M.I.T. Laboratory for Computer Science

Request for Comments No. 206
May 1, 1981

An Introduction to the Use of the Mesa System

by Robert W. Baldwin

1. Introduction

This memo is the first part of a much longer memo intended to make Mesa programming easier. The most useful sections in this part are the ones on the Alto Executive, and the Mesa Debugger.

2. Getting Started

This section is a slight rewrite of the section of the same name in the Mesa Users Handbook.

2.1. Getting the Basic Mesa Disk

The best way to make a new Mesa disk is to copy an existing one. You can either copy someone's current disk (and then delete the uneccessary files), or copy the basic Mesa disk stored on the file server. All the features described in this document are provided in the basic disk, other people may have different or inconsistant versions.

To get the disk from the file server, load a blank or re-cycleable disk, and boot the program CopyDisk from the network. To do that, hold down the **BS** and **'** keys then press the **boot** button on the back of the key board. This boots the network executive (NetExec). Type **CopyDisk** to tell the NetExec to down load the disk copying program.

CopyDisk prompts with a *, type **copy** and it will ask you where to copy from. Answer **[Seal]<Disks>BasicMesa.dsk**. It will probably ask you to login. If you don't have an account,

and you should if you intend to do much programming, use the guest accout name = `user`, password = `user`. Next, CopyDisk asks where to copy to. Answer `DP0`. It then reminds you that the old contents of DP0 will be lost, and asks for two confirmations. If the copy fails (an unlikely event), just try again. If it fails again ask someone for help.

Once the disk has been copied, press the `boot` button with holding down any keys. This will cause the alto to boot from the disk (dp0) and it will start running the Executive (see XXX).

2.2. Installing the Debugger

After you have a disk you MUST install the debugger to establish the communication link between the Mesa Executive and the debugger. To do this, type the command `@MakeXMDebug` to the Alto Executive. Actually, this command invokes a list of commands in a file (see the section on the Alto Executive). These commands get the debugger from IFS, install it, and delete the unnecessary files.

2.3. Editing

Programs are prepared with the text editor Bravo (see the Alto user's Handbook for documentation). Bravo provides a rich set of formatting capabilities that greatly enhance the readablity of Mesa code either on paper or on the screen. The BasicMesa disk comes with a couple of templates for mesa modules, which make it possible to write well formatted code without knowing a lot about Bravo formating.

2.4. Compiling

The Mesa compiler has the easy to remember name `Compiler.image`. The `.image` means that it is a stand alone program. The command `Compiler n/c b/c u/c source1 source2 ...` compiles several sources files into object files. The default extention for source files is `.mesa` and the object files end in `.bcd`. The switches like `b/c` turn on various compiler options. The three listed respectively turn on: code to signal an error if you fetch data using a NIL pointer, code to signal an error if an array reference is out of bounds, and extra source checking to produce a warning if a variable appears to be used before it is initialized. Oddly enough, the default setting for these options is off.

Not all programs compile the first time. When an error is detected by the compiler it records the source statement and error message in the file `Foo.errlog`, where `Foo.mesa` is the name of the offending source file. See Appendix A of the Mesa User's Handbook for further information.

2.5. Binding

Several object files can be bound into a runnable program by using the Mesa program `Binder.image` or `Binder.bcd`. The command `Binder Sourco` reads the file `Sourco.config` and produces a file `Sourco.bcd` which can be run directly or bound into larger programs.

The file `Sourco.config` is a text file which decribes how to put together existing object files. The binder understands a full blown programming language called C/Mesa. The amazing and obscure feature of C/Mesa are described in the Mesa Language Manual. The key thing to remember is that `.config` files contain source code in the C/Mesa language which is compiled/interpreted by the Binder. C/Mesa is confusingly similar to Mesa, so be careful. See Appendix B of the Mesa Users Handbook for more information.

2.6. Running

To run a mesa program type `mesa filename.bcd` to the alto executive. This will invoke the program `Mesa.image` which serves as an executive and as a runtime support package and as a library of common routines. Mesa will automatically load in the named file and start executing it.

Notice that the Mesa language, the Mesa executive, the Mesa runtime system, and the Mesa loader are all called 'Mesa' without distinguishing which one is reallly meant. This will cause you endless confusion. See Appendix C of the Mesa User's Handbook for more information.

2.7. Debugging

To set break points and display the run time state of your programs, you must first invoke the debuger. The most common way is to hold down the control key and press the swat key (bottom blank key). The difference between most debuggers and the Mesa debugger is that it performs all operations at the source language level. See the section on the Debugger for more information.

3. The Alto Executive

3.1. Syntax

The Executive is the program that starts up when the Alto is booted. It gets commands from the user and executes them. Command lines do not have a rigid syntax beyond the fact that the first word of the line must be coercible into a program name. By convention, most programs use parse the following syntax:

```
<program-name>[/<global switches>] [<value>/<switch>]*
```

Where [] indicates an optional part, and []* indicates zero or more repetitions of an optional part. In the following command line, the global switch u tells Ftp to start in User-only mode. The local switch c indicates that the value is a command as opposed to a server name or file name. The Retrieve command is modified by the switch n which says to retrieve the file only if it is new (i.e., it does not already exist on your disk).

```
Ftp/u Connect/c Seal Retrieve/n <Alto>Ftp.run
```

3.2. Com.cm and Rem.cm

Before a program is executed a copy of the command line is placed in the file Com.cm. In fact Ftp looks in Com.cm for it's commands. If the command line contains multiple commands (separated by semicolons), all but the first command is copied into the file Rem.cm. When the Executive starts running, it looks in the file Rem.cm for additional commands, if there are none it accepts commands from the keyboard. Thus it is possible for a program to write commands into Rem.cm and have the Executive execute them. The programs DO and IF, described later, use Rem.cm to extend the capabilities of the Alto executive.

3.3. Special Characters

Several characters have special meanings to the executive. Some of the more useful ones are:

~ BS
Deletes the last character.

~ DEL
Deletes the whole line.

~ CR
Marks the end of a comand line.

~ ESC
Assume the previous word is a file name and complete as much of it as possible.

~ TAB
List all the files that match the previous word and clear the command line. See ?.

~ ?
Same as TAB, but doesn't clear the command line.

~ *
Matches zero or more characters in a file name. The command, `*.mesaTAB`, is a good way to get the names of all the mesa source files on your disk. Also, `Delete *.*`, is a very bad thing to do.

~ #
Matches any one (exactly one) character. Useful with TAB or ?.

~ '
Quote the next character. This is particularly useful for putting #'s in a command like `FTP 2'#4'#`.

~ ↑
Ignore the next character in the command line. This is used to make long command lines more readable by allowing the line to include CR's. It is mostly used in command files. See below.

~ ;
Seperates multiple commands typed in at the same time. For example, `FTP Ifs Retrieve/c MoreHints.memo; Bravo/n MoreHints.memo` will retrieve this file and enter bravo automatically.

~ ↑X
Control X expands any pattern matching characters in the command line, and replaces command files (see below) with their contents. Typing `Delete *$↑X` will show you what files you are about to delete. If the files are the ones you wanted to delete you can type CR, if not you can type DEL.

~ ↑C
Control C aborts the processing of the command line. If you type this after a CR but before the program has finished loading, then the command will be aborted. Similarly, if the executive is processing commands in Rem.cm, ↑C will abort the processing.

3.4. Command Files

Common command sequences can be put into files and invoked using the character `@`. For example `@DumpSources@` will execute the commands in the file DumpSources.cm. Reading input from a file can also be used in the middle of a command file. For example `FTP Ifs store/c @MyFiles@ ret/c @HisFiles@` will store all the files listed in the file MyFile.cm and retrieve all the ones in HisFile.cm. The name between the @'s does not have to be complete. The extension `.cm` is assumed. The second `@` is not necessary if it would be the last character on the line. Thus `@DumpSources` and `@DumpSources@` are the same. See the section on useful command files for interesting applications of `@`.

3.5. Aborting Programs

While a program is running it can usually be aborted by holding down the left-hand shift key and the lower blank key (called the `swat` key) on the right hand side of the key board. To enter the Mesa debugger hold down the control and swat keys. To enter the BCPL debugger, Swat, hold down the left-hand shift, control, and swat keys. For more information on the Alto Executive see [Ifs]<AltoDocs>Executive.tty.

4. IFS

4.1. Sub Directories

IFS can support a moderately useful notion of subdirectories using its general pattern matching scheme. To put a file in subdirectory `foo`, you just store it as `foo>filename`. FTP has some features to make this easier, see below. A useful thing to do when you make a new directory is to create a file called `foo>`, this way you can get a list of your subdirectory names by listing all files matching `*>`. Such a list is probably more useful to people trying to locate some file on your directory than it is for you.

4.2. Deleting Backup Versions

IFS has a subcommand for the delete operation which allows you to specify how many versions of a file to keep. To do this you have to Chat to IFS and say:

```
Delete foo*,
Keep 2
```

Don't forget the comma, there is no undelete operation on IFS. If you do not want to confirm each deletion caused by the sequence above, use this one instead:

```
Delete foo*,
keep 2
confirm
```

5. FTP

5.1. Sub Directories and FTP

To put a file in subdirectory `foo`, you just store it as `foo>filename`. The easy way to do this is to use the `directory` command of FTP to set your default directory to `YourName>foo`. The default directory name is prepended to any file name that does not begin with a <. The `directory` command prepends your name if the first character of the directory name is a >. So `directory >foo` is equivalent to `directory YourName>Foo`.

5.2. Abreviations

Commands from either the keyboard or the command line only need to be as long as necessary to distinguish them from all other commands. For example, `ret` and `retrieve` are the same. Unfortunately, `r` is not enought to specify `retrieve`, since it can be confused with `Rename`.

5.3. FTP Commands on the Command Line

FTP commands can be given on the command line instead of typed in by the user. This feature makes all kinds of command (.cm) files possible. For example, the command line `FTP Seal Dir/c AltoDocs Ret/c Executive.tty Pressedit.tty` gets two documentation files from Seal. The local switch, `/c`, after `Dir` and `Ret` indicate that these are commands and not file names. Retrieve and Store take lists of files as arguments, so the `/c` is necessary to tell when a file list ends and a new command begins.

Using the `Dump` command, several files can be compressed into one in order to reduce page breakage or directory clutter. The `Dump` command takes two or more arguments. The first one is the name of the dump file, the rest are the names of the files to put in it. The inverse operation of dump, load, takes one or more arguments. The first is the name of the dump file, the rest, if any, are the names of the specific files you want to retrieve.

There are other switches besides `/c`. For example, `ret/o` only retrieves a file if it exists on your disk (i.e., it is old). Similarly, `ret/n` only retreives new files. The `/s` switch renames a file as it is being stored or retrieved. For example `ret/s PeekUser.cm User.cm` gets a file called `PeekUser.cm` and puts it on your disk as `User.cm`.

5.4. Selective Loading From Dump Files

To reduce page breakage several files can be stored under one name. These names typically end in `.dm` and are made with the command `Dump`. To get all the files from a dump file you use the command `load`. However, from the command line you can tell FTP to load only specific files from a dump file. For example, the line `FTP Seal Dir/c Altosource Load/c CopyDisk.dm Disk.decl Main.bcpl` will un-dump and retrieve just the files `Disk.decl` and `Main.bcpl`.

FTP has many switches and features, see [Ifs]<AltoDocs>FTP.press for more information.

6. The Mesa Debugger

6.1. Interface

Windows. When the debugger starts up there are two overlapping windows on the screen. One accepts debugger commands and is used to print the results of those commands, while the other displays an arbitrary source file. The Source Window is linked to the debugger in the sense that you can set break points by selecting a statement and bugging the appropriate menu command (see Menus). Pressing the left mouse button in the upper left hand corner of a window will toggle its position along the Z-axis. That is, the window on top it will go to the bottom and vice versa. You can cause the window to zoom to maximum window size and back by pressing the left mouse button while the cursor is in the middle section of that window's black header. You can also change the size and shape of windows, create and destroy new windows and many other things. See the section 2 of the Mesa Debugger Documentation for more information.

Menus. Commands can be invoked by key-strokes or by menus. Invoking a command through a menu is calling 'bugging' the command. Bugging is a three step process. First you bring up the menu stack by holding down the right mouse button in some window. Next you select the desired menu from the stack of menus by moving the cursor over the menu's name and pressing the left

button. This will bring that menu to the top of the stack. If the menu you want is already on top, the previous step is not necesary. The last step is to move the cursor over the desired command and release the right button. Releasing the right button executes the command. If you accidently bring up the menu stack, move the cursor away from the menus before releasing the button.

Loading Windows. A window can be loaded manually or automatically as the result of some debugger command. To manually load a window type a debugger command of the form -- `Filename`, this is a comment and will have no effect on the debugger. Next, select the file name by pressing the middle mouse button with the cursor positioned over the name (see Selecting). Now go into the window to be loaded and (after bringing up the `Source` menu), bug the 'Load' command. The file is assumed to end in `.mesa` if no extension is given.

Command Completion. The debugger's command language may be thought of as either having automatic command completion, or as being character-oriented. For example, typing the two letter sequence "ds" will cause the command line `Display Stack` to appear, and will start the execution of that command. Actually, typing the words `display stack` in full will result in much confusion.

Editing. The debugger uses the standard Mesa (the system) editing conventions. The last character is deleted by the 'BS' key, the last word is delcted by control-W or the top blank key next to the `BS` key. Whole lines are deleted by the `DEL` key.

Selecting. Several commands work on a selected piece of text. To select a region, position the cursor over the first character then while holding down the left mouse button, move the cursor to the last character in the region and release the mouse button. The selected text will be surrounded by a gray box. If you hold down the middle button, words, rather than characters, will be selected. At the end of each line is a return character which displays as a space. It can be selected by either the left or middle mouse buttons.

Stuff It. The debugger has an additional editing command which can be invoked by a menu or by a key-stroke (middle blank key on the right side of the keyboard). `Stuff It` takes the current selection and appends it to the current command line. This is useful to avoid retyping a long command and to copy the result of a previous command to the current command. For example, the command `List Processes` produces a list of the octal addresses of the Process State Blocks. If

you then decide to **SEt Process** to one of them you can just select the appropriate address and press the **Stuff It** key. This way you avoid typing mistakes.

Defaults. The debugger keeps an extensive list of defaults which it remembers from one session to the next. Typing **ESC** will give you the default for the current command context (not to be confused with the name lookup context described below). The default is the last thing you typed in that command context. For example, if a command calls for a process identifier, **ESC** will produce the last process identifier you entered (typed or stuffed in). Using the **ESC** and **Stuff It** keys make debugging much easier. Try using them.

Aborting Commands. Most commands can be aborted by holding down the control and **DEL** keys.

6.2. Commands

6.2.1. Introduction

To increase readability, debugger commands given below are fully spelled out. However, to issue these commands to the debugger, only type the letters which are shown in upper case. For example, the command **Display Stack** is issued by typing **ds** or **DS** to the debugger.

The debugger generally ignores the difference between upper and lower case. This applies to commands, module names and variable names. Of course you can change this with the commands **CAse ON** and 2t'CAse OFf'.

6.2.2. Starting Up

The debugger must be installed before the first time you use it on a new disk. The command file **MakeXMDebug.cm** will retrieve the appropriate files from the file server and do the installation. To execute this command file type **@MakeXMDebug** to the Alto executive. Installing the debugger or getting a new Mesa.image can sometimes cure strange debugger behavior.

There are three ways to enter the debuger. From the Mesa executive you can execute the command **Debug**. To enter the debugger after a program is loaded but before it is running, you use the Alto executive's global switch **/d**. For example: **FooTest/d** will load FooTest.bcd into the Mesa system and enter the debugger before executing it. Lastly, when a program is running, holding down the **control** and **swat** (bottom blank) keys will enter the debugger.

6.2.3. Contexts

The Mesa debugger allows you to refer to modules and variables by names. Of course, for the debugger to find a name it has to be in the scope of the current context. The debugger's contexts consists of 1) a root configuration which defines the context in which module names are looked up, 2) a stack for the current process which is used to resolve names of local variables, and 3) a module name for looking up global variables and for finding sources statements to set break points or call procedures. To resolve a variable name, the debugger searches the stack of the current process in LIFO order. For each procedure it first checks the local variables and then checks the global variable in the module which implements the procedure. The lookup algorithm for module names is similar, but is based on the order in which modules were loaded into Mesa. The last module loaded in the current configuration is the first one checked.

When you enter the debugger the curent context is set to the running context. You can find out what the current context is by the command **CUrrent context**.

The command **SEt Root configuration** sets the context in which module names are looked up. Once the root is set, any module defined in that configuration can be referenced. Configurations are modules like any other, so if you have nested configurations you must first set the root to the outer most one, and work your way inward. To get a list of all the configurations you use the 'List Configurations' command.

When looking at local variables you will get the most recent instance of a particular name. This can cause problems with recursive procedures. You can use the **Display Stack** command to examine any local variable on the stack, but you can not assign to it using the debugger's interpreter (see below), since it can only reference the most recent instance.

The module context can be set explicitly by the **SEt Module context** command (the module name must be in the current configuration). After this command, the command **Display Stack** can be used to look at the module's local variables, and the interpreter will be able to reference those variables by name. The module context is used to look up procedure names for break points and interpreted calls (see below). It is also used to find break points set at particular statements in the Source Window.

For information amout the process context see the section on Multiple Processes.

6.2.4. Examining Variables

There are two ways to examine a variable: with the **Display Stack** command and with the interpreter. In both cases the trick is setting the current context to include the desired variable. See the section on Contexts.

The **Display Stack** command can do many things. I quote from the Mesa Debugger Documentation:

Display Stack follows down the procedure call stack. At each frame, the corresponding procedure name and frame address are displayed. You are prompted with a >. A response of **V** displays all the frame's variables; **P** displays the input parameters; **R** displays the return values [the name **(anon)** is used for unnamed values]; **N** moves to the next frame; **J** jumps down the stack n (decimal) levels (IF N IS GREATER THN 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 the source file into the Source Window; **L** just displays the source text; and **Q** or DEL terminates the **Display Stack** command. When the current context is a global frame, the **N** and **J** subcommands are disabled. When the debugger cannot find the symbol table for a frame on the call stack, only the **J**, **N**, and **Q** subcommands are allowed. For a complete description of the output format see section 6 of the Mesa Debugger Documentation.

Variables can also be examined by typing their name to the interpreter. To enter the interpreter, type the debugger command " ". That's right, the space character is a command. The interpreter is capable of evaluating a subset of Mesa, so you can use it to evaulate expressions involving variables. A useful special case of this is selecting sub-records and following pointers (e.g., **listHead.next.next**). The interpreter can also assign to variables, see the section on Assigning to Variables. For a complete description of the interpreter see section 5 of the Mesa Debugger Documentation.

The commands **Octal Read** and **Octal Write** can be used to examine and change specific memory locations. Both of them take an octal address and a decimal word count as their arguments.

6.2.5. LOOPHOLE

A very important feature of the interpreter is that it can display a variable using a different type than the one it was declared with. This is mainly useful when you are using a CLU programming style that hides the representation of objects from clients. To display a record of type `Foo` when all you have is a variable 'bar' of type `POINTER TO {hiddenFoo}`, you give the interpreter the expression `bar%@Foo↑`. The `%` says to treat the thing on its left (bar) as a variable of the type given on its right (@Foo). The `@` is short hand for `POINTER TO`. The `@` must not be followed by a space, or else it will be confused with the `AddressOf` operator. The `↑` dereferences the previous expression, so the debugger displays a record of type `Foo` instead of a pointer to that record.

6.2.6. Assigning to Variables

The interpreter handles the back-arrow operator, so you can assign to variables. Assignment can be used to overcome small bugs in the program and thus avoid Mesa's long edit-compile cycle for small bugs. It can also be used for simple testing.

The commands `Octal Read` and `Octal Write` can be used to examine and change specific memory locations. Both of them take an octal address and a decimal word count as their arguments.

6.2.7. Address of Variables

The unary operator `@` produces the address of the thing on its right. It must be followed by a space to avoid confusion with the `POINTER TO` type specifier. For example, to assign to the variable `bar` (of type POINTER TO Foo), the address of the third element of the array `Foos`, you would have the interpreter evaluate the expression (assume that the `:=` is a back-arrow character:

```
bar := @ Foos[3]
```

6.2.8. Referencing Items in a Defs File

Variables, types, and procedures which are defined in a Defs file can be referenced by using the `$` operator. In the same way that `foo.bar` selects the `bar` component of the record `foo`, the term `StackDefs$StackObject` selects the type `StackObject` from the file `StackDefs.bcd`. This sort of qualification is most useful in the interpreter and with the `Interpret call` command.

6.2.9. Calling procedures

The debugger has a primitive procedure interpreter. The command `Interpret call` asks for a procedure name (which must be in the current module context or qualified by a $), and then asks for the arguments (in octal) one word at a time. If an argument is a two word LONG CARDINAL, you give the low order word first. Basically, you must give this command an image of what will go onto the stack when the procedure gets called.

6.2.10. Looking at the User's Screen

There are two ways to look at the user's screen. The first is by the debugger comand `Userscreen`. This shows you what the screen looked like when the debugger was entered. If the program is also keeping a typescript file (the default for the procedures in IODefs), then you can examine that file in a window. Just make a new window or re-use the source window and load the file `Mesa.TypeScript`.

6.2.11. Break and Trace Points

Unlike most debuggers, Break points and Trace points are very similar. Trace points are useful when you do not intend to do much looking around in the debugger. When a Trace point is hit, the top of the stack is automatically displayed. Unlike the regular Display Stack command, typing `q` causes execution to proceed rather than ending the Display Stack command. If you want to do anything beside look at the stack, you have to type the command `d` to enter the regular debugger mode. Break points just enter the debugger with the context set to the running environment.

To set either Break or Trace points you can use the commands `SEt Break` or `SEt Trace`. Debugger commands can be used to set points at the entry to or exit from a procedure. To set a point at a specific statement, you load the Source window with the appropriate file, and use the commands on the 'Source' menu. Note that in both case the current context must include the name of the procedure in which the point is set. This is usually accomplished by the command `SEt Module context` to the module which contains the procedure.

Other related commands are `List Breaks`, `List Traces`, `CLear All Breaks`, `CLear All Traces`. These commands have the obvious effects.

6.2.12. Muliple Processes

When Mesa was extended to handle multiple processes, so was the debugger. One problem with this extension is that Mesa (the language) does not provide any way to name processes, thus the main diculty with processes is figuring out which process is the one you want. Once you have done that you use the command `SEt Process context` to identify which stack you want to look at. Remember that all processes are in the same address space, and that the module and configuration context is unaffected by changing processes.

Each process has a stack. To change the stack context, you change the process context. Not surprisingly, this command is called `SEt Process context`. Mesa does not have a high level way of naming processes, so this command accepts as its argument an expression which evaluates to the address of a Process State Block (PSB). The value returned by the Mesa `FORK` expression is the address of the new process's PSB. Thus, the most convenient argument to `SEt Process context` is the name of the variable which received the value returned by `FORK`. Of course, the naming context must be set to include that variable. To get a list of the addresses of all the PSB's use the command `List Processes`. That command will also tell you the name of the procedure which is on the top of each process's stack. If that isn't enough information to locate the process you want, the sub-command `Root` of the `Display Process` command will give you the name of the procedure which was forked to start that process. `Display Process` takes an argument which evaluates to a PSB address.

A process is always on exactly one queue. There are three types of queues: the Ready List for runnable processes, condition variable queues for processes waiting for notifies or timeouts, and monitorlock queues for processes trying to enter a monitor. The Ready List can be examined by the `Display Ready list` command. Monitorlocks and condition variables are discussed below.

6.2.13. Condition Variables and Monitor Locks

The debugger does not have a high level display for these primitive Mesa types. An obvious way to display a condition variable is to show which processes are waiting on it and how long until each one times out. Instead, condition variables are displayed as two octal numbers. The low order 15 bits of the first word is a pointer to the tail of a circularly linked list of processes (actually PSB's) waiting on that condition. The high order bit is only used by naked condition variables (i.e., one associated with hardware interrupts), and is set if a notify occurred while no process was waiting on

the condition. The second word is the initial value of the timeout timer. The time when a waiting process will timeout (not how long until that time) is stored in the PSB for each process waiting on the condition. For example, a condition which displayed as `0,0` has no processes waiting on it and no timeout (zero means that timeouts are disabled); `3256B, 144B` has a process waiting on it and an initial timeout of 144 (octal) ticks.

Monitor locks are displayed in a similarly primitive way. The most significant bit is one if it is unlocked, and zero if it is locked. The low order 15 bits points to the tail of a circularly linked list of processes waiting for the monitor lock. For example, a monitorlock which displays as `3256B` is locked (i.e., some process is in the monitor) and at least one other process is waiting for the lock; `100000B` is unlocked with no processes waiting for the lock.

The queue of waiting processes for both monitorlocks and condition variables can be displayed by using the `Display Queue` command. Its argument is either the name of a queue variable (monitorlock or condition) or the octal address of a queue.

For more information look at the Mesa system code. The modules: Process, ProcessDefs, PSBDefs, and ProcessOps are most relevant.

Table of Contents