c-m-n-p-s`.

Note: *The IncludeChecker only checks for consistency of the files that you specify*. Thus, the list of files that you give should include, for example, any important system files upon which your files are dependent.

You should also inspect the compilation command before executing it, since the IncludeChecker's idea of what should be recompiled may not be the same as yours.

If a source file but no BCD is found on the disk, the IncludeChecker outputs a warning on the display; in addition, it adds that file to the compilation command if `/c` and `/-n` are in effect. A warning is also displayed if a BCD is found that was created by an obsolete version of the Compiler; its source file is also added to the compilation command.

The IncludeChecker lists the file names of the compilation order and the consistent compilation command by inclusion depth, with the files that are the most deeply included first. Within that constraint, definitions modules are printed before program modules. In general, then, the "lowest level" definitions modules appear first, while the "highest level" program modules appear last.

As an example of the IncludeChecker's use, the command line

```
>IncludeChecker IC/c IODefs IOPkg LexiconDefs Lexicon
LexiconClient
```

will produce a consistent compilation command in `Line.cm` and the output shown below on `IC.list`.

```
Compilation Order (by inclusion depth):
  LexiconDefs streamdefs stringdefs
  IODefs
  oldstringdefs systemdefs tty windowdefs
  IOPkg Lexicon LexiconClient


IODefs (4-May-80 16:20:37 60#203#) (compilation source: 14-Apr-80 17:37:16)
```

```
includes
  streamdefs
  stringdefs

IOPkg (28-May-80 9:30:01 60#203#) (compilation source: 28-May-80 9:08:43)
    (source on disk: [same]) includes
  IODefs (4-May-80 16:20:37 60#203#)
  oldstringdefs
  streamdefs
  tty
  windowdefs

Lexicon (28-May-80 9:30:29 60#203#) (compilation source: 28-Apr-80
17:02:20)
    (source on disk: [same]) includes
  IODefs (4-May-80 16:20:37 60#203#)
  LexiconDefs (14-May-80 10:48:49 60#205#)
  oldstringdefs
  systemdefs

LexiconClient (28-May-80 10:02:50 60#203#) (compilation source: 28-May-80
10:02:14)
    (source on disk: [same]) includes
  IODefs (4-May-80 16:20:37 60#203#)
  LexiconDefs (14-May-80 10:48:49 60#205#)
  oldstringdefs

LexiconDefs (14-May-80 10:48:49 60#205#) (compilation source: 18-Apr-79
19:19:11)
    (source on disk: [same]) includes nothing


IODefs is included by
  IOPkg                 Lexicon
  LexiconClient

IOPkg is included by nothing

Lexicon is included by nothing

LexiconClient is included by nothing

LexiconDefs is included by
  Lexicon               LexiconClient
```

BcdSignals

BcdSignals is an Alto/Mesa program which will produce a signal listing from a `.bcd` file; it works much like the Alto/Mesa SignalLister for listing the signals in a `.image` file (see the *Mesa User's Handbook*). To produce the signal listing `Foo.signals` from `Foo.bcd`, type to the Alto Executive:

```
>BcdSignals [octalNumber/switch] Foo[/switches]
```

where

/n	takes `octalNumber` to be the global frame index of the first frame in this BCD. This will normally be the first free global frame index in the system into which the BCD will be loaded.
/x	takes `octalNumber` to be the StartPilot loadmap form of a global frame index. This is the number in brackets beside the module name in the loadmap. It should be 200B times the octal number used with the /n switch.
/p	lists the name, byte PC, and length of each procedure in `Foo` on `Foo.procs`.
/s	list the signals of `Foo` on `Foo.signals` (default).

As usual, a `-` or ~ can be used to invert the sense of the `/s` switch.

Distribution:
Mesa Users
Mesa Group
SDSupport

Inter-Office Memorandum

To	Mesa Users	Date	October 27, 1980
From	Jim Sandman, John Wick	Location	Palo Alto
Subject	**Integrated Mesa Environment**	Organization	SDD/SS/Mesa

XEROX

Filed on: [Iris]<Mesa>Doc>CommandCentral.bravo (and .press)

This memo documents a small executive called Command Central; this Tool is intended to be installed with the Debugger and can be used to invoke the Compiler, the Binder, and client programs, all of which upon completion are directed to return to Command Central rather than to the Alto Executive. The idea is that, while programming in Mesa, you enter Command Central's control only once, and you rarely have to leave it; this is made possible by the editor that is now included in the Debugger, as well as by the context switching facilities provided by Command Central.

Installation

To include Command Central in the Debugger, type the following Alto Executive command when installing, after retrieving `<Mesa>Fetch.bcd` (which contains **TinyPup**, **Stps**, and the **FileTool**) and `<Mesa>Utilities>CommandCentral.bcd`. (If you have more than 64K of memory, be sure to consult the Installation section of the Debugger documentation before proceeding.)

```
>XDebug Fetch/l CommandCentral/l
```

While it is possible to use Command Central without also installing the **FileTool**, including it will help minimize the number of times you have to leave the Mesa environment. If you have enough memory on your machine, you might consider installing other Tools with your Debugger as well (e.g., **ChatTool**, **SendMessageTool**).

Note: Tools loaded via the command line are initially inactive (i.e., no window is showing); move the cursor into the gray area outside all windows and use the menu found there to activate them.

Entering Command Central

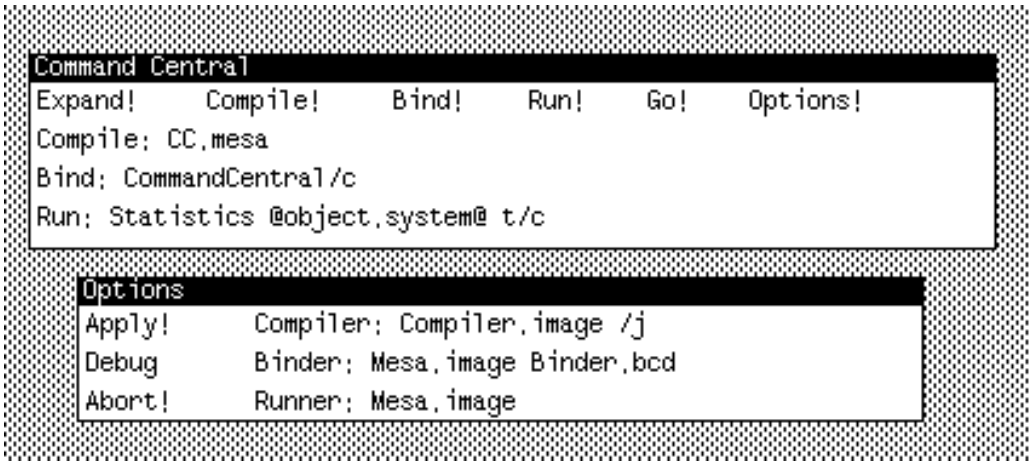
When using Command Central, the Debugger becomes the executive from which all programs are invoked. To first enter this environment, type

```
>Mesa/d
```

to the Alto Executive. You can now use the **FileTool** to retrieve the modules you wish to work on and the Tools editor to modify them. When you have finished your changes, turn your attention to the Command Central window.

Command Central Window

This window provides command lines for compiling, binding, and running your program, the context switching commands, and an option sheet; it also supports the standard window operations (scrolling, growing, etc.).



The three fields contain command lines for the Compiler, the Binder, and the System; their contents are written to `Com.cm` when the commands are invoked. (A special global switch /q is added so that control is returned to the Debugger rather than to the Alto Executive.)

The `Compile!`, `Bind!`, and `Run!` commands invoke the appropriate programs using the command lines constucted from the `Compile:`, `Bind:`, and `Run:` fields, respectively. The `Run!` command can be used to invoke `.bcd`, `.image`, and `.run` files (see below). The `Go!` command constructs a combined command line using all non-null fields and executes the appropriate programs in order. Note that the Debugger will complain if you issue any of these commands while a file is being edited (since the edits might be lost as a result of executing them).

The parameter fields also recognize command files preceeded by the traditional at-sign (e.g., `@file.cm`); the `Expand!` command will expand all such references into their contents and write the result back into the window. Indirect references are also automatically expanded when any of the other commands are invoked.

The `Options!` command produces the options window which allows you to specify the names of the compiler, binder and system you are using. The names are parsed to allow default switches to be included. The `Apply!` command will save the new names and the `Abort!` command will restore the names to their previous state (the default names are shown above). Both `Apply!` and `Abort!` will remove the options window.

If the Compiler or the Binder detect errors (and the pause switch is in effect), they will invoke the Debugger with an appropriate message instead of pausing. You can then load the appropriate error log into a window and step through it and your source file together. Because the file index of the error is included in each message, the `position` menu command can be used to find the source of the error quickly.

Invoking Other Programs

Any Mesa `.bcd` which expects to be loaded into `Mesa.image` and obtains its commands from the command line (`Com.cm`) can be invoked by Command Central using the `Run:` field and the `Run!` command. (As above, the global /q switch is added to the command line so that control will return to the Debugger.) Some obvious programs which you might include on your disk are **Access** and **Print**.

You can also run arbitrary `.image` and `.run` files using Command Central, but unless they have made provision to return control to the Debugger, they will exit to the Alto Executive upon completion. Use the `Mesa/d` command to reenter Command Central.

Limitations

If you use the `Compile!`, `Bind!`, `Run!`, or `Go!` commands when you are in the middle of a debugging session (at a breakpoint or an uncaught signal, for example), the state of the client will be lost. In particular, normal termination processing of the client will not take place (e.g., open files will be left dangling).

Distribution:
- Mesa Users
- Mesa Group
- SDSupport