DB: THE ATARI DEBUGGER

RELEASE 2, VERSION 90/01/24

BY ALLAN PRATT

Table of Contents

Table of Contents

CHAPTER 1

DB: THE ATARI DEBUGGER

Db is a debugger for the Atari ST and TT series of 68000-family computers. It is intended to replace **sid** as the assembly-level debugger of choice. It is not a source-level debugger, but it does handle both Alcyon C and Mark Williams C† (new and old) symbol table formats.

Db can use any of the ST's character devices for its input and output, including the screen, the serial port, and the MIDI port. The I/O device is selected with a switch on the command line (or in the TTP window if started from the desktop).

Db is capable of debugging programs running on one machine while the bulk of the debugger runs on another. This is called *remote debugging*, and permits debugging of operating systems while they boot, for example. This feature is described in the chapter REMOTE DEBUGGING.

USAGE

From a command shell, db can be started as follows:

db [*options*] [*program* [*args* ...]]

If started as a TTP program from the desktop, the arguments line looks the same without the word **db** at the beginning.

OPTIONS

Db can use many different devices for its input and output. This makes debugging graphics- and keyboard-oriented programs easier.

These *options* on the command line select the output device to use:

-g

Use GEMDOS to access the ST screen and keyboard. This is the default case, but it does have limitations. See the section **DB AND GEMDOS** in the chapter **OPERATING SYSTEM CONSIDERATIONS** for more information.

---

†Mark Williams C is a trademark of Mark Williams Company.

-b*N*

Use the BIOS to access the ST screen and keyboard. Sometimes this helps when debugging a program which itself does BIOS I/O, because using GEMDOS calls can mess up type-ahead and the like.

You can (optionally) specify which BIOS device to use by placing the BIOS device number after the **-b**: "-b3" means "use BIOS calls for input and output, and use BIOS device number 3 (the MIDI port). The argument is in decimal. Any number at all may be used here, including numbers which are not in fact BIOS device numbers; in this case, the debugger will probably crash, and it is likely that you will have to reset your machine.

-s

Use the serial (RS232) port. A terminal or an ST running a terminal program must be connected via a "null modem" cable, and its keyboard and screen are used for communicating with the debugger. (You can even use a modem connection to a terminal or computer, but this is extreme.) The baud rate, parity, etc. for the serial port must be set before starting the debugger in this mode.

-m

Use the MIDI port. An ST running a terminal program which uses the MIDI port must be connected with a double-MIDI cable (i.e. MIDI OUT -> MIDI IN and MIDI IN -> MIDI OUT). One such program is **miditerm**, included on the distribution disk with the debugger. It is a minimal terminal emulator program, but it gets the job done.

In the last two modes, the debugger controls the serial or midi port hardware directly, without going through GEMDOS or the BIOS, so there are fewer limitations on debugging programs which use GEMDOS or the BIOS. However, the the limitations with respect to the operating system always apply, except when remote debugging. See the section **DB AND GEMDOS** in the chapter **OPERATING SYSTEM CONSIDERATIONS** for more information. Also, see **iodev** and **bdev** in the section on debugger variables.

Each of the options **-g**, **-b**, **-s**, and **-m** can be followed by the letter **x**: this controls the printing of non-standard characters when using the **d** (dump) command. Non-standard characters are those with ASCII codes 128 and up. Normally, these are printed in the ASCII part of the dump command's output. When **-s**, **-m**, or **-b** with a device-number code is used, printing of these characters is suppressed, because they confuse most terminals. The presence of the letter **x** (e.g. **-sx** or **-bx1**) re-enables printing of these characters, which can be useful if your terminal is in fact another Atari computer with the same extended character set. The **x** modifier also controls the use of inverse video for error messages: if the Atari ST

extended character set is used, the VT52 code for inverse video will be used too.

In addition, the following *option* controls loading of the initialization script:

-ifile

The debugger normally searches for and executes a startup file when it is run. The -i option disables this. With the optional *file* argument, the normal startup file is not loaded, and *file* is loaded in its place. There must not be a space between the **-i** option and the *file* argument: "**-imyfile**". See the section **USING THE DEBUGGER** in this chapter for more information.

Usage examples:	
db	start the debugger; use GEMDOS for I/O.
db -s myprog.prg -z	use the serial port for I/O; load myprog.prg for execution, with the command-line argument -z.

TERMS

Several terms are used throughout this document which must be defined here.

The *client* is the program you are debugging.

The *head* is the part of the debugger which handles all the user input and output. The commands you type are translated by the head into commands for the *stub*. It is the stub which causes the client to run, processes breakpoints, and catches exceptions like bus error. The stub reports these events to the head, which reports them to you.

When you are *remote debugging*, the head runs on the *master* machine, and the stub and client run on the *slave* machine. The head gives commands to the stub and receives the stub's responses through the *communications* layer, which actually talks over a MIDI cable.

The term *debugger* is used to refer to the head, stub, and communications; in short, everything but the client (program) and the user (human).

You cause the client to execute instructions with the **g** (go), **t** (trace), **u** (untrace), and **v** (verbose-trace) commands, collectively known as *trace/go* commands. A *stop* is anything which causes a trace/go to stop: a bus error, address error, or other processor *exception*, a breakpoint whose count has reached zero, or a memory checkpoint which becomes true. Memory checkpoints are evaluated at times called *opportunities*, which occur when

processing exceptions, including the illegal-instruction exception caused by breakpoints and the trace exception which happens between instructions of a trace.

You can put a list of commands to be executed in a file, and cause those commands to be executed by the debugger using the **load** command. Such files are called *scripts*. Also, *procedures* consisting of debugger commands, arguments, and local variables are available.

USING THE DEBUGGER

When the debugger is started, it processes its GEMDOS command line first. If there are any *options* (like **-m** or **-s**) they are checked and dealt with. Then, if there is a *program* argument, that program is loaded and set up for executing. It becomes the client. If there are any *args* they are placed in the client's basepage, as GEMDOS command-line arguments to it. When the client is completely set up and ready to run, the debugger prints out its basepage information (text size, environment pointer, etc.). This client set-up amounts to the same thing as using the **exec** command.

The debugger then looks for and loads your configuration file (that is, it executes the commands found there; such files are called *scripts*). The first place it looks is the current directory, for a file called **db.rc**. If that file doesn't exist, it looks for the file named in the environment variable DBRC. If there is no such environment variable, it looks for the file **db.rc** in the directory named by the environment variable HOME. If none of these files exists, the debugger simply continues with the start-up procedure.

When remote debugging, the autoload procedure is the same, except that the debugger looks for **rdb.rc**, then the file named in the environment variable RDBRC, followed by **rdb.rc** in the HOME directory.

In either case, the **-i** option on the debugger's command line inhibits the loading of a startup file. If the **-i** option has a *file* argument, that file is loaded instead. The debugger searches for the *file* in the current directory first, then in the HOME directory.

Whether or not there was a *program* argument to execute and/or a startup file, the debugger ultimately displays its prompt, a colon (":"). Any time you see the colon prompt, the debugger is waiting for you to type a command line. Command lines consist of commands and their arguments. Multiple commands on one command line are separated by semicolons (";"). Multiple-letter commands must be separated from their arguments by a space (e.g. "**where** 12322"), while single-letter commands don't need a space (e.g. "d12322" or "d 12322").

You can always use ^S (control-S) to stop the debugger's output and ^Q to start it again. You can usually use ^C to abort a command, especially commands which generate long listings.

All numbers printed by the debugger are in hex. All numbers you type are

assumed to be hex, unless prefixed with @ (decimal), ^ (octal), or % (binary).

When debugging programs compiled under Mark Williams C, you need to play a trick before you start the program. See the section **DB AND MARK WILLIAMS C** in the chapter **OPERATING SYSTEM CONSIDERATIONS** for more information.

When **remote debugging**, the debugger will display its version number, then wait for the stub to respond before loading the configuration script.

CHAPTER 2

EXPRESSIONS, RANGES, AND STRINGS

This chapter describes how values are entered into the debugger, mostly as arguments to commands. An *expression* is something which boils down to a single numeric value. A *range* is something which boils down to a starting address and a length: a range of addresses. A *string* is something which boils down to a series of single-byte values. A section on each follows.

EXPRESSIONS

An *expression* can be used any time a numeric value (like an address or count) is expected. All expressions evaluate to 32-bit integers. Overflow is checked when reading a constant (so the hex constant $FFFFFFFF0 would cause and error because it requires 36 bits). Overflow is not checked in any other situation. There are two kinds of expressions: *simple* expressions and *complex* expressions.

SIMPLE EXPRESSIONS

Simple expressions contain no operators and are not enclosed in parentheses. There may not be any spaces in a simple expression. Simple expressions take one of the following forms:

hex constant
$hex constant
A hex constant has the obvious value. The leading '$' is optional: with no prefix, a number is assumed to be hexadecimal. Hex constants consist of an optional sign (+ or -) followed by one or more of the digits 0-9, A-F, and a-f.

Examples: 0, 1, 3FA, 13aD4, $ffffa4d0, $-5b30 (same as $ffffa4d0).

@decimal constant
A decimal constant begins with an at-sign ("@"), then an optional sign (+ or -), then one or more digits 0-9. It has the obvious value.

Examples: @0, @99, @-32768 (same as ffff8000).

^octal constant
An octal constant begins with a circumflex (up-arrow, "^"), then an optional sign (+ or -), then one or more digits 0-7. It has the obvious value.

Examples: ^0, ^77, ^20000000000 (same as 80000000).

%binary constant

A binary constant begins with a percent-sign ("%"), then an optional sign (+ or -), then one or more digits 0-1. It has the obvious value.

Examples: %0, %1010, %1000000000000000 (same as 00008000).

.symbol

A leading period ('.') indicates that what follows is a symbol specification. The value of the expression is the 32-bit value in the symbol's value field. A symbol specification can simply be the name of the symbol (e.g. ".start") or something more complex. See the chapter **SYMBOLS AND DEBUGGER VARIABLES** for more information.

Examples: .main, .gemlib:xmain:__main:L3

`variable

A leading backquote ('`') indicates that what follows is a debugger variable name. The value of this expression is the value in the corresponding debugger variable. See the chapter **SYMBOLS AND DEBUGGER VARIABLES** for more information.

Examples: `d0, `clientbp, `mtype

&variable

A leading ampersand ('&') indicates that what follows is a debugger variable name, and the value of this expression is the *address* of the storage for indicated variable *in the stub's memory*. These variables should not be changed, since the debugger's local copy of the variable might overwrite your change. However, these addresses can be used in memory checks to set checkpoints on the values in registers.

See the section **DEBUGGER VARIABLES** in the chapter **SYMBOLS AND DEBUGGER VARIABLES** (especially the subsection **Client Registers**), and the section **MEMORY CHECKPOINTS ON VALUES IN REGISTERS** in the chapter **THE CLIENT, BREAKPOINTS AND CHECKPOINTS: DETAIL** for more information.

Examples: &d1, &pc, &sr

$

The dollar-sign alone is short for `$. This temporary variable is set to the result of the last math command (that is, just an expression on the command line). In addition, the f (find)

command sets $ to the address of the start of the first match.

Example: $

COMPLEX EXPRESSIONS

A *complex expression* is a LISP-like expression containing parentheses, operators and operands in prefix notation. Unlike simple expressions, there may be spaces in a complex expression.

The first element within the parentheses of a complex expression must be an operator. The second and subsequent elements must be expressions (simple or complex) which are used as operands for that operator.

In the following table of operators, "exp ..." means "one or more expressions." For the logical operators (and the **if** command), zero is "false" and anything nonzero is "true." The logical operators themselves all return the number 1 if the expression is TRUE.

FORMAT	COMMENTS
MATH	
(+ exp ...)	Add the expressions together
(- exp1 exp2)	Subtract exp2 from exp1
(* exp ...)	Multiply the expressions together
(/ exp1 exp2)	Divide exp1 by exp2
(% exp1 exp2)	Return exp1 modulo exp2
BITWISE	
(& exp ...)	Bitwise AND the expressions together
(| exp ...)	Bitwise OR the expressions together
(^ exp ...)	Bitwise EXCLUSIVE OR the expressions
(~ exp)	Bitwise NOT (invert) the expression
(>> exp1 exp2)	exp1 >> exp2 (that is, exp1 shifted right by exp2 bits (zero fill))
(<< exp1 exp2)	exp1 << exp2 (that is, exp1 shifted left by exp2 bits)
LOGICAL	
(= exp1 exp2)	TRUE if the expressions are equal (also ==)
(&& exp1 exp2)	Logical AND of the two expressions
(|| exp1 exp2)	Logical OR of the two expressions
(^^ exp1 exp2)	Logical EXCLUSIVE OR of the two expressions
(! exp)	Logical NOT of the expression
(> exp1 exp2)	TRUE if exp1 > exp2 (unsigned)
(< exp1 exp2)	TRUE if exp1 < exp2 (unsigned)
(s> exp1 exp2)	TRUE if exp1 > exp2 (signed)
(s< exp1 exp2)	TRUE if exp1 < exp2 (signed)
MEMORY	
(lpeek exp)	Returns the longword at address exp
(wpeek exp)	Returns the word at address exp
(peek exp)	Returns the byte at address exp

Here are some examples of complex expressions and how they evaluate:

EXPRESSION	VALUE	COMMENTS
(+ 2 3 3)	8	simple addition
(- 7 5)	2	simple subtraction
(* (+ 2 1) 3)	9	nested complex expressions
(+ `clientbp 100)		gives the client's text base
(lpeek (+ 4 `d0 `a0))		the addressing mode 4(a0,d0.1)
(+ (+ 3 1) (/ (* 2 8) 4))	8	in algebraic notation: (3 + 1) + ((2 x 8) / 4)

RANGES

A *range* is a way to specify a block of memory. A range consists of a start address and either an end address or a count. For most commands which take a range, the start and count values have defaults, so not all parts of the range need to be typed in.

A fully-specified range can look like "*start*,*end*" or "*start*[*count*]" (where *start*, *end*, and *count* are expressions, and the brackets and commas must be typed as shown). If the *end* address is present, it is the first address *not* included in the range: **100,200** specifies the range of addresses from 100 to 1FF, inclusive.

Various parts of the full specification can be omitted. A range which uses the default start address looks like ",*end*" (note the leading comma, showing that *start* was omitted) or "[*count*]" (the brackets set off *count* and show that *start* was omitted). If you want the default *count* the range just looks like "*start*" (which also looks like any other expression).

Here are some examples and the ranges they specify, assuming the default start is 100 and the default count is 80 (all numbers are hex):

RANGE	FIRST	LAST	COMMENTS
200[70]	200	26F	no defaults; *start*[*count*] form
200	200	27F	default count of 80
[70]	100	16F	default start; [*count*] form
80,100	80	FF	no defaults; *start*,*end* form
,200	100	1FF	default start; ,*end* form

Sometimes the start and/or count fields have no defaults; in these cases, they must be specified. Also, the *start*[*count*] form is not always allowed. This is the case for the **g** (go) command, where a count of bytes to execute does not make sense.

The default start and count values are listed in the descriptions for all commands which take a range argument.

STRINGS

Strings are used mainly by the **f** (find) and **s** (memory set) commands. A string consists of characters surrounded by double-quotes (*"string"*) or single-quotes (*'string'*). The string acts like the sequence of bytes represented by the characters between the quotes, with the following escapes:

ESCAPE	MEANING
\b	backspace ($08)
\e	escape ($1B)
\f	formfeed ($0C)
\n	linefeed ($0A)
\r	carriage return ($0D)
\t	tab ($09)
\\	the single character backslash ($5C)
\?	the special "wildcard" escape (see **find**)
\xXX	the byte $XX where XX is two hex digits

Quotation marks are also used to set off parts of commands and keep semi-colons from splitting up a command. See the chapter **PROCEDURES, IF, GOTO, DEFER, AND ALIAS** for more information.

CHAPTER 3

THE CLIENT, BREAKPOINTS, AND CHECKPOINTS: AN OVERVIEW

RUNNING THE CLIENT PROGRAM

Once there is a client ready to run (loaded with the **exec** command or with a *program* argument on the debugger's command line), you can cause it to run with the **g** (go), **t** (trace), **u** (untrace), and **v** (verbose-trace) commands. Collectively, these are called trace/go commands. What follows are cursory descriptions. See the chapter **THE CLIENT, BREAKPOINTS AND CHECKPOINTS: DETAIL** for more information.

The **g** (go) command runs the client at full speed. It will only stop when something exceptional happens, like hitting a breakpoint or causing a bus error. You can also stop it by hitting the stop button, if you have one. See the section **STOP BUTTONS** in the chapter **REMOTE DEBUGGING** for more information.

The **t** (trace) and **u** (untrace) commands cause the client to execute just a few instructions (sometimes just one) and then stop and display the registers. The **v** (verbose trace) command causes the client to execute one instruction, display those registers which have changed, then execute the next instruction, and so on. You can "trace through" a subroutine this way, or even trace through entire programs. The advantage is that the client doesn't get out of your control: the stub gets an *opportunity* to check memory checkpoints between each instruction, and you can stop the client after executing a certain number of instructions, even if those instructions are part of (say) an infinite loop. Naturally, tracing is significantly slower than full speed, because of all the processing going on in the stub. For just one or a few instructions, however, the speed doesn't really matter much, anyway.

Trace and untrace are almost identical. They differ in their treatment of the "trap" instruction. See the section **TRACE AND UNTRACE** in the chapter **THE CLIENT, BREAKPOINTS AND CHECKPOINTS: DETAIL** for more information.

BREAKPOINTS

Breakpoints allow you to stop the client program when it is about to execute the instruction at a specific address. A *counted breakpoint* allows you to stop the client the *n*-th time the instruction is executed.

You set breakpoints with the **b** command. When you set breakpoints and use the trace/go commands, the trace/go is stopped if the PC matches any breakpoint address and the count for that breakpoint (if any) has expired.

See the chapter **THE CLIENT, BREAKPOINTS AND CHECKPOINTS: DETAIL** for more information.

MEMORY CHECKPOINTS

Memory checkpoints cause a stop based on the contents of memory, rather than before executing a particular instruction. You set checkpoints with the **m** command. When you set checkpoints and do a trace/go, the trace/go is stopped when any of the checkpoint expressions becomes TRUE. Note the word "becomes" -- memory checkpoints are "edge triggered" rather than static.

Checkpoints are of two types: range and comparison. Range checkpoints cause a stop when a change is detected in a range of memory (e.g. an array or the screen). Comparison checkpoints cause a stop when the comparison evaluates to TRUE when previously it was FALSE.

Unlike breakpoints, which cause an exception in the processor, memory checkpoints need to be evaluated by the stub. The times when the stub gets a chance to evaluate checkpoints are called *opportunities*. Briefly, opportunities occur between instructions of a trace (verbose or normal) or untrace, and during the processing of a breakpoint (even if that breakpoint, because of its count, doesn't cause a stop).

Since memory checkpoints only get evaluated during an opportunity, they can only cause a stop at those times. Thus, all you know is that the expression became TRUE sometime between the previous opportunity and this one. In the case of trace and untrace, the opportunities come between every instruction. But in the case of a go command, you don't always know just when the previous opportunity was. Furthermore, the checkpoint might have become TRUE and then FALSE again since the last opportunity.

Breakpoints cause an opportunity even when their counts have not yet expired. You can provide an opportunity explicitly by placing a breakpoint with a count of "never" -- for instance, at the beginning of a loop. Such breakpoints never cause a stop by themselves, but always cause an opportunity.

See the chapter **THE CLIENT, BREAKPOINTS AND CHECKPOINTS: DETAIL** for more information.

CHAPTER 4

COMMANDS

The debugger prompts the user for a command with a colon (":"). Commands can also come from text files (see the **load** command), aliases (see the **alias** command), procedures (see the chapter **PROCEDURES**), and the client (see the **indirect** command). In each case, multiple commands can be specified on one line by separating them with semicolons (";"). If you really mean to use a semicolon (for example, in an argument to the **print** or **echo** commands), the argument containing the semicolon can be enclosed in quotation marks ('"') or apostrophes (also called "single quotes:" "'"). See the chapter **PROCEDURES, IF, ALIAS, AND DEFER** for more information.

The simplest kind of command is simply an *expression*. Typing an expression alone causes that expression to be evaluated, and the result to be printed in hex, decimal, octal, and binary. The result is also placed in the debugger variable `$ for future use. This kind of command (which usually just does some math) is called a *math command*.

In the following list of debugger commands, these syntax rules are used:

Brackets ("[ ]") surround optional items. Italics are used for the name of something you type: "d *range*" means the letter 'd' followed by a range specification. Three dots ("...") means the previous item can be repeated one or more times. Several alternatives enclosed in braces and separated by a vertical bar ("{ a | b }") means *either a* or *b*, but not both. Several items surrounded by both brackets and braces means you can use one of the things inside the braces, or nothing at all:

transcript [{ **off** | **flush** | **printer** | *file* [a] }]

means that the following forms are valid:

transcript	none of the alternatives
transcript off	the **off** alternative
transcript flush	the **flush** alternative
transcript printer	the **printer** alternative
transcript myfile	the *file* alternative without 'a'
transcript myfile a	the *file* alternative with 'a'

Note that sometimes the brackets and braces should really be typed: this is the case for the brackets in range specifications and the braces in an indirect operand to a memory checkpoint. The description of the command should make these exceptions clear.

The commands are divided into these groups:

SECTION	COMMANDS
Breakpoints and checkpoints	b, nb, m, nm
Trace and go	t, u, v, g
Memory handling	l, d, s, f
The Client and symbols	exec, args, getsym, sym, nosym, ?, where, stack
Registers and variables	set, x, vars, stubstate
Remote commands	wait, check, terminate, continue
Procedures and Aliases	procedure, plist, global, local, goto, alias, unalias, noalias
Files and aliases	read, write, load, unload, reload, bgoto, fgoto
Miscellaneous commands	bind, abort, #, transcript, gag, exit, q, quit, help, echo, print, if, indirect, !, dir

BREAKPOINTS AND CHECKPOINTS

You use breakpoints to make the client stop at a particular place. You use memory checkpoints to make the client stop when a particular set of conditions occurs. See the chapter **THE CLIENT, BREAKPOINTS, AND CHECKPOINTS: DETAIL** for more information.

b

b [*#index*] [*address* [{ *count* | **never** }]]

The **b** command alone lists the active breakpoints. With an *address*, it sets a breakpoint (with a count of one) at that address, and removes all other breakpoints there. With a *count*, it sets a counted breakpoint at the address. With **never**, it sets a breakpoint which will never cause a stop. (This is useful because it creates an opportunity for memory checkpoints.)

With no arguments, **b** lists all breakpoints. The list appears in a form suitable for saving (with **transcript**) and restoring (with **load**).

If the *#index* argument is present, the new breakpoint is placed in slot number *index*. If there was already breakpoint in that slot, the old one is removed first. This option is useful when using auto-execute aliases. See the section **AUTO-EXECUTE ALIASES** in the chapter **PROCEDURES, IF, GOTO, DEFER, AND ALIAS** for more information.

```
Examples:

b             list all breakpoints in the table
b .main       set a breakpoint to stop at the label "main"
b .main 1     same as above
b .loop 3     set a breakpoint to stop the third time the
              instruction at "loop" is executed
b #4 .loop    set a breakpoint at .loop in slot #4, re-
              placing whatever breakpoint was in that
              slot, and replacing any other breakpoint at
              that address.
```

nb [{ *address* | *#index* }]

The **nb** command alone removes all breakpoints. It asks for verification first: space, 'y', and 'Y' mean "go ahead." Any other key aborts. With *#index*, it removes breakpoint number *index*. With *address*, it removes all breakpoints at *address*.

```
Examples:

nb            clear all breakpoints (asks for verifica-
              tion)
nb #1         clear the breakpoint in slot number 1
nb .loop      clear all breakpoints at the label "loop"
```

m
m [*#index*] *range*
m [*#index*] *address.size*
m [*#index*] *address* [*.size*] *op* { *value* | {*iaddr*} | **old** }

The **m** command alone lists all memory checkpoints. The list appears in a form suitable for saving (with **transcript**) and restoring (with **load**).

With a *range*, it sets a range-type checkpoint. The default count for *range* is 2 (a word); there is no default start.

With an *address* and *size* it sets a comparison checkpoint which will become TRUE when the value there changes. The command "**m** *address.size* **!= old**" does the same thing.

Note that *address* may be a complex expression; see the examples.

The last form sets a comparison checkpoint, as follows:

The *.size* field is either **.b**, **.w**, or **.l**. The size field can be omitted, but if it is present, there must be no space between it an the *address* argument. That is, "**.flag.b**" is correct for a byte-size

checkpoint address and size, while ".flag .b" is not. If *.size* is missing, the default is .w (two bytes).

(Unfortunately, since the memory checkpoint command treats the trailing part of the *address* argument as a size indicator (.b, .w, and .l), you can't have a checkpoint on a compound symbol specifier whose last component is a two-character symbol starting with a period ('.').)

The *op* (operator) can be one of the following:

OPERATOR	COMMENTS
s> s<= s>= s<	Signed comparison
u> u<= u>= u<	Unsigned comparison
== !=	Equal, not equal
vs vc	Overflow set, overflow clear
> <= >= <	Same as signed
=	Same as ==
cs cc	Same as u< and u>=

If the operand is enclosed in braces, it is indirect: *iaddr* is the address of the operand used for the comparison. When the checkpoint is evaluated, as many bytes are fetched from *iaddr* as are used at *address* -- that is, the *size* of the checkpoint controls them both.

If the operand is the word **old**, it means to use the initial value at *address* for the subsequent comparisons. This lets you catch a byte, word, or long value when it changes, and is faster than the equivalent range-type checkpoint. The "old" value is reloaded internally at the start of each trace/go command.

Otherwise, the operand is evaluated as an expression, and its value is used for the comparisons.

Note that for the indirect comparison type, a pair of braces encloses the second operand ("{*iaddr*}" in the example). You really type the braces; they are not there to show syntax.

If the *#index* argument is present, the new checkpoint is placed in slot number *index*. If there was already a checkpoint in that slot, the old one is removed first. This option is useful when using auto-execute aliases. See the section **AUTO-EXECUTE ALIASES** in the chapter **PROCEDURES, IF, GOTO, DEFER, AND ALIAS** for more information.

```
Examples:

m                   List memory checkpoints
m .foo > 10         Stop when foo.w (default size) > 16 ($10).
m .foo > old        Stop when foo.w exceeds its initial value.
m .buf[10]          Stop when anything in the 16 bytes starting at  buf
                    changes.
m #3 .buf[10]       Same as above, but place the checkpoint in slot #3.
m 438.1             Same as "m 438[4]"
m (+ 2 2).w         Same as "m 4.w" and "m 4[2]"
m 438.1 < {43C}     Stop when the (long) value at 438 is less than  the
                    (long) value at 43C.
m 12030 != old      Stop when 12030.w changes value.
m 12030[2]          Stop when 12030.w changes value (see below).
m 12030             Same as above (default count is 2).
```

Notice the last three examples. They all seem to do the same thing: stop when either of the two bytes starting at 12030 changes. The range type is less desirable, though, for checking small areas (one, two, or four bytes), because the range type computes the CRC (cyclic redundancy check) for the range, and compares it to what the CRC was when the trace/go started. This takes a long time, and, more importantly, changes can actually be missed if both the original and new contents result in the same CRC value.

nm [{ *address* | *#index* }]

The **nm** command alone, like the **nb** command, clears all the memory checkpoints. It asks for verification first: space, 'y', and 'Y' mean "go ahead," any other key aborts. With *address*, the command clears all checkpoints with that address. With *#index*, the command clears checkpoint number *index*.

```
Examples:

nm          Clear all checkpoints.  Ask
            for verification first.
nm #3       Clear checkpoint number three.
nm .flag    Clear all checkpoints with the
            value of "flag" as the ad-
            dress.
```

TRACE AND GO

The trace and go commands are the only ones which cause the client to execute instructions. When they stop, the reason for the stop is printed (e.g. "Breakpoint"), the client's registers are displayed, and the instruction at the (new) PC is disassembled.

When the conditional branch instructions are disassembled at an address that matches the current PC, either because you used the **set** command with no arguments, or after a trace, or during a verbose trace, the letter 'T' or 'F' will appear between the address and the opcode: means the condition is false, and the branch will not be taken. This applies to other conditional instructions as well, such as **seq**. It does not apply to conditional floating-point instructions.

The CPU register display includes a mnemonic display of the SR. The mnemonics are as follows:

SU	supervisor mode
TR	trace bit set
IPL=*x*	*x* is the IPL
CS, CC	carry set, clear
ZR, NZ	zero set, clear
VS, VC	overflow set, clear
XS, XC	extended carry set, clear
MI, PL	sign bit set, clear

t [{ *count* | **x** | **w** }]
The **t** (trace) command causes the client to execute in "trace mode." With no count, the client executes one instruction. With a count, the client executes that many instructions. With a count of 'x', the client executes "forever" -- until a breakpoint, memory checkpoint, or exception causes a stop.

With a count of 'w', the t command executes one instruction at full speed. This is handy if it is a "jsr" or "bsr" instruction: in those cases, the whole subroutine is executed all at once, and the trace stops at the instruction following the "jsr" or "bsr."

See the section **TRACE AND UNTRACE** in the chapter **THE CLIENT, BREAKPOINTS AND CHECKPOINTS: DETAIL** for more information.

u [{ *count* | **x** }]
The **u** (untrace) command is just like the **t** (trace) command, except that the client executes in "untrace mode." This means that trap-type instructions are *not* treated specially. Note that **uw** doesn't make sense and isn't allowed.

v [{ **u** | **w** }] [*count*]
The **v** (verbose-trace) command begins another kind of trace: before each instruction is executed, it is disassembled and displayed on the screen. After it executes, the values of all registers which changed are displayed. Then the next instruction is disassembled, and so on. Use ^S to pause the trace, ^Q to continue it, and ^C to stop it.

With no *count*, the v command will trace forever (until a stop or until ^C is used). The verbose trace executes in "trace" mode, meaning that a trap handler is executed as though it were a single instruction. With a *count*, that many instructions are disassembled and executed.

With 'u', this command traces instructions in "untrace" mode.

With 'w', instructions are traced (in trace mode), but the **bsr** and **jsr** commands are treated specially: they are executed at full speed, like the **tw** command. Also, the **vw** command stops when it encounters the **rtd**, **rtr**, **rte**, or **rts** instruction.

Examples:

t	trace one instruction
t 4	trace four instructions
tx	trace forever (until a stop)
tw	execute through a subroutine at full speed
u	untrace one instruction
u 4	untrace four instructions
ux	untrace forever (until a stop)
v 9	verbose-trace 9 instructions
v	verbose-trace forever (until a stop)

g [*range*]

The **g** (go) command causes the client to execute at full speed. It turns control of the computer over to the client, after setting the breakpoints. The go will only stop when a breakpoint, exception, or the stop button causes a stop.

The default start address for *range* is the current PC. The default count means "forever." In fact, you can't specify a count for this range; you can only use the "*start*" or "*start*,*end*" or ",*end*" forms of the range. If you specify an *end*, a temporary breakpoint is set at that address. This is sometimes called "go until," because you are saying, "Go until this spot, then stop." See the examples below for more.

Examples:

g	Go forever (until an exception or breakpoint)
g .main	Set PC to main, then go.
g ,.subproc	Set a temp. breakpoint at subproc, then go.
g .main,.subproc	Set PC to main, set a breakpoint at subproc, and go.

Note that the "go until" forms actually go until the end address *or*

some other exception. Note also that they clear the temporary breakpoint when the go stops, for whatever reason. Finally, note that there must be at least one breakpoint slot available for the "go until" to work.

MEMORY

The following commands display and set memory in various ways.

l [*range*]
The l (list) command disassembles memory into 68000 mnemonics. The default start for *range* is the place where the last l command left off, but the **exec** command and all the trace/go commands set the default start to the new PC after the command is finished. The default count for *range* produces 12 lines of disassembly, not any particular number of bytes.

The list command takes the *range* as a guideline: the last instruction it disassembles is the one containing the last byte of the range, even if the instruction extends beyond that byte.

The disassembly listing you get looks something like this:

```
myprog:
00012214    move.l    #$12214,a1        myprog
0001221A    lea.l     $12214(PC),a1     myprog
0001221E    move.l    a1,$12004         myvar1
00012224    move.l    $12004,$12008     myvar1; myvar2
0001222E    move.l    $4BA.w,d1         clock
00012232    addq.l    #3,d1
00012234    bra.b     $12214            myprog
```

The listing has four columns: the disassembly address is printed in the first column, then the opcode and size, then the operands, and finally any symbols matching the values used in the operands.

If there are any symbols with the same value as the address of the instruction being disassembled, they are printed out above the disassembly line (like the label "myprog:" above).

The names in the right-hand column are the names of symbols matching the operands, separated by commas. If there are two numeric operands, and there is at least one symbol matching each of them, the symbols for each operand are separated by a semicolon.

Operands which are less than $100 do not get matching symbols printed: it would be too confusing, since so many symbols lie in this range, and picking out the one which mattered in any particular instruction

would be impossible for the debugger and difficult for the user. You can **list all** the symbols with a given value using the **where** command.

If you are on a 68020 or 68030, the 68881 floating-point coprocessor instructions are disassembled, and the 68030 PMMU instructions are disassembled. (The 68851 PMMU shares some instructions in common with the 68030's PMMU, but no effort has been made to disassemble for the 68851 specifically.) See the description of **discpu** in the chapter **DEBUGGER VARIABLES** for more information.

If an instruction can not be disassembled, the listing will show ".dc.w *xxxx*" where *xxxx* is the value at that address.

Examples:	
l	list 12 lines starting where the last l left off
l .main[10]	list from the label "main" up to and including the instruction which ends at or after main+$10
l `pc[1]	list the (single) instruction at the current PC

d [{ **w** | **l** }] [*range*]
The **d** (dump) command dumps memory. The default start for *range* is the place where the last dump left off. The default count is 128 bytes. If **w** or **l** is specified, the command dumps words or longwords, respectively. If neither is present, bytes are dumped.

The memory dump consists of lines with the starting address on the left, the memory bytes (or words or longs) in the middle, and the ASCII representation of the memory on the right. The ASCII representation shows the character associated with each byte in memory, if that character is in the "printing character" set (32-127,160-254 on the Atari ST). See the section **OPTIONS** in the chapter **DB: THE ATARI DEBUGGER** for more information.

The *range* argument rounded up to a multiple of the size (2 for **w** and 4 for **l**).

The **d** command alone, with no range or size specifier, dumps 128 bytes starting where the last dump left off, and *in the last format used*. The command "**dl0**" dumps 32 longwords starting at zero; if followed simply by "**d**" another 32 longwords will be dumped: the size specifier is preserved. A **d** command with a *range* will reset the size to word, long, or byte (if neither **w** nor **l** is specified).

Examples:	
d	dump 128 bytes in the last format
dw	dump 64 words (128 bytes)
dl	dump 32 longs (128 bytes)
d [10]	dump 16 bytes
dl 8[1]	dump the bus-error exception vector
dw `sp	dump the stack (as words)

s [{ **w** | **l** }] [*addr* [*value* ...]]
s [{ **w** | **l** }] *range value*
s *addr string*

The **s** (memory set) command is used to change the contents of the client's memory. In the first two forms, the presence of **w** or **l** indicates that words or longwords are to be set. If neither **w** nor **l** is present, bytes are set.

In the first form, if any *value*s are present, the byte (or word or longword) at *addr* is set to the first *value*. If there are many *value*s, they are placed in memory consecutively starting at *addr* and incrementing *addr* by the appropriate number (1, 2, or 4 bytes).

If *value* is not present, memory is set interactively. A memory address is printed on the screen, followed by the (byte, word, or long) value currently there. At this point you can just hit the "return" key to skip to the next location, or type a new value (plus "return") to be placed at that address, or a single period (".") (plus "return") to terminate the set command. ^C will also terminate the command.

The second form fills the specified *range* with the (byte, word, or long) *value*. If the size of the *range* is not a multiple of the unit (1, 2, or 4 bytes), it is rounded up.

The third form sets the memory starting at *addr* to the bytes represented by *string*. The string is placed in client memory as-is: it is not null-terminated.

If exactly two or four bytes are being set, and they start at an even address, the **move.w** or **move.l** instructions are used. This can be important if the address in question refers to a memory-mapped I/O device.

See the section **STRINGS** in the chapter **EXPRESSIONS, RANGES, AND STRINGS** for more information.

Examples:	
s 400	set bytes interactively starting at $400
sl 400	set longs interactively starting at $400
sw 380[80] 1234	Fill 64 words with $1234
sw 6F0 FF20 12 0 -2	set these words at 6F0..6F7: FF20 0012 0000 FFFE
s 6F0 "Testing\r\n\x00"	Set a C-type string (null-terminated) at 6F0

f [{ **w** | **l** }] *range value* ...
f *range string*

The **f** (find) command prints out the beginning address of areas of memory within *range* which match the target pattern. It also sets the debugger variable **$** to the address of the first match.

The first form takes a size specifier (**w** for word, **l** for long, or nothing for byte) and a sequence of *values*. The *values* are treated as being of the indicated size, and are used as the target pattern for the find. The asterisk ("*") is a special *value* which will match *any* byte (or word or long): it is a wildcard.

The second form takes a *string* as the target of the find. See the section **STRINGS** in the chapter **EXPRESSIONS, RANGES, AND STRINGS** for more information.

For the find command (and only the find command), the string escape "\?" is a one-byte wildcard, which matches any value.

Note that each individual *value* is expanded or truncated to the size of the find (byte, word, or long), then split into the component bytes. Ultimately, the target is always a sequence of bytes. This means that a **fw** or **fl** command can actually find matches at odd boundaries.

The find command always lists the address of each match. If what you are looking for is found often, the list will be long and useless. You might consider using the **gag** command to suppress the list; the **$** variable will still be set to the address of the first match. See the **gag** command for details.

Examples:	
f1 0,400 FC0008	Find the four bytes 00 FC 00 08 in the range 0..3FF
fw `a1[100] 100 * 300	Find the six bytes 01 00 * * 03 00
f `a7[100] "x\?z"	Find the three bytes 78 * 7A

THE CLIENT AND SYMBOLS

These commands load the client and manage the debugger's symbol table. See the chapter **SYMBOLS AND DEBUGGER VARIABLES** for more information.

exec [{ *program* [*args* ...] | **on** | **off** }]

The **exec** command loads the named *program* and sets it up for execution. It also loads the symbols from that *program*, and sets the GEMDOS command-line arguments to *args*, if any. The debugger variable "clientbp" is set to the basepage of the loaded program. Finally, the basepage information of the client is displayed.

With no arguments, **exec** displays the basepage information at `clientbp (usually the basepage of the last-execed client).

Normally, when a client uses Pexec and executes a child, a message is sent to the debugger with that child's basepage address. The "**exec off**" command disables this. "**Exec on**" re-enables it. When remote debugging with the resident stub, exec is off by default; you can enable it with "**exec on**" when the client is stopped (e.g. because you hit the stop button).

When you start the debugger with a *program* argument on the command line, it performs an **exec** command for that *program* and any *args* following it.

When you are not remote debugging, you can use **exec** to load clients. You must exercise care, however. Once you load one client, you may not be able to load another. The first must either terminate or execute the GEMDOS call Mshrink, to make memory available to the second client. Also, if you stop one client while it is in a GEMDOS trap, then try to use the exec command to load another client, GEMDOS will bomb ungracefully: it is not reentrant. See the chapter **OPERATING SYSTEM CONSIDERATIONS** for more information.

You can't use **exec** to load programs when remote debugging. The first form still works, however, to display basepage information, and the **exec on** and **exec off** commands work.

```
Examples:

exec                        display basepage information
exec myprog.prg             load myprog with no arguments
exec myprog.prg -o xyz      load myprog with command-line
                            arguments "-o xyz"
```

args [*args* ...]

The **args** command sets the command-line arguments for the most recently **exec**-ed client to *args*. If there are no *args*, the command-line arguments in the client's basepage are cleared out.

```
Examples:

args          clear out the argument area of the client
args -o xyz   set the argument area to "-o xyz"
```

getsym *program* [*textbase*]

The **getsym** command loads symbols from the named *program* file. GEMDOS programs are relocatable, so you must supply the *textbase* argument to relocate the symbols. Some programs, notably those which are placed in ROM, are absolute, and need no relocation. You don't need a *textbase* argument for these.

This command is used to get symbols for a program which is already loaded, usually when remote debugging. Be sure that the program file you load symbols from is the same program file that the client was loaded from; otherwise, the symbols may not match up.

The **exec** command loads symbols from the client program file automatically: no additional **getsym** command is necessary.

When not remote debugging, do not use this command if you have stopped the client in the middle of executing a GEMDOS system call: this command uses GEMDOS to read the file, and GEMDOS is not reentrant. See the chapter **OPERATING SYSTEM CONSIDERATIONS** for more information.

```
Examples:

getsym myprog.prg `pc   load symbols from myprog.prg, relocat-
                        ing them by the current PC. Right
                        after an exec, `pc is the text segment
                        base address of the process.
getsym myfile.rom       load symbols (absolute: no relocating)
```

sym *name value*

The **sym** command creates a new symbol in the symbol table, *Name* and *value* are used for its name and value. The new symbol will be treated just like all the existing symbols in the symbol table.

nosym

The **nosym** command deletes the entire symbol table. Because of the way the debugger stores symbols, this memory is not recoverable: if the symbol table took up 12K and you use the **nosym** command, you will simply lose that 12K from the debugger's memory space for the rest of the session. **Db** will ask for verification before doing this, and will report that the memory was "dropped on the floor."

? [*symbol*]

The **?** command displays the symbol table. If a *symbol* argument is present, it lists from that symbol onward. Otherwise, it starts at the beginning. Use ^S to pause the listing, ^Q to resume it, and ^C to abort.

The symbol list consists of the symbol's name, its value, and its type, both in hex and in English: each bit of the type has a name associated with it, and if that bit is set the name is printed. If the bit is clear no name is printed. In parentheses, the name of the symbol's segment is displayed using Mark Williams C's conventions, if the type field indicates one of the MWC segments.

```
Examples:

?           list the whole symbol table.
? main      list the symbol table, starting with "main"
? .main     same as above
```

where [*expression*]

The **where** command shows symbols with values at or before the value of *expression*. If *expression* is absent, the current PC is used.

Where shows the value of *expression*, then lists the symbols with that value. If there are none, it looks for the next lower valued symbol, and lists all symbols with that value, with their offset from the *expression*.

Consider the following examples, assuming that the symbols "myprog" and "start" have the value 12000 (hex), "main" has the value 12030, and "loop" has the value 12038.

	COMMAND	OUTPUT
(a)	where 12030	12030: main
(b)	where 12034	12034: main + 4
(c)	where 12038	12038: loop
(d)	where 1203A	1203A: loop + 2
(e)	where 12000	12000: myprog, start
(f)	where 12006	12006: myprog, start + 6
(g)	where 30040	12FFE: loop + 1E002
(h)	where 0	No symbols at or before 0.

The last few examples need more explanation. Examples (e) and (f) show that two symbols with the same value will both be printed if necessary. Example (g) shows that the output of **where** is not always meaningful: 30040 is probably well beyond the intended scope of the label "loop," but since that is the symbol with the next lower value, it is displayed. Example (h) shows what happens when there are no symbols at or before the value of *expression*.

The **where** command with no argument shows the **where** list for the current PC. This is useful when a trace/go command has stopped because of, say, a bus error: you can find out what procedure the PC is in just by typing **where**.

stack

The **stack** command tries to perform a stack traceback using the Alcyon C calling conventions. The traceback listing always starts with the current PC, and shows a **where**-type list for that location. Then the frame pointer (a6) and stack pointer (a7) are reloaded like an unlk (unlink) instruction, and the new PC is taken off the stack. The new PC and a **where**-type list for it are printed, and the process repeats.

The traceback stops when the end of the stack is reached (i.e. the new frame pointer is zero), or there is some error in the traceback (odd or zero address, etc.).

The **stack** command tries to be clever: if the current instruction is "link," it deduces that you are at the start of a procedure, and that the top longword on the stack is the return PC. If the current instruction is "rts," it assumes that the unlk instruction has already executed, and, again, the top longword on the stack is the return address. These are not always valid assumptions, but they work well enough for un-optimized Alcyon C compiler output, and for most other compilers using the link/unlk conventions.

If your program does not follow the C calling conventions, or follows them differently (e.g. using something other than a6 as the frame pointer), this traceback will do you no good.

REGISTERS AND VARIABLES

These commands manipulate the client's registers and the debugger variables.

set [*variable* [*value*]]
x [*variable* [*value*]]

The **set** command alone displays the client's CPU registers: the PC, both stack pointers, the SR, and all the data and address registers. In addition, it disassembles the instruction at the PC (like **1`pc[1]** would).

With a *variable* argument, **set** displays the value of the given variable. With both a *variable* and a *value* argument, *variable* is set to *value*.

The **x** command is just an alias for **set**: it's there for compatibility and because some people like one-character commands.

When the conditional branch instructions are disassembled because you used the **set** command with no arguments, after a trace, or during a verbose trace, the letter 'T' or 'F' will appear between the address and the opcode: 'T' means the condition is TRUE, and the branch will be taken; 'F' means the condition is false, and the branch will not be taken. This applies to other conditional instructions as well, such as **seq**. It does not apply to conditional floating-point instructions.

The CPU register display includes a mnemonic display of the SR. The mnemonics are as follows:

SU	supervisor mode
TR	trace bit set
IPL=*x*	*x* is the IPL
CS, CC	carry set, clear
ZR, NZ	zero set, clear
VS, VC	overflow set, clear
XS, XC	extended carry set, clear
MI, PL	sign bit set, clear

Examples:

x	show the CPU state
set sr	show the SR
set sr 0700	set the SR to 0700 (IPL 7)
set t1	show the debugger variable t1

vars

The **vars** command lists all the debugger's built-in variables. It is provided as a reminder.

stubstate

The **stubstate** command displays the stub variables and their values. See the section **DEBUGGER VARIABLES** in the chapter **SYMBOLS AND DEBUGGER VARIABLES** for more information.

REMOTE DEBUGGING COMMANDS

The following commands only have meaning when remote debugging. They are not available when debugging on a single machine. See the chapter **REMOTE DEBUGGING** for more information.

wait

The **wait** command is used to synchronize the head and the stub after the slave machine is reset, or the **terminate** or **continue** commands are used, or any other time that the head is out of synch.

check

The **check** command is used to check the integrity of the connection between the head and the stub. It is meant for debugging the debugger. It presents you with a list of keys it responds to, and begins a feedback test. When an asterisk ('*') appears, a successful turnaround has occurred. When the letter 'S' appears, the head could not send a command to the stub. When the letter 'T' appears, the stub did not respond to the command. When the letter 'Z' appears, the size of the responding packet was not as requested. In normal debugging, this command is not used.

terminate

The **terminate** command causes the client program to terminate. What actually happens is that the stub executes the GEMDOS call Pterm, which terminates whatever the current GEMDOS process is. Thus, this can be used to terminate the client, or a child of the client.

continue

The **continue** command gives the stub a "go" command, but does not wait for the "go" to stop. It returns immediately to the command prompt. At this point, you may use any command which does not require access to the stub state, the stub variables, the client registers, or any other memory on the slave machine or interaction with the stub. Basically, this means the **getsym** command and math commands (i.e. just an expression at the command prompt). Two more commands you can use after a **continue** are **wait**, to resynch when the "go" stops, and **quit**, to leave the head while the client is still running. Finally, the !

(shell) command can be used to run a program locally.

PROCEDURES AND ALIASES

procedure [*name* [*args* ...]]
The **procedure** command allows you to define procedures. See the chapter **PROCEDURES, IF, GOTO, DEFER, AND ALIAS** for more information.

The **procedure** command alone lists the names and argument lists of all procedures currently known by the debugger. This can serve as a reminder of what a procedure does and how to use it, if the procedure's name and its arguments' names are well chosen.

plist [*name* ...]
The **plist** command lists procedures (including name, argument list, and body). With no arguments, it lists all procedures currently known by the debugger. With one or more arguments, it lists those procedures. The list appears in a form suitable for saving (with **transcript**) and restoring (with **load**).

global [*name* ...]
The **global** command creates global variables by name. One or more *names* can be specified to create one or more global variables. If one of the *names* already exists as a global, nothing happens.

With no arguments, all global variables *and their values* are listed. The list appears in a form suitable for saving (with **transcript**) and restoring (with **load**).

If a *name* argument begins with a minus sign ("-"), any global variable with that *name* is *removed*.

local [*name* ...]
The **local** command creates local variables by name. One or more *names* may be specified to create one or more local variables. Local variables are visible only inside the procedure where they were created, or at the top level (outside all procedures). When the procedure exits, they are removed. They do not hold their values from one invocation of a procedure to another.

With no arguments, all local variables *and their values* are listed. The list appears in a form suitable for saving (with **transcript**) and restoring (with **load**). (This is mainly useful for debugging procedures, not for actually saving the state of local variables.)

If a *name* argument begins with a minus sign ("-"), any local variable with that *name* is *removed*.

goto *label*

The **goto** command causes a jump in a procedure from the current point to the specified *label*. Labels in procedures look like comments ("#:label").

The **goto** command can be used to create very powerful constructs. With auto-execute aliases, the possibilities are virtually unlimited: a breakpoint can cause a script to be loaded or a procedure to execute, and with **if** and **goto** anything can happen.

SAMPLE PROCEDURE

```
procedure sample maxval
  # This procedure shows the first `maxval nonnegative integers.
  local count ; set count 0

  if (< `argc 1) abort Too few args to procedure sample

  #:loop
    print -n -d `count
    set count (+ `count 1)
    if (< `count `maxval) goto loop
  print
.
```

After loading this procedure (that is, typing it in or loading it from a script), this might happen:

```
: sample
Too few args to procedure sample
: sample 3
0 1 2
: sample @20
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
:
```

See the chapter **PROCEDURE, IF, GOTO, DEFER, AND ALIAS** for more information.

alias [*name* [*expansion*]]

The **alias** command lets you create your own commands which are combinations of other debugger commands. The easiest explanation is by example: if you use the command "**alias foo dl 0[8]**" and later enter the command "foo," the expansion of "foo" (in this case, "dl 0[8]") will be executed. In other words, once you alias a *name* to an *expansion*, subsequent uses of that *name* as a command result in the *expansion* being used in its place.

There may be several commands in an *expansion* -- enclose the whole *expansion* in quotes, and separate the commands with semicolons, like this: **alias show "dw .var1[2] ; dw .var2[2]"**

An alias may contain other aliases. For instance, if you alias "dumpword" to expand to "dw", the above alias could be written **alias show "dumpword .var1[2] ; dumpword .var2[2]"**

To change an alias, just redefine it with another **alias** command. To remove an alias, use **unalias**.

Alias with no arguments lists all aliases. **Alias** with one argument displays the alias for that *name*. The list appears in a form suitable for saving (with **transcript**) and restoring (with **load**).

If an alias contains itself, or contains an alias which contains the first, an infinite loop can result. To prevent this, the debugger will only expand 256 aliases in one line; more than that, and it assumes an infinite loop has occurred and reports the fact. The debugger might also run out of memory for keeping track of aliases before this happens.

See the chapter **PROCEDURES, IF, GOTO, DEFER, AND ALIAS** for more information.

unalias *name ...*
The **unalias** command deletes all the *names* from the alias list. You can replace an alias simply by redefining it: you don't need to remove it first.

noalias
The **noalias** command deletes all aliases. It asks for verification before doing so.

FILES AND SCRIPTS

These commands have to do with data files and script files.

read [*file* [*address*]]
The **read** and **write** commands are used to transfer data from disk to the client's memory and back. The "disk" in question is always the one local to the head; this is not the same as the stub's disk in a remote-debugging system.

Read with no arguments displays the starting address and size of the last file read. With two arguments, it reads the named file into the client's memory starting at the given *address*.

When not remote-debugging, a third form is allowed: with a *file* argument but no address, **read** will use the operating-system call **Malloc** to allocate enough memory for the named *file*, then read it in to that memory. This is useful for patching a file, because you don't care where it gets loaded in. Note that there must be enough memory

available to the operating system for the file, or the Malloc will fail. This is especially a problem if you **exec** a program but don't let it return memory to the OS: it is likely to have all of memory allocated to it.

write *file* [*range*]

The **write** command is the companion to **read**. With both a *file* and *range* argument, it writes the memory in that range to the file. With only a *file* argument, it uses the start and size information from the **last read** command. If the file already exists, the user is asked to verify that he wants to overwrite it.

load *file*

The **load** command causes debugger commands to be read from a file rather than from the keyboard. The file should contain normal ASCII text, with lines separated with CR/LF. Each line is read in and interpreted exactly as if it was typed at the debugger's colon (":") prompt. Other input, such as verification, still comes from the keyboard.

These files are called *scripts*. By convention, script files (except for the startup files **db.rc** and **rdb.rc**) have the extension ".DB," as in "SETUP.DB."

A script can contain the **load** command itself. In this respect, **load** can be used as something of a subroutine call. *No check is made for infinite loops.*

Some commands are only meaningful when used in a script; they are **bgoto**, **fgoto**, **unload** and **reload**.

In a script file, long commands can be split onto several lines. When a line in a script ends with a backslash ('\'), the next line is tacked onto it as though it was a continuation of the same line. This is not the case for lines read from the keyboard.

Example:

```
print -s (lpeek (+ \
                 (lpeek (+ `clientbp 24)) \
                 2c))
```

Because of the '\' characters, the the three lines are merged into one. (This command prints the name of the first environment variable in the environment of the *parent* of the current process. It is a useful exercise for the reader to examine exactly how it does this.)

Long lines split with '\' will cause line numbers in error messages to be inaccurate: the line number counter counts logical lines, not physical lines.

unload

Unload causes the script currently being loaded (with the **load** command) to end. If you think of **load** as a subroutine call, this can be used as a premature "return" statement. This amounts to an **fgoto** command to the end of the script, but is faster.

It is an error to use this command when not loading a script.

reload

Reload causes the script currently being loaded to be rewound to the beginning. It amounts to a **bgoto** to the start of the file, but is faster.

It is an error to use this command when not loading a script.

bgoto *label*
fgoto *label*

The **bgoto** and **fgoto** commands change the flow of control in scripts. The *label* argument is the exact text of the line you wish to go to, and may only be one word. Usually, this is a comment, like "**#begin**" or "**#loop.**" For example, consider the following text file:

```
echo line 1
xt0 0
#begin
print -n `t0
if (< `t0 10) bgoto #begin
echo
echo end of loop
```

Loading this file will cause the following output:

```
line 1
0 1 2 3 4 5 6 7 8 9 A B C D E F
end of loop
```

The **fgoto** command has the limitation that the line containing its *label* argument must be *after* the current position in the script. The **bgoto** command rewinds the file, then compares each line against the *label* argument, while **fgoto** does not rewind the file first. If the label is after the current point in the file, **fgoto** is faster, especially in large scripts.

It is an error to use these commands outside of a script.

As a rule, script files are best used to for setup scripts and loading procedures. Use aliases for little things you plan to do more than once, and procedures for complex things with looping and such. Aliases and procedures are kept in memory, not on disk, and in procedures, the labels are indexed so a **goto** executes much faster. The **fgoto** and **bgoto** commands are really leftovers from the days when the debugger didn't

have procedures.

MISCELLANEOUS COMMANDS

bind [*string* [*code*]]

The **bind** command allows you to bind a string to a key. After that, when you use that key, the *string* will be used as if it had been typed from the keyboard. *code* is the ASCII code of the key to bind to: codes 0 through 31 are allowed (the control keys), except for 13, which is carriage-return. (Rebinding carriage-return would be disasterous!)

With no arguments, **bind** lists the current key bindings. The list appears in a form suitable for saving (with **transcript**) and restoring (with **load**).

With one argument, **bind** prompts you to hit the key to which you want the *string* bound. This is useful if you don't know the key's code offhand. You should use the actual keystroke here: hold down "Control" and press the key in question.

```
Examples:

bind                        list bindings.
bind "1`pc[1]\r" 1          bind the string to ^A.
bind "dw.xlist[10]\r"       prompt for a key; bind to that key.
```

abort [*args* ...]

The **abort** command prints out its arguments (usually an error message of some sort) in exactly the same way as the **print** command, and then it returns to the command prompt. Any script which was loading, procedure which was executing, alias which was executing, or deferred commands which were pending are forgotten: the debugger is reset to the very top level, and waits for user input.

#

The # command introduces a comment. The rest of the line is ignored. The # character isn't properly a command at all: it is processed by the command-line reader. When it appears in the position where a command is expected, the rest of the line it's on are thrown out.

transcript [{ *file* [**a**] | **off** | **flush** | **printer** }]

Transcript starts a transcript of all the output from the debugger, and all the input from the user. The *file* argument tells what file to keep the transcript in. When you leave the debugger for any reason (short of resetting the head machine) the transcript file is saved and closed. The command "**transcript off**" stops the transcript explicitly.

and saves and closes the file.

Transcript printer causes debugger output to go to the printer as well as the screen. Note that output is buffered in a transcript buffer, so the printer will always be slightly behind what is on the screen. (The buffer size might be as much as 4K.) BIOS calls are used to send the transcript data to BIOS device 0.

Transcript flush flushes the buffered transcript information explicitly. This is especially useful when transcripting to the printer, because otherwise recent information will not have been printed yet.

Transcript alone tells the state of transcripting (on or off).

The **a** option to the **transcript** *file* form means "append" and causes the transcript to be appended to the transcript file; otherwise, any existing file with that name is removed (without warning).

The **transcript** command must be used carefully unless you are remote debugging. On a single-machine system, you should be careful not to stop the client while it is processing a GEMDOS call. When the transcript buffer fills up, it needs to be flushed to disk, and this is done with GEMDOS calls. If the client is in the middle of a GEMDOS call, this will crash your system.

Note that the only ways to stop the client while it is in GEMDOS are to use the **u** (untrace) command when at a "**trap #$1**" instruction, use the stop button, or cause a bus error or other exception in GEMDOS. If you avoid these conditions, transcripting should be safe even when debugging locally. See the chapter **OPERATING SYSTEM CONSIDERATIONS** for more information.

gag [{ **on** | **off** }]

The **gag** command causes output to be suppressed. With no arguments, or with the **on** argument, output is suppressed until the next time the debugger needs to wait for user input. With the **off** argument, the suppression stops, and output resumes. You might use the **gag** command in conjunction with **transcript**, so the information goes to the transcript file without also being printed on the screen:

```
: transcript disasm ; gag on ; l.main[2000] ; gag off ; transcript off
```

The above commands disassemble eight kilobytes of code and place the disassembly in a file called "disasm." The disassembly is not displayed on the screen. Without the **gag** command, the text would scroll by on the screen, taking a much longer time.

exit

The **exit** command is used to terminate the stub and leave the debugger. Whether remote debugging or not, this command causes all machines involved to return to a quiet state. In a single-machine model, the

debugger will remove itself and the stub and return to the desktop or shell. If you are remote debugging, the head will tell the stub to remove itself, then remove the communications layer from the slave, and finally remove the communications layer from the master. Again, both machines should return to the desktop or shell.

q
quit

The **quit** command exits the debugger. If you are remote debugging, it does *not* cause the slave to terminate or exit or even to stop. It can be used after a **continue** command, or after stopping a **wait** condition with ^C, to let the client run while you do something else on the master machine.

When not remote debugging, this command is identical to **exit**.

help [*topic*]

The **help** command alone lists the debugger commands, the operators for complex expressions, and the built-in variables, with a brief reminder of what they do.

With a *topic*, the command gives a little help on that topic. Currently the only *topics* available are command names and built-in debugger variable names.

echo [**-n**] [**-i**] [**-**] *args* ...

The **echo** command writes the *args* to the debugger output device (usually the screen). The *args* are written on one line, each separated by a single space. The **-n** switch will suppress the newline at the end of the output; this can be used to concatenate the output of multiple **echo** or **print** commands. The **-i** switch causes the output to be in inverse video, like error messages from the debugger. The **-** switch (just a dash with no letter after it) is used when the *args* start with a dash: it means "don't try to interpret the next argument as a switch."

Examples:

echo	echo nothing plus a newline to the output device.
echo -i Error	echo the word "Error" in inverse video.
echo -n Error	echo the word "Error" with no newline after it.
echo -i -n Error	echo "Error" in inverse with no newline.
echo - -foo-	echo the word -foo-. Note that "echo -foo-" wouldn't work, because **echo** would try to interpret "-f" as a switch.

print *args ...*

The **print** command is very like the **echo** command, except that it does not simply write its arguments out: it evaluates them, and those that evaluate to a number print as the value, not the text: "echo (+ 2 5)" displays "(+ 2 5)" while "print (+ 2 5)" displays "7."

Those arguments which do not evaluate to a number get printed as they stand, so "print basepage = `clientbp" displays the word "basepage" and the equals sign, and then the value of the expression `clientbp.

The format of the displayed numbers can be set with switches within the **print** command. The switches are as follows:

PRINT COMMAND SWITCHES

SWITCH	MEANING
-x	Set output radix to hex.
-d	Set output radix to decimal.
-o	Set output radix to octal.
-b	Set output radix to binary.
-x4	Set output radix to hex, and use at least 4 spaces. The number 3A would print as " 3A" (works for other radixes, too).
-x08	Set output radix to hex, use 8 spaces, and zero-fill the 8-space field. The number 13A3E would print as 00013A3E. (Works for other radixes, too.)
-s	String; see below.

The '-s' switch causes arguments which evaluate to numbers to display the string at that address in the stub's memory, as ASCII characters, up to the first zero byte. A number, like '-s9', causes the string to be at least that many characters wide, and to be left-justified in the field. A leading zero and a number, like '-s09,' causes the string to be at least that wide, and to be right-justified in the field.

if *predicate command*

The **if** command works as you might expect: if the *predicate* evaluates to TRUE (nonzero), the *command* is executed. If the *predicate* is FALSE (zero), the *command* is not executed.

The *command* part of an **if** command can be several commands, in the same way that an **alias** can be several commands: if the *command* argument is enclosed in quotes (single or double), it may contain several commands separated by semicolons.

```
Examples:

if (= `d0 0) echo d0 is zero.              Simple condition.
if (< `t0 10) goto begin                   Part of a loop in a procedure
if (= (wpeek `sp) 1) \
    "print (lpeek (+ `sp 2));defer g"      See below.
```

The last example above might be an auto-execute alias for a breakpoint: if the word at the top of the stack is 1 when the breakpoint is hit, the longword after that is printed and the client is allowed to start up again. Note the compound command, with the semicolon protected by quotes, and the use of **defer** to start the client the next time the debugger would normally display the prompt.

See the chapter **PROCEDURES, IF, GOTO, DEFER, AND ALIAS** for more information.

indirect *addr*

The **indirect** command causes the client memory starting at *addr* to be read into a local buffer and executed as if it was typed at the command prompt. The command ends with the first zero byte.

EXAMPLE:

```
: s .buf "echo hello\x00" ; indirect .buf
```

This example sets the string "echo hello" (plus a zero byte) into the client memory, then executes the command at that address. Obviously, it prints the word "hello" on the screen.

! [*command-name* [**args** ...]]

The ! (shell) command attempts to execute its argument as a GEMDOS command. The first word of the argument should be the full filename (including the drive and path) of a GEMDOS program file (usually of type .PRG, .APP, .TOS, or .TTP). When the program finishes, you will be returned to the debugger right where you left off, and the debugger will report the exit code of the program.

With no arguments, the shell command attempts to create a shell by executing the file whose name is the value of the environment variable SHELL.

Under some shells, the *command-name* need not be a full-blown pathname. See the section **THE SHELL COMMAND IN DETAIL** in the chapter **OPERATING SYSTEM CONSIDERATIONS** for more information.

dir [*pathname*]

The **dir** command shows a directory listing. With no *pathname* argument, it lists all files in the current directory. With a *pathname* argument, it lists files in the directory specified by *pathname*. *Pathname* may be a wildcard expression, or may consist of a path followed by a wildcard expression (e.g. "***.***" or "**src\db??.c**").

Be careful of ending **dir** commands with a backslash ("\") in scripts: the trailing backslash will be taken as a continuation character, and the next line will be tacked onto the current one. Using, for instance, "**A:*.***" rather than "**A:**" has the same effect and avoids the problem entirely.

Examples:

dir	list all files in current directory
dir A:*.*	list all files in the root of drive A
dir src*.c	list all files in the subdirectory **src** with extension ".**c**" (C program source files)

CHAPTER 5

THE CLIENT, BREAKPOINTS AND CHECKPOINTS: DETAIL

This chapter goes into more detail concerning the client, breakpoints, and checkpoints.

THE CLIENT'S MEMORY

The client's memory is accessed by the debugger in chunks of anywhere from one byte up to one kilobyte. As a rule, when the head wants to examine the client's memory, it asks the stub to copy some into a buffer and send it over. Such copying is done as bytes, to avoid address errors.

However, if the head asks for exactly two or four bytes at an even address, the **move.w** or **move.l** instruction will be used. This means that word-addressed I/O registers will behave as expected.

The following commands show some times when this happens, assuming the *addr*s are even):

Command	Comments
d*waddr*[1]	Dumps *exactly one* word.
d*waddr*[2]	Also dumps one word (the number in brackets is always the number of *bytes* in question, not the number of "things").
sl*addr*	Begins interactively setting longwords. (By reading and writing them them as longs.)
(wpeek *addr*)	Both **wpeek** and **lpeek** act this way.
m*addr*.w != { *addr2* }	The operands of a comparison memory check are read as words or longs, as appropriate.

The **f** (find) command always treats the thing you are looking for as a stream of bytes, so words and longs don't have meaning. The **indirect** command, the special message type F0xx, and the **-s** form of **print** also read client memory in chunks, not as words or longs.

TRACE AND UNTRACE

Trace and untrace are really two modes of the same command. They both single-step through the client. The difference is that trace mode treats instructions which cause traps specially, while untrace mode does not.

Instructions which are treated specially are: TRAP, TRAPV, line-A ($Axxx), and line-F ($Fxxx). However, on a 68020 or 68030, line-F is not treated specially. The TRAPcc instruction isn't, either.

If the PC is at one of the special trap instructions and you use the t command, the result will be that the trap instruction (and therefore the trap handler) will be executed at full speed. When you next see the prompt, the PC will at the instruction after the trap.

If you use the u command in the same situation, only the trap instruction itself will be executed, not the whole handler. When you see the prompt, the PC will be at the first instruction of the trap handler. and the supervisor stack will hold the trap exception frame.

Trace mode treats the trap instructions specially so you don't have to worry about stopping the client in the middle of the operating system, and so the OS will execute at full speed. This way you can set memory checkpoints and then sav **tx** to trace through your program forever, with an opportunity between each instruction, but without slowing down OS calls and without the possibility that you will stop in the middle of the OS itself (which is deadly when not remote debugging). Untrace mode is provided so you can debug a trap handler itself.

The v verbose-trace command without the u modifier is like trace: it executes a trap handler as though it were one instruction.

MESSAGES

A **message** is a special type of communication from the client to the head. Messages don't come from the stub; they come from the client itself, or from another part of the debugger. For instance, when you use the **exec** command, a message is sent telling the head the basepage address of the program that was loaded. If the load fails, or the client later terminates, another message is sent to inform the head (and hence the user) of this, too.

A program being debugged can send messages, too. Messages consist of a 16-bit message number and a 32-bit message argument vector. The negative message numbers are reserved for use by the debugger, but a client may use the positive message numbers freely. A client sends a message to the head as follows (in C):

```
xbios(11,5,msg_number,msg_argv);
```

Msg_number is a 16-bit integer and msg_argv is 32 bits (e.g. a pointer or a long integer).

Remember, negative message numbers are reserved for the debugger's use. When a message is received by the head with a positive message number, the message number and argument vector are displayed, and the client is stopped. See the section **AUTO-EXECUTE ALIASES** in the chapter **PROCEDURES, IF, GOTO, DEFER, AND ALIAS** for more on what happens when messages arrive.

Note that messages provide an *opportunity* as well as a *stop* when they happen during a trace/go.

Message types in the range $F000 to $F0FF are special: they are commands from the client to print something on the user's screen. The message argument vector holds the starting address of the (ASCII) text to display, and the lower byte of the message number holds the length of the text. If the lower byte is zero (that is, message number $F000), the debugger prints the text up to the first null byte. This means that you can print some text on the debugger's output (and cause an *opportunity* and a *stop*) with the following line (in C):

```
xbios(11,5,0xf000,"This is my message");
```

Some C macros such as the following would be useful:

```
#define DBMSG(msgnum,msgargv) xbios(11,5,msgnum,msgargv)
#define DBTEXT(s) DBMSG(0xf000,s)
```

Debugger messages can be used from any language which gives access to the Atari ST's XBIOS. Note that the stub itself masquerades as XBIOS function code 11 (decimal); do not use this call for anything but sending messages.

BREAKPOINTS IN DETAIL

Breakpoints work internally as follows: When a trace/go is started, the instruction at each breakpoint address is saved, and the illegal instruction is placed at those addresses. Then the client is started. If the processor comes across an illegal instruction, it generates an exception, which the stub catches. It checks to see if the address of the illegal instruction matches any of the breakpoints that were set. If so, the count value of the breakpoint is decremented (but not through zero). If the result is zero, the trace/go stops and all the instructions with breakpoints are restored to their original values. Otherwise, the trace/go continues, starting with the instruction which was "under" the breakpoint (i.e. the one replaced by the illegal instruction).

MEMORY CHECKPOINTS IN DETAIL

Checkpoints have two phases: the initialization phase and the evaluation phase. The initialization phase occurs when the head tells the stub to begin a trace or go. The evaluation occurs during opportunities such as between instructions of a trace and while processing a breakpoint.

Comparison checkpoints

If the **old** keyword was used in setting the checkpoint, the value at the *address* is read into the operand field as the first part of the initialization. Then, all comparison checkpoints are evaluated once, and their current state (true or false) is saved.

At each opportunity, the comparison checkpoints are evaluated: the state (true or false) is computed again. If it's the same as the old

state, there's no stop. If the old state was TRUE and the new state is FALSE, the new state is saved, but there's still no stop. If the old state was FALSE and the new state is TRUE (i.e. the comparison has *become* true), the checkpoint causes a stop.

Range checkpoints

Range checkpoints are initialized by computing the CRC value for the region in question. That value (16 bits) is stored in the checkpoint slot. When an opportunity arises, the CRC is computed again. If it doesn't match the initial value, the checkpoint causes a stop.

Note that the CRC is not an infallible method for detecting changes. Some changes can cause the region to compute the same CRC value as before.

MEMORY CHECKPOINTS ON VALUES IN REGISTERS

With the ampersand prefix (e.g. &d1) you can get the address where the stub stores the values of CPU registers during checkpoint evaluation. What you have to realize is that the address you get is the address of the high-order byte of the value. For memory checks on d1.l, then, "**&d1.l**" is the correct address specification for the **m** command. If you want to perform your memory check on d1.w, "**(+ &d1 2).w**" is the address expression you want. For d1.b, "**(+ &d1 3).b**" is what you would use.

To compare two registers to each other, you would use the indirect comparison checkpoint type. Say you want to stop when a1.l is greater than a2.l: the command "**m &a1.l > {&a2}**" accomplishes this. Of course, to compare words, you have to shift the addresses by two: "**m (+ &d1 2).w > {(+ &d2 2)}**" stops when d1.w > d2.w.

It is also important to remember that not all CPU registers are longs: the SR is stored as a word, so "**&sr.w**" is the address for the whole SR, and "**(+ &sr 1).b**" is the address for the CCR part of the SR. See the section **Stub Variables** in the chapter **SYMBOLS AND DEBUGGER VARIABLES** for a complete list of stub variables.

CHAPTER 6

SYMBOLS AND DEBUGGER VARIABLES

Db can load symbols from programs and other sources. In addition, the **sym** command can be used to create entries in the symbol table to assist debugging. Debugger variables are values the debugger makes available to the user by name, such as the basepage of the program last loaded, and the type and argument vector of the last message, along with eight temporary storage locations for use at the user's whim. Also, a user can declare new global variables by name, and even local variables within procedures.

SYMBOLS

Symbols are loaded from programs being debugged using the **exec** and **getsym** commands. These commands add the symbols from the files they load to the debugger's internal symbol table. The value of a symbol in the table can be used in an expression by prefixing it with a dot: '**.symx**' yields the number in the value field of the symbol 'symx'.

Symbols which refer to addresses in the text, data, or BSS segments of a program are *relocatable* symbols. In the program file, they have values as though the program were running from absolute address zero. Of course, programs don't run there, so the program loader (and the debugger) must *relocate* the values of these symbols to reflect the address at which the program is actually loaded. *Db* takes care of this automatically.

If you specify "**.main**" and there is no symbol **main** in the symbol table, but there is a **_main**, the debugger provides the leading underscore for you. Specifically, the following variations are tried: prepend underscore; append underscore; truncate at 8 chars; prepend underscore *and* truncate at 8 chars.

CONSTRAINED SYMBOLS

A program file may have been produced by linking several modules together. These modules each had some global symbols and some local symbols. If you ask it to, your linker will include either both kinds of symbols, just the global symbols, or no symbols in the program file. Global symbol names are usually unique in a program file, but local symbol names might not be: there might be a local symbol called "start" in both "filea" and "fileb," for instance.

If you have *aln* or another linker following the same conventions, you can specify the file name before the symbol name to differentiate these two: '**.filea:start**' is different from '**.fileb:start**'. If fileb came from the library (archive) mylib, the full specification is '**.mylib:fileb:start**'. Furthermore, there is something called a "confined" symbol: a symbol whose scope extends to the two unconfined

symbols surrounding it. These symbols begin with '.', '~', and 'L'.

(Symbols beginning with 'L' are generated by some compilers (notably Alcyon C) as internal labels. Strictly speaking, they are not confined: they are unique within each source file. However, they are considered confined so when their full specification is printed by the debugger, you can see what procedure they occur within.)

In general, symbols are uniquely identified by the names of all the levels enclosing them: the levels of enclosure are archives, files, unconstrained symbols, and constrained symbols.

Take the following code fragment, for example:

```
; file init.s in archive mylib

clrmem:
        move.w  #COUNT-1,d0
        move.l  #START,a0
.loop:  clr.b   (a0)+
        dbra    d0,.loop
```

The full specification of the symbol **.loop** is:

.mylib:init:clrmem:.loop

Be careful to distinguish between the period which introduces a symbol specification and the period which is the first character of a symbol's name. If this .loop is the only one in the symbol table, it could be specified simply as "..**loop**".

Still another way to differentiate symbols, useful for files linked without symbols of type 'file', is the number-sign ('#'). "**.symx#4**" refers the fourth occurence of symx in the symbol table.

DEBUGGER VARIABLES

Debugger variables carry information which you can read, change, and use in expressions. You can see the names of all the built-in variables with the **vars** command, you can see or set the value of a variable with the **set** (or **x**) command, and you can use the value in an expression with the backquote ('`') prefix. Finally, you can get the address of the *stub variables* with the ampersand ('&') prefix. The stub variables are special because their true values come from the stub. A copy of these variables is kept in the head, and when you trace or go, they are written to the stub. When the trace/go finishes, their (possibly changed) values are read back from the stub.

(In fact, the true values of all of the stub variables is read from the stub when first you read or set any of them. If you change a variable, the new values are all written to the stub the next time you trace or go. This saves time when you don't read or set them.)

All debugger variables are stored as a longword in the head, and most are stored as a longword on the stub. The ones stored as words on the stub have "(word)" after them in the following table. To use these in a comparison-type memory checkpoint, you would use, for example, "&sr.w" to refer to the status register. Two variables, **sfc** and **dfc**, are stored as bytes.

Stub Variables

The Stub Variables contain information about the stub.

NAME	DESCRIPTION
cputype	The type of CPU the stub is on (68xxx, word).
version	The version number of the stub (word).
nbreaks	The number of breakpoint slots (word).
nmems	The number of memory checkpoint slots (word).
stubcode	Pointer to the start of the stub.
breakptr	Pointer to the breakpoint array.
memptr	Pointer to the memory checkpoint array.
stubbp	Basepage address of the stub process (for symbols).
clientbp	Basepage address of the last-exec'ed client.
exspace	See below.

The **exspace** variable contains the address of stub memory where exception stack frame information is placed. The whole exception stack frame is copied from the stack to this space: see the processor documentation for the sizes and meanings of the stack frames.

Client Registers

The Client Register variables are the ones which mirror the actual CPU registers of the client.

NAME(S)	DESCRIPTION
sr	The status register (word).
d0 - d7	The data registers.
a0 - a6	The address registers.
ssp	The supervisor stack pointer.
usp	The user stack pointer.
pc	The program counter.
sfc dfc	680x0 registers (byte).
msp vbr cacr caar isp	680x0 registers.
a7 sp	Translated to usp or ssp based on sr.

Other Built-in Variables

All other variables are not stored in the stub: they are just in the debugger.

NAME(S)	DESCRIPTION
t0-t7	Eight temporary variables you can use any way at all.
$	Holds the value of the last math command or the first match address from the last **f** (find) command.
mtype	Holds the **type** of the last user message received.
margv	Holds the **argv** of the last user message received.
rwstart	Holds the start address of the last file read or written.
rwsize	Holds the size of the last file read or written.
iodev	Holds the current I/O device number (see below).
bdev	Holds the current BIOS I/O device number (see below).
discpu	Holds the CPU type for disassembly (see below).

The **discpu** variable holds the last two digits of the CPU type, *in decimal*: 00, @10, @20, or @30 for 68000, 68010, 68020, or 68030. Instructions which are legal on a 68030 but not on a 68000 through 68020 will not be disassembled if **discpu** is not @30.

The **iodev** variable holds a number which tells the debugger what I/O device to use:

VALUE	MEANING
0	GEMDOS (screen / keyboard)
1	Serial port (polled)
2	BIOS (see below)
3	MIDI (polled)

Values not listed above cause an error.

When the value of **iodev** is 2, BIOS calls are used for input and output. The BIOS calls take a device-number argument, and that device number is taken from the variable **bdev**. No check is made to see if you have set a sensible value here.

Normally, the debugger starts up using GEMDOS (**iodev** value 0). Using the **-s**, **-b**, and **-m** options on the debugger command line causes it to start up using another value (1, 2, and 3, respectively).

USER-DEFINED VARIABLES

The **global** and **local** commands create new variables by name. **local** is generally used only in procedures: it creates variables which is

visible only while executing in that procedure. **global** creates variables visible from anywhere. In each case, you use the variables the same way you use any others: you put backquotes before their names.

CHAPTER 7

PROCEDURES, IF, GOTO, DEFER, AND ALIAS

WHAT IS A PROCEDURE

A procedure is a list of debugger commands which is stored in memory and executed by name. A procedure consists of the following parts:

1. The procedure name.
2. The list of arguments.
3. The list of commands making up the procedure.

Once you've created a procedure, you call it by using its name as a command, followed by as many expressions as the procedure has arguments. The commands in the procedure body are executed as if they came from the keyboard or a script file.

Procedures can call other procedures, nesting to any depth (limited by the amount of memory the debugger started with). They can contain any debugger command except **procedure** itself.

Procedures can use the **local** command to create variables which exist only during the execution of the procedure, and are visible only within the body of the procedure.

One local variable, **argc** ("argument count"), is created for every procedure. It tells how many arguments were provided for the procedure. You can call a procedure and give it fewer arguments than it calls for. However, if you provide too many arguments, you will get an error message. You can create a procedure that can be called with fewer than the maximum number of arguments and still do something useful.

SAMPLE PROCEDURE

Here is a sample procedure:

```
procedure sample maxval
  # This procedure shows the first `maxval nonnegative integers.
  local count ; set count 0

  if (< `argc 1) abort Too few args to procedure sample

  #:loop
    print -n -d `count
    set count (+ `count 1)
    if (< `count `maxval) goto loop
  print
.
```

The first line is the *procedure declaration*: it starts with the **procedure** command, then the name of the procedure ("sample"), then the argument list. This procedure takes one argument, "maxval."

The next line is a comment, telling what the procedure does. The third line is the **local** command: it creates a local variable, visible only inside this procedure, called "count." Local variables start out with no particular value, so it's immediately initialized to zero by the **set** command.

The next line is blank. You can have blank lines in procedures. When the procedure is stored, they get translated into lines which start with "#," meaning the whole line is a comment.

Next, we have an **if** command. This checks the variable **argc** to see if the procedure was in fact given an argument. (You can't provide *more* arguments than the procedure calls for, but you might provide fewer.)

The next line (after the second blank one) is a label. You can tell it's a label because it starts with the two characters #: (hash colon). When the flow of control in the procedure gets to this line, it will be treated as a comment (since it starts with #). When storing the procedure, however, the debugger sees this as a label, and saves this position in the procedure under the name after the colon (in this case, "loop").

The **print** and **set** commands do what you'd expect, as does the **if**. The **goto** after the **if** takes as its argument the name of a label in the procedure. The label can be anywhere in the procedure. Labels, remember, begin with the characters "#:."

After the **if** is another **print** command: this terminates the line which all those **print** commands with the **-n** switch were writing to. This **print** command is outside the loop, and so is not indented as far. The indentation is totally up to the programmer, and is used to make the control structures of the procedure clearer.

The dot on the last line is just that: a dot, a period. That marks the end of the procedure. It's not a command: it's recognized in the procedure-creation phase as the end marker.

Running this procedure looks like this (the colon is the debugger prompt):

```
: sample 9
0 1 2 3 4 5 6 7 8
:
```

MORE DETAILS ON PROCEDURES

Procedures need a little more explaining. They have some restrictions and unexpected side-effects.

In the first place, **the goto command must be the last command on a line.** The implementation of the **goto** command is a little strange, and the upshot is that it takes effect at the end of the line it's found on. Other commands after a **goto** will execute before the **goto** itself does. You are not encouraged to take advantage of this, and it might change in the future. Just live under this restriction: make sure no command ever comes after a **goto** command on a line.

Second, remember that local variables are searched before global variables, but built-in variables are searched first of all. A global or local with a name like "pc" will never be seen; the debugger variable "pc" will be used instead. A local with the same name as a global, however, *will* be seen:

```
global myvar
set myvar 3
procedure foo
  local myvar
  set myvar 10
  print myvar
.
foo
print myvar
```

The above sequence will print "10" followed by "3" because the local myvar is seen inside the procedure, while the global myvar is seen outside it.

PROCEDURE-RELATED COMMANDS

The **procedure** command with no arguments lists the *procedure declaration* for all procedures. This includes the name and the argument list. This can serve as a reminder of what a procedure does and how to use it. if the procedure's name and its arguments' names are well chosen.

The **procedure** command with one or more arguments begins the creation of a procedure. The first argument is the name of the procedure to create. and the subsequent arguments are the names of the procedure's arguments.

When typing a procedure in from the command prompt (as opposed to loading it from a file), the debugger prompts you with a double-colon ("::")

prompt for each line. The lines you type are not interpreted at all, only stored. The end of the procedure is marked by a line consisting of a period only. At that point, the debugger scans the procedure for labels and stores the procedure name, its argument names, and the label positions in the procedure list. Only at this point is any old procedure by this name removed from the list: if you use ^C to abort the creation of the procedure, a pre-existing procedure with that name will not have been removed.

The **plist** command with no argument lists all procedures in a form suitable for saving (with **transcript**) and restoring (with **load**). They begin with the **procedure** command and end with a period alone on a line. With one or more arguments, the **plist** command lists only those procedures.

DEFER AND ALIAS

This section describes the **defer** and **alias** commands, and offers some advanced advice on using the debugger.

It is unfortunate but true that you may have to read this whole section before you can really understand and use any of it. The explanations of **alias**, **defer**, and compound commands are of necessity given in terms of each other. Please be patient and read through this a couple of times.

ALIAS

The **alias** command takes two arguments: a name, and an expansion for that name. After this, any time the name appears as a command, it is replaced (textually) with the expansion.

(In the examples in this chapter, a line beginning with a colon (':') shows a command which you can type in to the debugger. The colon itself should not be typed: it represents the debugger's prompt.)

```
: alias foo dw.table[10]
```

After using the above command to define an alias for "foo" the "command" **foo** can be used, and it will expand to "dw.table[10]" (which dumps the 16 bytes starting at the label "table" as words). This is a very simple example of the alias command, but still quite a timesaver for commands you use a lot.

Aliases are expanded in place in the command line, and any arguments to the alias appear at the end of the expansion. The following (extremely useful) alias illustrates this:

```
: alias rwfind f`rwstart[`rwsize]
```

Now, the new "command" **rwfind** can be used to find values or text in the file which has just been read with the **read** command: "rwfind 'some text'" expands to the command "f`rwstart[`rwsize] 'some text'" which will find

all occurrences of the quoted text in the file.

AUTO-EXECUTE ALIASES

Auto-execute aliases have special names: they start with the letters "**br**" or "**mc**" or "**msg**" and they are executed when a corresponding breakpoint, memory checkpoint, or message event happens, respectively. For example, when the breakpoint in slot zero causes a stop, the debugger looks for an alias called "**br0**" and executes it if it exists.

Breakpoint aliases start with **br** and end with the slot number they are attached to (as one upper-case hex digit): **br0** through **brF** if there are 16 breakpoint slots. Memory checkpoint aliases start with **mc** and end with the memory checkpoint slot number, also as one upper-case hex digit. Message checkpoints start with **msg** and end with the message number they handle, as *four* upper-case hex digits: **msg0000** for message type zero, **msg0FCA** for message type $0fca.

When several events happen at the same time, such as multiple checkpoints or a checkpoint and a breakpoint, only one auto-execute alias is executed. Breakpoint aliases are checked for first (in ascending numerical order), then checkpoints (also in order), and finally messages. The first one of these which exists, and *only* that one, is executed.

If none of these exists, the default action is taken: the breakpoint, checkpoint, and message type and vector are displayed on the screen.

Note that the auto-execute alias for an event can itself cause a trace/go. If it does, and that trace/go is stopped by an event, the auto-execute aliases are checked again and the first matching one is executed, so the right combination of events and auto-execute aliases can cause a lot to happen automatically. See the examples below for more.

When you set an auto-execute alias, be careful to remember that it is there. For instance, if you set a breakpoint someplace, and create an auto-execute alias for that breakpoint, and then you remove the breakpoint, the auto-execute alias is still there. If you set another breakpoint and it happens to go in the same slot as the first one, the auto-execute alias will be triggered by it, probably resulting in something you didn't expect or want. **Unalias** is the command which removes one or more aliases from the debugger's alias table.

COMPOUND COMMANDS, introduced

If you enclose the expansion argument to **alias** (or **defer** or **if**) in quotation marks, it can contain more than one command:

```
: alias foo "echo xtable;dw.xtable[10];echo ytable;dw.ytable[10]"
```

Now, when you use **foo**, four commands (two echoes and two dumps) will be executed. Again, this can be a great timesaver. As explained below, it can be the key to really powerful macros.

DEFER

The **defer** command takes one argument: a command to be executed the next time the debugger returns to the top level for user input. That is, when the debugger is about to print its prompt, the last thing it does is execute any deferred command. The purpose of this is to allow for automatic execution of the client and looping in macros, without using the alias stack.

Only the last **defer** command is remembered. **Defer** with no arguments causes the debugger to forget any existing deferred command.

Here is an example of the use of the **defer** command:

```
: b.endloop
: m #0 &d7.l != old
: alias mc0 "print d7 changed: new value `d7;defer tx"
: tx
```

If the client is about to start a loop, and the user wishes to be notified when d7 changes, the above sequence will do the trick. It will stop with the breakpoint at the end of the loop, and each time d7 changes the auto-execute alias mc0 will be executed. This alias displays the new value of d7, then tells the client to continue executing rather than returning to the command level.

The above example would still work if the last command in the alias were simply **tx** rather than **defer tx**, but it would soon fill up available memory with the stacking of alias expansions: using one alias in another amounts to a procedure call.

Defer can also be used as a trick to allow arguments to an alias. Remember that an alias expands from a command (like mwc or rwfind) into the expansion text, in place. Any arguments to the alias are tacked on after the expansion:

```
: alias foo "echo one two"
```

would cause "foo x y z" to expand to "echo one two x y z." A macro to print the Nth longword in a table starting at .table might be as follows:

```
: alias nthlong "defer print (lpeek (+ .table (* `t0 4)));xt0"
: nthlong 3
```

This works because the command nthlong is substituted with the text of the alias, and the '3' is tacked to the end of that. Because of the **defer**, the command "xt0 3" will be executed before the **print** command, so t0 will have the value 3 by then, and the value at (.table + (3 * 4)) gets

printed.

When the "argument" you're trying to provide is a number, it's far better to use a procedure:

```
procedure nthlong n
  print (lpeek (+ .table (* `n 4)))
.
```

This use of **defer** is really just a leftover from when the debugger didn't have procedures. It's still useful for string arguments, though. (Or it will be until the debugger gets strings as a data type...)

COMPOUND COMMANDS, explained

As you can see from the examples above, the **if**, **defer**, and **alias** commands each take a command as an argument. That argument can be a *compound command* consisting of more than one simple command if it is enclosed in quotation marks:

```
alias mycmd "echo start mycmd;l;print end of mycmd"
```

Executing the above command, then the command "mycmd," will cause the legend "start mycmd" to appear, then a disassembly listing of 12 lines starting at the current disassembly pointer, then the legend "end of mycmd." (Sure, it's silly, but it's just an example.)

The important point is that the semicolons are enclosed in quotes, making them part of the argument to "alias" rather than being interpreted as separating the alias command from the l and the **print** command. Without quotes,

```
alias mycmd echo start mycmd;l;print end of mycmd
```

the alias for mycmd would be "echo start mycmd" -- the **echo** command stops with the first semicolon, and the l and **print** commands are executed in turn.

The alias for Mark Williams C argument string handling uses this trick: the alias itself consists of two commands, a find (**f**) and a set (**s**):

```
: alias mwc 'f (lpeek (+ `clientbp 2c))[800] "ARGV=" ; s $ 5a'
```

Note the use of single quotes around the alias, and double quotes around the string argument to the find command. Single quotes match single quotes and double matches double, but their interpretations are identical.

You can nest these expansions:

```
alias setbp \
    "b #2 .mainloop; alias br2 'echo loop;dw.table[10];defer g'"
```

Once you create this alias, if you use the command **setbp** a breakpoint will be set, *and an alias will be created* which will get executed when that breakpoint is hit (see the section **AUTO-EXECUTE ALIASES** in this chapter). The alias which **setbp** creates, called **br2** (to attach it to breakpoint slot 2), contains a compound command as its expansion: the compound command prints a message, dumps the first eight words of a table (16 bytes), and then lets the program continue executing.

(Unfortunately, you can't nest more than two levels of compound commands, because only the single- and double-quote characters protect semicolons, and any more of them would look like closing, not opening, quotes.)

Another use for auto-execute aliases might be to show something on the screen at the start and end of a certain procedure:

```
: b #3 .myproc
: alias br3 "echo entering myproc ; defer g"
: f1 .myproc[800] 4e5e4e75
: b #4 $
: alias br4 "print myproc returns `d0 ; defer g"
```

Note the **f** (find) command in this sequence: it searches from the start of the procedure, for 2K bytes, for the longword $4e5e4e75. That is two 68000 opcodes: UNLK and RTS. Every procedure compiled with Alcyon C ends with these two instructions, and the likelihood of finding that exact byte pattern anywhere in the procedure except the end is very small, so the chances are that breakpoint #4 will be set at the UNLK instruction. (If the procedure is more than 2K bytes long, the **find** should have a longer count.)

The **f** (find) command will dump the locations of all matches on the screen, even though all we are interested in is getting $ set to the address of the first one. You can suppress this needless output by surrounding the **f** command with **gag on** and **gag off**. See the section **GAG** in the chapter **COMMANDS** for more information.

CHAPTER 8

OPERATING SYSTEM CONSIDERATIONS

Db must operate within the constraints imposed by the Atari ST operating system. When these constraints prevent using db in the manner needed, the user should consider *remote debugging*. See the chapter **REMOTE DEBUGGING** for more information.

DB AND GEMDOS

When you don't specify a command-line *option* like **-s** or **-m** for input and output, the debugger uses GEMDOS to access the screen and keyboard. It is important to know, then, that two programs can't be using GEMDOS at the same time. If you stop the client while it is executing a GEMDOS system call (like Fopen or Cconout), and the debugger uses GEMDOS to print to the screen, GEMDOS will lose track of the client, and the next **g** command will create havoc.

If you use the **t**, **v**, and **g** (trace, verbose-trace, and go) commands exclusively, and avoid **u** and **vu**, there should be no problem, because they will never stop while the PC is in GEMDOS. However, if you use the **u** (untrace) or **vu** (verbose-untrace) commands, you could stop while in GEMDOS, and that would be bad news.

Furthermore, if the debugger is using GEMDOS for input and output, and you hit the STOP button while the PC is in GEMDOS, you are in the same boat. So the lesson is to use **t**, **v**, and **g** exclusively when using GEMDOS for input and output, and don't use the stop button unless you are sure the PC is not in GEMDOS or the BIOS.

Even when it's not using GEMDOS for its input and output, the debugger uses GEMDOS for certain commands, like **exec** (to load a file and set it up for execution) and **getsym** (to load symbols from a file). Thus, you should be sure that the client is not in the middle of GEMDOS when using these commands. Another command which can cause even more trouble is **transcript**, because the user has little control over when the buffer will be written to disk. When debugging locally, use **transcript** with extreme care, making sure that you don't stop while the PC is in GEMDOS.

When remote debugging, none of this applies, because the slave and the master have two independent GEMDOSes.

DB AND MARK WILLIAMS C

Mark Williams C† uses a different symbol table format from Alcyon's. Notably, symbols are stored in 16 characters, not just 8. Also, global variables in C get an underscore character *appended* to them, rather than *prepended* as is the convention among most C compilers. (The reason for doing this is so you don't have to worry about name collisions with assembly language: by not using a leading underscore (or trailing, in the case of MWC), you know you won't be using the same name as a C variable.) **Db** correctly interprets Mark Williams C symbol tables, both in the old (before version 3.0) and version 3.x formats.

Mark Williams C and some other environments use a trick to pass more than 127 characters' worth of command-line arguments to their programs. The trick is to use the environment variable ARGV, because the value of an environment variable can be any length at all. There is a problem with this approach, however: since the environment is inherited from one process to another, the child can't tell if the ARGV in its environment really came from its parent. MWC programs will take the *debugger*'s arguments as their own.

The way to fix this is to force the MWC program to think that there are *no* arguments in its environment. There is an automatic way to do this: place this **alias** command in your db.rc file and use it after you **exec** an MWC program, but before the first trace/go command:

```
alias mwc 'f (lpeek (+ `clientbp 2c))[800] "ARGV=" ; s $ 5a'
```

This alias searches in the client's environment (the address of which is at `clientbp+$2c) for the word "ARGV=" and changes the first letter of that word to a 'Z'. This prevents the MWC argument-parsing code from finding "ARGV=" in its environment (because it now reads "ZRGV=") and the program will therefore look in the basepage for command line arguments.

If you don't understand this whole discussion, or why the alias above works, that's okay: just place the alias in your autoload file (usually "db.rc"), and type the command "mwc" after you **exec** a client compiled with MWC. Then use the **args** command to pass arguments to the client.

DB AND THE XBIOS TRAP

Db uses XBIOS function code 11 (that is, trap #$e when the word on the top of the stack is $000b). The program you are debugging may install a handler for trap 14. However, if the program is a resident utility (sometimes called "TSR" for "terminate and stay resident") you have to be careful when debugging it. Specifically, the debugger replaces the old vector for trap 14 when it exits. Since your program linked into the trap after the debugger did, the debugger can't know how to remove itself from the linkage, so it simply clobbers the trap 14 vector, removing your handler from the trap.

---

†Mark Williams C is a trademark of Mark Williams Company.

You can still debug TSRs which use trap 14, however. You can either run the TSR before running the debugger, or run the debugger, and then **exec** your TSR, let it run until it terminates (and stays resident), and then **exec** a program to test it, all without leaving the debugger. If you run the TSR before running the debugger, you should arrange for the TSR to let you know its text base address, so you will be able to use **getsym** to load its symbols for debugging.

Naturally, since the debugger itself uses trap 14 function code 11, no user program should use that same function code.

THE SHELL COMMAND IN DETAIL

The ! (shell) command can be used to leave the debugger temporarily, execute a command, and re-enter the debugger where you left off. What it does is execute its argument as a command, with the GEMDOS Pexec function. This requires that there be enough memory available to the operating system to run that program. This is often not the case; if you try it and get "insufficient memory" then that is the problem.

Some shells use the system variable _shell_p in a special way. **Db** tries to detect these shells. The presence of such a handler lets you pass _shell_p a command line like "grep foo *.c" and let the shell figure out where to find grep, how to load it, and how to pass "foo" and "*.c" (or "all the files which end in .c") as its arguments.

You tell the debugger that you have such a shell by setting the environment variable SHELL_P to the value "yes" before starting **db**. In most shells, the command to do this is

```
    'setenv SHELL_P yes'
or 'setenv SHELL_P=yes'
```

With no arguments, the ! command looks in your environment for the variable SHELL. If it's found, the value is assumed to be the full filename, including the path and file type, of your shell, and that file is executed.

When the program you execute (or the shell from $SHELL) exits, you reenter the debugger exactly where you left off, with all the same state you had before you left.

Note that this command will only work when it is okay to make GEMDOS calls. See the section **DB AND GEMDOS** in this chapter for more information.

EXCEPTIONS

The trace/go commands can all cause the program being debugged to execute instructions which cause exceptions in the 68000 processor. Most of these exceptions are caught by the debugger. In particular, bus error, address

error, etc. (exception numbers 2 through 9) are caught, as well as the spurious and uninitialized interrupt vectors. In addition, the debugger has a provision for a "stop button:" hitting the stop button will cause the client to stop. See the section **STOP BUTTONS** in the chapter **REMOTE DEBUGGING** for more information.

The debugger contains a list of those exception vectors which it takes over. The debugger restores all the vectors it takes over on exit. If your program or some program in your system uses a vector which the debugger considers an error, like one of the reserved vectors, or "spurious interrupt," or "format error," then you are just out of luck; you will have to use the debugger carefully or not at all.

When a trace/go command causes one of these exceptions to occur, execution is immediately stopped and control is returned to the debugger. The **pc** and **sr** are saved from the exception stack frame, and all other registers keep their values. Note that after bus error and address error on a 68000, the **pc** will not have a reliable value: the instruction causing the exception is *near* the **pc**, probably somewhere from two to ten bytes before it.

When a trace/go command stops because of an exception, the **where** command is convenient to determine what procedure was executing at the time: it reports name of the symbol closest to, but not after, the current **pc**.

DB, TOS, AND 68030

The debugger and TOS both run on 68030's (the Atari TT), but some shoehorning was required. One such shoehorn was a privilege violation handler. On the 68000, the instruction "**move sr,d0**" is not protected. On the 68010 and up, it is. Some ST programs use this instruction, especially to save the condition code register (CCR), which is part of the SR.

To make those programs work on the 68030, Atari placed a privilege violation handler in the OS. If a "move from sr" instruction caused the violation, the handler writes a new instruction in that place: "**move ccr,d0**" (of course, this works for any destination, not just d0).

Since the debugger catches exceptions (because they usually mean bugs in your program), the debugger has to do the same thing. If you have a "move from sr" instruction in your program and you run it on a 68010, 68020, or 68030, the debugger might demote it into a "move from ccr" instruction. If this causes your program to fail, now you know why.

DEBUGGER MEMORY USAGE

The debugger must share memory with the rest of the operating system and with the client being debugged. Under TOS, all programs are allocated the largest single block of free memory, and if they plan to start up other processes they must give some of that memory back to TOS.

The debugger program has a variable which controls how much memory it gives back to TOS. That variable can be found from the outside because it is the first longword of the data segment of the debugger program file. (This also applies to **rdb**, the remote debugger.)

In addition to the client, this "outside" memory is used by the **read** command when no specific address was provided. The debugger's internal memory is used for such things as storing procedures and aliases, user variables, and stack frames when executing procedures and expanding aliases. Finally, the ! (shell) command uses this "outside" memory.

If you find that the mix of debugger memory and client memory does not suit you, either because the debugger takes too much (the client can't load or reports that it's "out of memory" somehow), or because the debugger takes too little (the debugger reports "out of memory" when you load symbols or execute procedures), you can change this variable.

The variable controls the debugger's memory usage by controlling how much of the initial block the debugger keeps, and how much it returns to TOS:

VALUE	MEANING
-1	Keep the whole block. Not very useful for a debugger.
0	Keep only a bare minimum. Not likely to last long.
1	Keep 1/4 of the block, free 3/4 for clients.
2	Keep 1/2.
3	Keep 3/4, free only 1/4 for clients.
+other	Other positive numbers keep that many bytes exactly.
-other	Other negative numbers return that many bytes exactly.

The first two values (-1 and 0) are not likely to be useful. If the debugger keeps all of memory, there isn't any left for the client. If the debugger keeps hardly any memory, it might not have enough to keep track of its internal data structures.

For a local debugger (not remote debugging), a value of 1 is usually right. This leaves lots of room for the client, but the debugger keeps enough for symbols, procedures and the like. If you have a great many symbols and a small program, you might need to bump this up to 2.

For a remote debugger, 2 or 3 are usually good enough. A remote debugger uses the memory it keeps the same way as a local one, but the external memory is used only for the ! (shell) command. If you have a great many symbols, -1 might even be necessary, but in that case you will not be able to use the shell command.

You configure the debugger by actually changing the program file on disk. Once the debugger has started, it's too late for that debugger. Here is an example debugger session where the user creates a new debugger program file (called "DB3.TOS") which has the value 3 in this control variable:

```
A   : read db.tos
B   Done. Start=17D240, size=27DC2
C   : sl (+ `rwstart 1C (lpeek (+ `rwstart 2)))
D   196340: 00000001 3
E   196344: xxxxxxxx .
F   : write db3.tos
G   Done. Start=17D240, size=27DC2
H   : exit
```

On line A, the user reads the executable file in. The debugger reports the result on line B. Line C is an **s** (memory set) command: look at the complex expression carefully, and you'll see that the address is ultimately the first longword of the data segment. (Or just type it in as shown: it'll work even if you don't understand it.)

Line D shows the old value of the variable, 1, and the user's new value, 3. Line E shows the next address, and the user stops the memory set command by entering a period alone on a line. On line F the user writes the new file out, and on line H she quits.

CHAPTER 9

REMOTE DEBUGGING

You can use *db* as a remote debugger. This means that you can have the main body of the debugger (the *head*) on one machine (the *master*), and a little bit of the debugger (the *stub*) plus the program you are debugging (the *client*) on another machine (the *slave*).

The advantages are that the debugger doesn't use up the slave's memory and other resources (screen, keyboard, disk), and the program being tested doesn't put the debugger machine (presumably the one with all your files on the hard disk) at risk. Also, there are no restrictions in terms of GEMDOS use between the client and the debugger, since there are two machines and two GEMDOSes. Finally, you can use the debugger to debug an operating system: on one machine, you would need a working GEMDOS to load the debugger, but when remote debugging you can actually debug the OS as it boots, and you can set breakpoints in interrupt handlers.

When remote debugging, the *master* (the machine with the bulk of the debugger) communicates with the *slave* (the machine with the stub and client) through a bidirectional MIDI cable. This actually a pair of MIDI cables, with one cable connecting the MIDI OUT port of the master with the MIDI IN port of the slave, and the other cable connecting the MIDI OUT port of the slave to the MIDI IN port of the master. Using the MIDI port provides faster communication than the serial port, and leaves the serial port available so you can debug terminal emulators and other applications which use it.

To use remote debugging, you have to load the stub into the slave machine. There are two ways to do this: you can start a program containing the stub which initializes itself and then loads your client program, or you can arrange for the stub to be resident in the machine and then load the client the way you do any other program.

In both cases, you run the remote debugger head, **rdb**, on the master machine.

The first method involves using the program "STUB.TTP" on the slave. This program takes the name of the client (and any arguments to it) as command-line arguments, loads the stub, then loads the client. When the client is loaded and ready, the stub sends a message to the head. Then, you debug the client as usual. When the client terminates, the stub sends another message to the head. If you use the **exit** command on the head, the stub will be told to exit as well. It terminates the client, unloads the stub, and both machines will return to the desktop or shell.

The second method requires that you establish a resident stub. This can be done by running a "terminate and stay resident" program (called "STUBRES.PRG") on the slave machine.

When remote debugging using the resident stub, you will not get messages when

programs start up. In all other respects, the stub is active (i.e. it still informs the head of bus errors, etc.). You have to stop the slave (with the stop button) and explicitly enable client-startup reporting with the command **exec on**.

When remote debugging, the normal cycle is like this: The user starts Brdb on the master machine, then starts the client on the slave machine, either with STUB.TTP or after executing STUBRES.PRG. The head simply waits for the first activity from the stub.

Eventually, the stub sends a message to the head (e.g. CLIENT, STOP BUTTON, BUS ERROR) and waits for the head to send it instructions. In response to commands from the user, the head sends instructions to the stub (e.g. a user command "dump" means the head has to ask for the contents of the client's memory from the stub). When the head sends a command to the stub, it waits for the reply before doing anything else. Usually, the replies come quickly: a one-instruction trace, for instance, takes only a fraction of a millisecond to execute. When the reply comes, the head can continue its business.

On the other hand, the reply may be a long time away, or may never come: consider a **g** (go) command which leads the client into an infinite loop. The reply will never come, and the head would be waiting forever to find out what the result of the "go" was.

For this reason, the debugger does not wait forever for trace/go commands to finish. After about 10 seconds, the message "Waiting... Press ^C to stop waiting." appears. The head goes on waiting for the stub to respond, but if you hit ^C (control-C), the head will stop waiting and return you to the prompt. The client is still running, and the effect is like a **continue** command.

The **continue** command causes the head to tell the stub to run the client like a "go" command, but *it doesn't wait for a reply*.

When the slave is busy running the client, either because of a **continue** command or because a **go** command didn't reply and was stopped with ^C, the debugger returns you to the command prompt. You can continue issuing debugger commands. Naturally, since the slave is busy running the client, you can't issue any commands which need to access the stub. This leaves only a couple of useful commands: the symbol-table commands **getsym**, **where**, and **?**, and the expression-type commands (where you type an expression and see the answer).

One command which is especially useful is **quit**, which, when remote debugging, doesn't touch the slave at all, but returns the master to the desktop. If you want to let the client run and then leave the debugger, just type **continue** and then **quit**. The client will continue to run. You can even re-enter the debugger, and it will reestablish communications with the stub when the client stops.

[Sometimes, the head and stub cannot reestablish communications, because they are out of synchronization. When this happens, you have to reset the client, hit ^C on the debugger and say **quit**, and start over.]

If you issue a command which needs to use the stub, but the slave is busy running the client, you will get the message "You must stop the client and use the **wait** command." This is how you resynchronize the head and the stub. Where **continue** issues a command and doesn't wait for the reply, **wait** waits for a reply without issuing a command. What it gets might be a repetition of the reply to the previous command, or it might be a new reply (as after a **continue** or timeout).

STOP BUTTONS

The debugger supports something called a *stop button*. A stop button is any button which causes an external interrupt to occur. One of the easiest and least-obtrusive interrupts to wire a stop button to is the "ring indicator" interrupt on the serial (modem) port. The instructions for building a stop button are at the end of this section.

The stop button can be used any time when remote debugging, but should be used very carefully when debugging on one machine, especially when using GEMDOS or the BIOS for input from the keyboard and output to the screen. See the section **DB AND GEMDOS** in the chapter **OPERATING SYSTEM CONSIDERATIONS** for more information.

Hitting the stop button causes an interrupt to occur. The handler for this interrupt checks the stub state to see if it is in a trace or go. If so, the trace/go is stopped, and the exception "stop button" is reported to the head.

If the stub is executing, rather than the client, the fact that the stop button got hit is remembered. When the stub would restart the client, is stops instead. This will happen if, for example, the stub is processing a counted breakpoint whose count has not reached zero when you hit the stop button.

If the stub is idle (i.e. waiting for a command) and you hit the stop button, nothing happens: there is nothing to stop.

The stop button will only work if the interrupt priority level (IPL) does not mask the interrupt the button causes. In the case of the ring indicator on the ST, this means that the IPL must be less than 6. It usually is, but if you have an infinite loop at IPL 6 or 7 you can't use the stop button to get out of it. If you had a stop button wired to the non-maskable interrupt (NMI), the IPL would not matter.

Wiring A Ring-Indicator Stop Button

Ring Indicator is an easy stop button to wire because the connections you need are outside the ST itself, and are available on almost any RS232 connector. The debugger, therefore, contains code to initialize and enable the ring-indicator interrupt.

What you need to do is wire a button which normally connects Pin 22

(ring indicator) to Pin 7 (ground), but when pressed connects Pin 22 to Pin 4 (RTS, always asserted). This will detected as a change by the hardware, and cause the interrupt. Don't worry about debouncing the button: this is done in software. It turns out to be just lucky that the hardware does detect a change this way: zero volts (from Pin 7) is not really a valid signal on Pin 22.