Commodore®

AMIGA®

SOFTWARE TOOLKIT™

Commodore®

AMIGA®

SOFTWARE TOOLKIT™

DISCLAIMER

THIS INFORMATION IS PROVIDED "AS IS" WITHOUT REPRESENTATION OR WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, ANY REPRESENTATIONS OR ENDORSEMENTS REGARDING THE USE OF, THE RESULTS OF, OR PERFORMANCE OF THE INFORMATION, ITS APPROPRIATENESS, ACCURACY, RELIABILITY, OR CURRENTNESS. THE ENTIRE RISK AS TO THE USE OF THIS INFORMATION IS ASSUMED BY THE USER.

IN NO EVENT WILL COMMODORE, ITS AFFILIATED COMPANIES, NOR ITS EMPLOYEES, BE LIABLE FOR ANY DAMAGES, DIRECT, INDIRECT, INCIDENTAL OR CONSEQUENTIAL, RESULTING FROM ANY DEFECT IN THE INFORMATION, EVEN IF COMMODORE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

THIS DISCLAIMER SHALL SUPERSEDE ANY VERBAL OR WRITTEN STATEMENT TO THE CONTRARY.

CONTENTS

ACKNOWLEDGMENTS

Special thanks are extended to the following engineers and programmers who exerted a lot of mental, and physical, energy and made Software Toolkit possible:

David Conroy
Eric Cotton
Matt Dillon
Andy Finkel
Phillip Lindsay
Chuck McManis
Carl Sassenrath
Carolyn Scheppner
John Toebes (for the idea behind WatchMem)

This manual was compiled and edited by Isabelle Mc Vey, with sections supplied by Rob Peck.

Chapter 1. Introduction to Software Toolkit

The Software Toolkit diskette contains several utilities to assist developers with programming their Amiga computers. Software Toolkit is used in conjunction with the Workbench™ and contains utilities that allow you to:

*perform extensive debugging of programs
*monitor your system's memory and performance
*change the print parameters of screen dumps
*edit multiple files at one time
*link binary files into loadable files
*print the contents of binary files
*recover information from corrupt floppy disks

This list is only a sampling of Software Toolkit's capabilities. Software Toolkit's five drawers contain almost 20 utilities, and there are over a dozen additional AmigaDOS™ commands on the disk. Each of these drawers, icons, and commands is explained in this manual.

NOTATIONAL CONVENTIONS

This manual follows the standard AmigaDOS outline: Format, Template, Purpose, and Specification. Some text conventions you should be familiar with are:

COMMAND	AmigaDOS commands appear in capital letters to distinguish them from the rest of the text.
< >	Angled brackets enclose arguments that you need to specify. For instance, <filename> means that you must enter a specific file name along with the AmigaDOS command.
[ ]	Square brackets enclose options; options will be accepted by AmigaDOS but are not required.
{ }	Braces enclose items that can be repeated any number of times (or not at all). For example, {<args>} means that a number of arguments can be given but are not required.
...	Three dots indicate a series that can be continued.
|	A vertical bar is used to separate a list of options of which you can choose one. For instance, [OPT R|S|RS] means that you can choose the R option, the S option or both (RS) options.

The following standard abbreviations are used to define syntax:

arg	argument
char	character
expr	expression
n	number (always an integer)
sym	symbol

WORKBENCH VS. CLI

Most of the utilities can be run both from the Workbench and CLI. When running a utility from CLI, be sure to specify the correct, complete path name. For instance, you must type:

```
1> SwToolkit:Tools/CONTROL
```

(or df1:Tools/Control, if the Software Toolkit disk is in your first external disk drive). If you simply enter:

```
1> CONTROL
```

the CLI responds with an "unknown command CONTROL" message.

Another alternative is to use the AmigaDOS PATH command to change the system's search path. When your Amiga is booted with your Workbench disk, the system generally searches the System, Utilities, and C directories of the Workbench for any AmigaDOS commands entered. With the PATH command you can add directories on your Software Toolkit disk to this search list. For instance, go into the CLI and type:

```
1> PATH SwToolkit:Tools SwToolkit:Workbench SwToolkit:Debug SwToolkit:c
```

These directories will be added to the search path.

If you will be a frequent user of Software Toolkit, you may want to copy some of the more frequently used utilities to a special Workbench disk. If you are an extremely active developer, you may want to create several Workbench disks for various purposes.

ADDITIONAL REFERENCE MATERIALS

This manual assumes a working knowledge of the Amiga system. If you are unfamiliar with programming with the Amiga, you should consult the following reference materials:

The AmigaDOS Manual, published by Bantam Computer Books
The Amiga ROM Kernel Reference Manual: Libraries & Devices, published by Addison-Wesley
The Amiga ROM Kernel Reference Manual: Exec, published by Addison-Wesley

These books are available from your Amiga dealer or your local bookstore.

Chapter 2. WACK

This chapter explains the interactive, symbolic debugger called WACK, which is located in the C directory on the Toolkit disk. A working knowledge of programming on the Amiga is assumed. If you are a novice programmer, please refer to the reference material listed in Chapter 1 for additional information.

Format: WACK ["<program> [<arguments>]"] [COM <command file>] [OPT R|S|RS]

Template: WACK "PROGRAM,ARGUMENTS,COM/K,OPT/S"

Purpose: To perform interactive symbolic debugging of program files.

Specification:

As an aid to program development, WACK allows programs to be loaded and executed in a controlled environment. For instance, you can "trace" programs one instruction at a time, and you can set "breakpoints" on specific addresses. When these "breakpoints" are triggered, control is returned to WACK. You can then review the state of the machine's registers and memory.

<program> specifies the program file to be debugged. WACK automatically loads the program and its symbols, if any, into memory in a state ready for debugging. Note: If the program filename contains embedded spaces, then the filename must be enclosed with Escaped quotes. For example:

```
WACK "*"<program>*" <arguments>"
```

<arguments> specifies the arguments to be passed into the program. In other words, it specifies the program's command tail.

COM <command file> specifies a command file from which WACK attempts to load macros and key-macro definitions. This allows WACK to be customized with user defined macros prepared with an editor. If this option is not specified, the standard file "s:wack/wack.macros" is used instead. (In order for the system to find s:wack/wack.macros, you must use the ASSIGN command to assign S: to the Toolkit's s directory, or copy s:wack from the Toolkit disk to S:. Also, see the description of the load command on page 2-5.)

OPT R makes WACK system resident. Any program being debugged within WACK that makes a call to the EXEC function Debug() will now return control to WACK rather than invoking ROMWACK.

OPT S instructs WACK to establish communications via the serial device (9600 baud) rather than via the screen and keyboard. This option is used to remotely debug programs. It is particularly useful for debugging graphics applications since screen output is not interfered with in any way.

If OPT S is not specified, WACK creates its own window for the user command input and display output. This window is completely independent of any window used by the program being debugged. By using a separate window, WACK prevents the intermixing of debugging and program I/O. To maintain compatibility with

other types of display devices, such as a terminal on the serial port, WACK uses the window in a simple fashion. No cursor addressing or specially enhanced formatting is performed. All user input is accepted from the system keyboard while WACK's window is active.

When WACK opens its window, it displays its version number, then it attempts to load macro definitions from either a user specified file or s:wack/wack.macros. Next, WACK loads the specified program and its symbols. Finally, WACK displays a "ready" message.

If at any time you wish to exit from WACK, simply type "quit" and press the RETURN key. WACK then closes its window, unloads any program and symbols it had previously loaded, and terminates.

Examples:

```
1> WACK myprog
```

invokes WACK, and loads the program "myprog" and its symbols. Standard macro definitions are loaded from "s:wack/wack.macros".

```
1> WACK "play drum banjo" COM my.macros OPT S
```

invokes WACK, loads the program "play" and its symbols, passes the "drum" and "banjo" arguments to the program, reads in any macro definitions from the file "my.macros", and sets communications with WACK to be via the serial device.

Note: Although WACK loads the program to be debugged, it does not automatically run the program. Consequently, the current display address may not be valid for functions such as **showtask** (see EXEC.MACROS in Chapter 3), and a call to these functions may crash WACK.

WACK'S PROGRAMMING LANGUAGE

WACK's command interface provides quick interactive response through single key commands (key-macros) and maximum flexibility through the provision of a powerful command programming language.

To make full use of WACK, you should become familiar with its command programming language. Syntactically, it is similar to the programming language LISP. It provides a means of executing complicated command sequences and supports the creation of user defined commands called macros.

Three data types are supported by the language:

Numbers Numbers may either be in hexadecimal notation (the default) or in decimal notation. For decimal notation the number must be prefaced by a #.

Characters Characters are prefaced by a single quote. Control characters are represented by a caret (^) followed by the character. For example, CTRL-A is represented by '^a.

Strings Strings are represented by a sequence of characters enclosed in double quotes. For example: "Hello\n".

In the previous example, \n represents a newline character, as in the C programming language. All C type escape sequences are supported by WACK, including \ddd which represents an arbitrary byte-sized bit pattern, where ddd stands for an octal number.

The acceptable escape sequences are:

Sequence	Action
\n	new line
\t	tab
\b	backspace
\0	null character
\\	backslash
\'	single quote
\"	double quote
\ddd	byte with octal value ddd
\(	open parenthesis*
\)	close parenthesis*

**These commands must be used if a parenthesis is required as a character. They must also be used within a string since the syntax of the language specifies that any data item is automatically terminated if an "unescaped" parenthesis is encountered.*

All three data types (numbers, characters and strings) can be used as arguments to commands. Some commands expect their arguments to be of a specific data type. For instance, **print** expects its first argument to be a string. What **print** actually expects is a value that is the address of the string in memory. Just as in C, a string is represented in memory by a sequence of ASCII value bytes terminated by a null byte. Its value is the address of the first character in memory. All of the data types return a value; it is up to the command to interpret that value in the appropriate manner. (Throughout this manual, all WACK commands will be printed in lower case, boldface, letters to distinguish them from AmigaDOS commands.)

All WACK commands return a result. If parentheses are used to surround a command and its arguments, WACK will reply by printing out the value returned in hexadecimal notation.

```
(+ 8 3)
Value: B
```

For some commands, such as + shown above, parentheses are essential to discover the actual result. For other commands, such as **help**, the result returned is of no importance; therefore, there is no point enclosing them in parentheses (**help**, which displays a list of commands and their keyboard equivalents, always returns 0).

A command within parentheses can also be used as an argument to another command. This is called the nesting of commands and is allowed to any level. If you do not supply enough arguments to a command, WACK prompts for them with a question mark:

```
(+ 4)
?5
Value: 9
```

If too many arguments are supplied to a command, the extra ones are ignored:

```
(+ 4 5 6 7)
Value: 9
```

Input may be split over several lines since execution only takes place when the number of open and close parentheses are equal and the RETURN key is pressed:

```
(+
(- 7 5)
(* (+ 8 2) (/ 9 3)
))
Value: 20
```

The most frequently used commands are bound to predefined keys and can be executed by typing the relevant key. For instance, the **help** command is bound to both the "?" and the HELP key. Typing "?" or pressing HELP has the same effect as typing "**help**" and pressing RETURN.

Command files

A command file is a text file that contains WACK commands, principally macro and key-macro definitions. These definitions are loaded into WACK by using the COM <command file> item or the **load** command. This saves you from having to type long, complicated definitions at the keyboard. Command files can be commented by using a semi-colon. Any text following the semi-colon on the same line is ignored. For instance:

```
(macro printnum
   (print "%ld\n" (arg 1)) ;this is a comment
)
```

Program Files

Program files and their symbols can be loaded from within WACK by using the **load** and **bindsymbols** commands.

The **load** command can be used to load the macro definitions in the s:wack directory, macro definitions from some other file, or another program. As explained on page 2-1, WACK will automatically try to load the macro definitions from s:wack/wack.macros, but it will not *automatically* load any other macro files. To load other macro files you must specify the file by using the COM <command file> option or the **load** command. The format is:

```
load <filename> [<arguments>]
```

<filename> and <arguments> must both be specified as strings (enclosed in double quotes). If the file specified for <filename> is a program, WACK loads the specified program and its symbols and unloads any program that had been previously loaded. If the file specified is a macro filename, WACK will first check the s:wack directory. If the macro file is not in the s:wack directory, either use the PATH command to direct the search to the appropriate directory or specify the complete path when entering the filename.

You can load more than one macro file into WACK at the same time. However, you must use a separate **load** command for each filename.

If <argument> is supplied, it specifies the arguments to be passed into the program. If the file is a text file, WACK treats it as a command file. For example:

```
load "dir" "df1:"
```

loads the program "dir" and passes the argument string "df1:" to the program.

The **bindsymbols** command allows symbols to be bound to a program that is already in memory. The format is:

bindsymbols ‹filename›

If there is a program already loaded within WACK, **bindsymbols** loads any symbols in the ‹filename› program file and binds them to the loaded program. If there is no program loaded within WACK, **bindsymbols** assumes that the current display address specifies the start of a program segment list, and it attempts to bind the symbols loaded from the specified file to that program.

Once a program and/or its symbols have been loaded into WACK, the **unload** command can be used to remove the program from memory and to free its symbols. The format is:

unload

Output

WACK has one output command, **print**, which permits the translation of numeric quantities to character representation. It also allows the generation of formatted output. The format is:

print ‹control› ‹arg1› ‹arg2› ...

Print converts, formats, and prints its arguments on the screen under the control of the string ‹control›. The ‹control› string can contain two types of objects: ordinary characters and conversion specifications.

Ordinary characters are printed character by character. Conversion specifications, which are introduced by a percent sign (%), cause the print arguments (‹arg1› ‹arg2› . . .) to be converted into a specific format. In this way, **print** is similar in effect to the printf function in C.

The acceptable conversion specifications are:

Conversion	Print Specification
%ld	value is printed in decimal notation
%lx	value is printed in hexadecimal notation
%lc	value is printed as a character
%s	string (successive characters of the string are printed until a null character is encountered)

Here are a few examples of the use of **print**:

```
print "Hello, world\n"
Hello, world

print "Value in decimal - %ld\n" (7d0)
Value in decimal - 2000

print "Result is %lx\n" (+ 7a (+ 2c 3))
Result is A9

print "%s \(%lc\)\n" "character" `a
character (a)
```

EXAMINING MEMORY

Debugging a program often involves extensive examination of memory. WACK allows you to do this by:

* Absolute Addressing
* Relative Addressing
* Indirection
* Peeking Memory

Absolute Addressing

WACK provides two commands that permit absolute addressing: **show_frame** and **size_frame**.

The command **show_frame** is linked to the RETURN key. Absolute memory locations can be examined by typing the required machine address, in hexadecimal, then pressing RETURN. (You don't need to enter the full command.) For instance, if you enter:

```
2a7ec <RETURN>
```

You might see the following display. (Headings have been used to indicate the relative positions of the contents of the frame. Of course, these headings will not appear on the screen.)

```
Address     Memory-Contents         ASCII-values

044F10      732E 6C69 6272 6172     s . l i b r a r
044F18      7900 0000 0186 6006     y......^A.. `^F
```

Memory is displayed in fixed-size frames. Addresses are shown on the extreme left in hexadecimal notation. The contents of memory are displayed in word-size hexadecimal values. To the right of the Memory-Contents, the same loca-

tions are displayed as their equivalent ASCII characters. Control characters are represented by a preceding caret. Other non-printable characters are displayed as two dots (periods).

Decimal notation can be used if required; however, the decimal number must be prefaced by the character "#". The # symbol can be used anywhere a number is expected, so the memory location 2a7ec could also be referenced with the syntax #174060.

Within WACK there is the idea of the "current display address". Entering a value, such as 2a7ec, sets the current display address to that value and displays a frame starting at that address. Pressing RETURN at any time redisplays the current frame.

The current command returns the value of the current display address:

```
(current)
Value: 44F10
```

If a program and its symbols have been loaded into memory, then the **where** command can be used to report the location of the current display address relative to the program and its offsets. If the current display address is not within the range of the program, WACK will respond with the message "You're lost." (The **where** command is bound to the '@' key.) For instance:

```
@
DOSName + 0002   (hunk: 44D10  offset: 1FE)
```

In the previous example, DOSName is a program offset symbol. We are currently two bytes beyond DOSName and $1FE bytes away from the offset's hunk.

The **size_frame** command controls the amount of memory displayed in one frame. It is bound to the ':' key. The frame size may be modified by pressing the ':' key followed by the number of bytes desired. For instance, to request a frame size of 4 bytes, type:

```
:4
044F10 732E 6C69  s . l i
```

A frame size may be anything from 0 to 64K bytes; however, WACK always rounds the frame size up to an even number of bytes. A frame size of 0 is useful when writing to write-only registers.

Note: All of the Amiga custom chip registers (range $DFF000-$DFFFF) are either READ-ONLY or WRITE-ONLY. Trying to read a write-only register will destroy the value of that register and will probably crash your system! Trying to write to a read-only register will not produce a result of any value and may endanger future compatability.

To look at read-only custom chip registers, you must do it one register at a time. To do this, use the **size_frame** command to set the frame size to one word. For example:

```
:2
000000 0000 ....
dff006

DFF006 BE16 ..^V
```

To write to a write-only register, extra care must be taken. At no point before or after the write may a read cycle occur. This can be specified by setting the frame size to zero:

```
:0
dff180

DFF180=1234
```

Relative Addressing

There are a number of commands that update the current display address relative to its current value. We will continue using the example from the previous section and assume the current address is:

```
044F10 732E 6C69 6272 6172  s . l i b r a r
044F18 7900 0000 0186 6006  y......^A.. `^F
```

The **next_frame** command, bound to the '.' key, advances the current display address by a frame size and displays the frame at this new address:

```
.
044F20 0000 01B4 0001 1437 ....^A....^A^T 7
044F28 0021 0001 0000 0000 .. !..^A........
```

The **back_frame** command, bound to the ',' key, decreases the display address by a frame size:

```
,
044F10 732E 6C69 6272 6172  s . l i b r a r
044F18 7900 0000 0186 6006  y......^A.. `^F
```

The **next_word** command, bound to the '>' key and the SPACE key, advances the current display address by a single word:

```
>
044F12 6C69 6272 6172 7900  l i b r a r y..
044F1A 0000 0186 6006 0000 ....^A.. `^F....
```

The back_word command, bound to the '<' key and the BACKSPACE key, decreases the current display address by a single word:

```
<
044F10 732E 6C69 6272 6172  s . l i b r a r
044F18 7900 0000 0186 6006  y......^A.. `^F
```

The next_count command, bound to the '+' key, takes a single argument. It advances the current display address by the value of its argument:

```
+20
044F30 0000 0000 0000 0000 ................
044F38 FFFF FFFF FFFF FFFF ................
```

The back_count command, bound to the '-' key, takes a single argument. It decreases the current display address by the value of its argument:

```
-12
044F1E 6006 0000 01B4 0001  `^F....^A....^A
044F26 1437 0021 0001 0000 ^T 7.. !..^A....
```

Indirection

Programs often contain pointers to other parts of memory. Three WACK commands use such pointers: indirect, b_indirect, and exdirect.

The indirect command, bound to the '[' key, performs indirection on the contents of the longword at the current display address. The display address is set to the value of the contents of the longword at the current display address. For instance, if the current frame looks like this:

```
000004 0000 0628 00FC 0734 ....^F (....^G4
00000C 00FC 0736 00FC 0738 ....^G 6....^G8
```

Typing "[" produces:

```
[
000628 0000 1A4C 0000 07A2 ....^Z L....^G..
000630 0900 00FC 00A4 0400 ^I...... U..^D..
```

The b_indirect command, bound to the '{' key, treats the contents of the longword at the current display address as a BCPL pointer (BPTR) and left-shifts the longword by two before performing indirection. The display address is set to the value of the contents of the longword at the current display address left shifted by two. If the current frame is:

```
0210DC 0000 3089 0012 0002 .... 0....^R..^B
0210E4 112A 0001 AFEA 0001 ^Q *..^A......^A
```

Typing "{" produces:

```
{
00C224 0852 414D 2044 6973 ^H R A M   D i s
00C22C 6b00 0000 0000 0008  k...........^H
```

The **exdirect** command is the complement of **indirect** and **b_indirect**. **Exdirect** is bound to the ']' key and resets the current display address to the value it had before the last **indirect** or **b_indirect** command was executed. In fact, WACK maintains a stack of up to 20 indirections, so it is possible to use the indirection commands 20 times before using the **exdirect** command to "retrace" your movements through memory.

For instance, if the current frame looks like this:

```
000004 0000 0628 00FC 0734 ....^F (....^G4
00000C 00FC 0736 00FC 0738 ....^G 6....^G8
```

Typing "[" produces:

```
[
000628 0000 1A4C 0000 07A2 ....^Z L....^G..
000630 0900 00FC 00A4 0400 ^I...... U..^D..
```

Typing "[" again performs a further indirection:

```
[
001A4C 0000 258E 0000 0628 .... %......^F (
001A54 0900 00FC 559E 0400 ^I...... U..^D..
```

Any other commands can be executed without altering the indirection stack. For instance, a **next_word** command results in:

```
>
001A4E 258E 0000 0628 0900  %......^F (^I..
001A54 00FC 559E 0400 0090 .... U..^D......
```

Use "]" to backtrack the indirections. For example, typing "]" undoes the second indirection.

```
]
000628 0000 1A4C 0000 07A2 ....^Z L....^G..
000630 0900 00FC 00A4 0400 ^I...... U..^D..
```

Typing "]" again undoes the first indirection.

```
]
000004 0000 0628 00FC 0734 ....^F (....^G.4
00000C 00FC 0736 00FC 0738 ....^G 9....^G.8
```

Typing "]" again results in this message;

```
]
No indirections to undo
```

Peeking Memory

The @ command allows you to "peek" memory. It allows the contents of memory locations to be used within expressions, rather than just being displayed. @ has the following format:

```
(@ <address>)
```

@ returns the 16-bit word contents at the location in memory specified by the <address> argument. For instance:

```
(@ 6)
Value: 628
```

NOTE: Remember that the where command is also bound to the '@' key. Be careful not to confuse the two commands.

UPDATING MEMORY

Three commands allow you to update memory: **assign_mem**, :=, and @=.

The **assign_mem** command, bound to the '=' key, allows you to alter the word value (16 bits) stored at the current display address. **assign_mem** accepts a single argument; the value of the argument is the value to which the location is updated.

For instance, suppose the current display address is as follows:

```
02AAD0 6E64 7320 746F 2077  n d s   t o   w
02AAD8 6169 742C 2074 6865  a i t ,   t h e
```

Simply hitting the '=' key, without hitting RETURN, results in the following display:

```
02AAD0 6E64=
```

At the prompt, enter the argument value:

```
02AAD0 6E64=6162
```

The current address is updated to:

```
02AAD0 6162 7320 746F 2077  a b s   t o   w
02AAD8 6169 742C 2074 6865  a i t ,   t h e
```

If the current frame size is zero, the contents of the display address are not read before you update them. In this case typing "=" displays the following:

```
DFF09C xxxx =7fff
```

where xxxx represents the unread contents.

If after typing "=" you decide not to modify the contents, press RETURN without typing a value.

The := command allows you to update a specific location rather than the current display location. The format of := is:

```
(:= <address> <value>)
```

:= updates the word value (16 bits) at the location specified by the <address> argument with the new value specified by the <value> argument. The result returned is <value>.

For example:

```
(:= 2aad0 1234)
Value: 1234

2aad0
02AAD0 1234 7320 746F 2077 ^R 4 s   t o   w
02AAD8 6169 742C 2074 6865  a i t ,   t h e
```

The @= command has the same format as the := command:

```
(@= <address> <value>)
```

@= updates memory locations in the same manner as the := command, except that @= returns the original contents of the location being updated rather than the new contents.

MEMORY ACCESS

Three commands allow you to manipulate blocks of memory: **copy**, **find** and **fill**.

The **copy** command allows blocks of memory to be copied elsewhere. The format is:

```
copy <source> <finish> <destination>
```

<source> is the starting address of the block to be copied; <finish> is the ending address. The block is copied to the <destination> address.

For example, if the current frame is:

```
030000 7261 746F 722E 6900  r a t o r . i..
030008 0000 0000 0000 0000 ................
```

Enter:

```
copy 30000 30007 30008 <RETURN>
```

The result will be:

```
030000 7261 746F 722E 6900  r a t o r . i..
030008 7261 746F 722E 6900  r a t o r . i..
```

The **find** command searches a block of memory for a value. The format is:

```
(find <start> <finish> <value>)
```

<start> is the beginning address of the search; <finish> is the ending address. **Find** searches the block between <start> and <finish> for the word value (16-bit) specified by the <value> argument. It returns either the address of the location where the match was found or 0 if no match could be found.

For example:

```
030000 7261 746F 722E 6900  r a t o r . i..
030008 7261 746F 722E 6900  r a t o r . i..

(find 30000 30008 746F)
Value: 30002
```

The **fill** command allows a block of memory to be filled with a value. The format is:

```
fill <start> <finish> <value>
```

Fill updates the block of memory starting at the address specified by <start> and ending at the address specified by <finish> with the 16-bit word value specified by the <value> argument. The contents of the <finish> address are not changed. For example:

```
030000 7261 746F 722E 6900 r a t o r . i..
030008 7261 746F 722E 6900 r a t o r . i..

fill 30000 30008 1234
030000 1234 1234 1234 1234 ^R 4^R 4^R 4^R 4
030008 7261 746F 722E 6900 r a t o r . i..
```

ALLOCATING MEMORY

There are two commands used to allocate memory from within WACK for use during your debugging: **allocmemory** and **freememory**.

The **allocmemory** command allocates the number of bytes specified by its argument. Its format is:

```
(allocmemory <value>)
```

allocmemory returns a pointer to the base of the region. If allocation failed, it returns 0. For example:

```
(allocmemory 20)
Value: 2BE54
```

It is important to enclose the **allocmemory** command in parentheses; otherwise, the result returned, the pointer to the memory, will be lost. You will have allocated memory with no way of accessing it!

The **freememory** command frees the memory at the pointer previously allocated by **allocmemory**. Its format is:

```
(freememory <pointer>)
```

freememory returns the number of bytes in the region that was formerly allocated by **allocmemory**. For example:

```
(freememory 2BE54)
Value: 20
```

SYSTEM INTERACTION

The following list of commands simply call their corresponding Exec functions:

Command	Format	Function
permit	**(permit)**	Permits multitasking
forbid	**(forbid)**	Forbids multitasking
enable	**(enable)**	Enables interrupts
disable	**(disable)**	Disables interrupts

DISASSEMBLY

The **disassemble** command, bound to the ';' key, may be used to obtain a frame of program disassembly in 68000 mnemonics. The number of opcodes displayed depends on the current frame size and the number of bytes consumed by the opcodes.

WACK has the idea of a "current disassemble address", similar to the current display address. Any command that alters the current display address also sets the current disassemble address to the same location. The **disassemble** command updates the current disassemble address so that it becomes the location after the disassembly frame just displayed. However, **disassemble** does not alter the current display address.

WACK indicates the part of your program being disassembled by putting symbolic information to the right of the address field. For example:

```
02A9F4 [GLOBIN3+0000] move.l d0,d1
02A9F6 [GLOBIN3+0002] suba.l a0,a0
02A9F8 [GLOBIN3+0004] rts
02A9FA [OPENDOS+0000] movem.l a0-a1/a6,-(a7)
02A9FE [OPENDOS+0004] movea.l $4.w,a6
02AA02 [OPENDOS+0008] move.l a6,$2A7DC.l
```

In some cases it is not possible for WACK to give any symbolic information (for example, if you are disassembling memory which is not part of your program, or if your program did not have any symbolic information associated with it when loaded). After disassembly the current disassemble address is updated to the location after the location of the last instruction encountered.

ARITHMETIC

Several WACK commands allow you to perform arithmetic operations. The following table provides the format and an example of each command.

Command Format	Operation	Action
(+ <value> <value>)	Addition	Returns the sum of the two arguments.
Example: (+ 18a 9c) Value: 226		
(- <value> <value>)	Subtraction	Returns the difference of the two arguments.
Example: (- 3D3 2F1) Value: C2		
(- <value>)	Unary minus	Returns the negation of the single argument.
Example: (- 2dF) Value: FFFFFD21		
(* <value> <value>)	Multiplication	Returns the product of the two arguments.
Example: (* e3 #10) Value: 8DE		
(/ <value> <value>)	Division	Returns the quotient of the two arguments.
Example: (/ #121 #17) Value: 7		
(<< <value> <bit-count>)	Shift left	Returns the value of the first argument shifted to the left by the number of bits specified by the second argument.
Example: (<< 7a 4) Value: 7a0		

(>><value> <bit-count>)	Shift right	Returns the value of the first argument shifted to the right by the number of bits specified by the second argument.

Example: (>> 7a 3)
Value: F

(xor <value> <value>)	Bit-wise xor	Returns the result of "xoring" its two arguments.

Example: (xor 5 7)
Value: 2

As mentioned before, WACK commands can be nested within each other. In this way, arithmetic expressions can be built up. For example:

```
(* (- 33 (+ 2a 7)) (/ 5c 12))
Value: A
```

This is equivalent to (33 - (2a + 7)) * (5c / 12) in normal arithmetic notation.

SYMBOLS

WACK symbols are used for commands, offsets, user-defined commands (macros), registers and user variables. There are six types of WACK symbols: offsets, bases, primitives, registers, global variables, and macros. (Macros are covered in the next section of this chapter.)

A symbol name may contain any printable character, with the exception of a space or a parenthesis, but it cannot start with a digit. All WACK symbols are stored in a table that associates a value or binding with each symbol.

The most obvious example of WACK symbols are the program symbols that are loaded along with the program to be debugged. These program symbols are called offsets. When disassembling parts of a program, you can see that WACK labels each instruction using these offsets. The value of an offset is its absolute address.

A symbol may be used anywhere a value is expected. Thus symbol names may be used to reference various parts of your program's address space. For example:

```
startup

040040 23CF 0004 0274 23C0  #....^D^B t #..
040048 0004 027C 23C8 0004 ..^D^B | #....^D
```

Typing ";" (**disassemble**) produces:

```
040040 [startup+0000]  move.l  a7,$40274.l
040046 [startup+0006]  move.l  d0,$4027C.l
04004C [startup+000C]  move.l  a0,$40280.l
```

WACK creates some special symbols with the names `hunk_0, `hunk_1, . . . `hunk_n. Each of these symbols is called a base and is initialized with the values of the base addresses in memory of the related hunks.

All the WACK commands are primitive symbols.

The register symbols !D0, !D1, . . . !A6, !A7, !PC, !SP are predefined in the symbol table, enabling examination and alteration of the contents of the machine registers at any time while debugging your program.

Here are examples of their use:

```
(!SP)
Value: 22DA8

!a0
01F174 3A63 2F17 6163 6B20 : C / W A C K
01F17C 0000 0000 0000 0000 ^J ..........
```

To display the current values of all the machine registers, use the **regs** command. A typical display might look like this:

```
PC = 00002C268  SR = 0000      SSP= 00000000
D0 = 000000000  D1 = 00000000  D2 = 00000000  D3 = 00000000
D4 = 000000000  D5 = 00000000  D6 = 00000000  D7 = 00000000
A0 = 00001F414  A1 = 00009BF8  A2 = 00009440  A3 = 00000000
A4 = 000000000  A5 = 00F34D3C  A6 = 00F34D30  A7 = 00000000
```

User-Defined Symbols

The **addsymbol** command lets WACK users add their own symbols to the symbol table. These user-defined symbols are used as **global** variables. The format for **addsymbol** is:

```
(addsymbol <symbol> <value>)
```

The <symbol> and <value> arguments specify the name and value of the global variable. If a symbol of that name already exists in the table, the symbol will be overwritten with the new value. The **addsymbol** command returns the value of the newly created symbol. For example:

```
(addsymbol fred 4)
Value: 4
(+ fred 2)
Value: 6
```

The **remsymbol** command removes symbols from the symbol table. Its format is:

```
(remsymbol {<symbol>})
```

One or more symbols will be removed as specified by the {<symbol>} argument. The symbols may be of any type. Therefore it is possible to remove predefined symbols from the symbol table, as well as user-created symbols. If all the symbols specified in the {<symbol>} argument are in the table, **remsymbol** returns TRUE (non-zero). If not, it returns FALSE (zero).

The **remglobals** command removes all global-variable symbols from the symbol table and returns the number of symbols removed. Its format is:

```
remglobals
```

The **remmacros** command removes all macro definitions from the symbol table and returns the number of macros removed. Because macros take up a fair amount of memory, it is often advisable to use this command before loading or creating a new set of macros.

The **boundp** command is used to find out if a symbol is currently in the symbol table. Its format is:

```
(boundp <symbol>)
```

boundp returns TRUE if the symbol specified by the <symbol> argument is in the table; FALSE if it is not. For example:

```
(boundp help)
Value: 1 (equal to TRUE)
```

The = command allows you to assign new values to symbols. The format is:

```
(= <symbol> <value>)
```

The = command binds <symbol> to the new value specified by <value>. The symbol must be already in the symbol table, as = does not create symbols; it only changes their value. = works on all symbols except primitives and macros. If the wrong type of symbol is specified, WACK will give a "No location associated with symbol <symbol>" message. = returns <value> as its result. For example:

```
(= !D0 12345678)
Value: 12345678
(!D0)
Value: 12345678
```

Symbol Table Access

Think of the symbol table as a linked list of data structures called symbol nodes, one for each symbol in the table. Each symbol node contains the name of the symbol, its type (offset, base, global variable, primitive, macro, register), and its value. Several commands allow access to this table.

The **getsymbol** command is used to access selected symbol nodes. Its format is:

```
(getsymbol <symbol>)
```

getsymbol returns a pointer to the symbol node for the symbol specified by the <symbol> argument.

The **getkey** command allows access to symbols bound to key-macros (explained later in this chapter). Its format is:

```
(getkey <key-char>)
```

getkey returns a pointer to the symbol node of the symbol bound to the key specified by the <key-char> argument. If the key is not bound to any symbol, **getkey** returns 0.

The **symbolname** command returns a pointer to the name of the symbol. Its format is:

```
(symbolname <symnode>)
```

For example:

```
print "%s\n" (symbolname (getsymbol help))
help
```

The **symboltype** command returns a value denoting which type the symbol is. Its format is:

```
(symboltype <symnode>)
```

Here is a list of possible values:

Symbol Type	Value
Primitive	1
Offset	2
Base	3
Macro	5
Global	6
Register	7

The **symbolvalue** command returns the value bound to the symbol. Its format is:

```
(symbolvalue <symnode>)
```

This value has a different significance depending on the type of symbol. The values represent:

Symbol Type	Significance of Value
Primitive	Pointer to the internal function
Offset	Actual value
Base	Actual value
Macro	Pointer to structure representing the macro
Global	Actual value
Register	Pointer to data location holding the register's value

The **nextsymbol** command makes it possible to step through the symbol table. Its format is:

```
(nextsymbol <symnode>)
```

nextsymbol returns a pointer to the next symbol node in the table. If there are no more symbols, it returns zero. For example:

```
print "%s\n" (symbolname (nextsymbol (getsymbol step)))
stepover
```

WARNING: WACK does not check to see if the argument to these commands is a legal symbol node pointer. Thus it is possible to crash WACK by specifying an incorrect value.

MACROS

Macros, or user-defined commands, are symbols that are bound to expressions rather than to values. When a macro symbol is used in an expression, the expression that is bound to that symbol is evaluated. To increase the power of macros, WACK's language includes control flow commands and a mechanism for passing arguments into macros.

The **macro** command creates macro symbols. The format is:

```
(macro <symbol> <expression>)
```

It is similar to the **addsymbol** command except that it does not evaluate its second argument and that it creates a macro symbol rather than a variable symbol.

For example:

```
(macro calc
      (+ (* 1 2) (- 4 3))
)
Macro calc defined
Value: 0
```

Now, whenever the symbol "calc" is used within an expression, (+ (* 1 2) (- 4 3)) will be evaluated and its value returned as the value of the symbol calc. For example:

```
(calc)
Value: 3
(* calc 2)
Value: 6
```

Macro Specific Commands

There are four primitive symbols that are used within macro definitions: **arg**, **nargs**, **local**, and **return**.

The **arg** command allows arguments to be passed into macros. The format is:

```
(arg <arg position> [<prompt>])
```

arg returns the value of the macro's nth argument where <arg position> specifies n. For example:

```
(macro double (* (arg 1) 2))

(double 3)
Value: 6
```

If no argument was supplied in the position requested, then WACK prompts for an argument. The string <prompt>, if specified, is used for this purpose; if not, a question mark is used, as with primitives.

The **nargs** command takes no arguments and simply returns the number of arguments specified at the macros invocation. It does not attempt to evaluate them. For example:

```
(macro my_nargs (nargs))
macro my_nargs defined
(my_nargs arg1 arg2 arg3)
Value: 3
```

The local command creates local symbols for use while the macro is being evaluated. When the macro is terminated or evaluated, the local symbols disappear. The format is:

```
(local {<symbols>})
```

Local symbols are not stored in the symbol table but in their own local stack. They are given the initial value of zero. For example:

```
(macro remainder (local i j)
      (= i (arg 1 "dividend?")) (= j (arg 2 "divisor?"))
      (- i (*i (/ i j )))
)
Macro remainder defined
Value: 0
(remainder 8 3)
Value: 2
```

The expression (= <local> (arg n)) is often used within macros to bind the value of an argument to a local symbol.

The return command is used to terminate macros prematurely. Its format is:

```
(return [<value>])
```

return immediately terminates the evaluation of the current macro and returns <value> as the macro's result. If <value> is not specified, return returns 0.

Note: return should not be confused with <RETURN>, which refers to the RETURN key.

The listmacro command allows any macro previously entered to be listed to the screen for inspection. Its format is:

```
listmacro <macro-name>
```

Additional Wack Commands

Several commands that are not solely for use within macros but which greatly increase their power are covered in this section.

A number of **comparison commands** are available in WACK. Comparison commands return a TRUE or FALSE result. The value for FALSE is zero; any other value is taken to be TRUE. However, all the comparison commands actually return 1 for TRUE.

COMPARISON COMMANDS

Command	Format	Action
<	(< <value> <value>)	less than
>	(> <value> <value>)	more than
==	(== <value> <value>)	equality
<=	(<= <value> <value>)	less than or equal to
>=	(>= <value> <value>)	more than or equal to
!=	(!= <value> <value>)	not equal to

All comparison commands compare their first argument with their second. For instance, < returns TRUE if its first argument is less than its second argument. For example:

```
(== 3 2)                    (!= 3 2)
Value: 0                    Value: 1
```

Logical commands perform logical functions and return a TRUE or FALSE result. They can also be used in conjunction with the comparison commands.

LOGICAL COMMANDS

Command	Format	Action
and	(**and** <value> <value>)	logical and
or	(**or** <value> <value>)	logical or
not	(**not** <value>)	logical not

For example:

```
(and (> 2 3) (== 1 1))
Value: 0

(or (>= 1 2) 3)
Value: 3

(not 1)
Value: FFFFFFFE
```

WACK has four control flow commands: if, select, while and for. These commands specify the order in which other commands are executed.

The if command allows commands to be evaluated conditionally. Its format is:

```
if <condition> <then-expr> [<else-expr>]
```

The <condition> is evaluated; if it is TRUE (a non-zero value), <then-expr> is evaluated. If it is FALSE (a value of zero), and if <else-expr> is present, <else-expr> will be evaluated instead. Note that <else-expr> is optional. For instance:

```
if (<= 0 1) (print "TRUE\n") (print "FALSE\n")
TRUE
```

The select command allows selective evaluation of commands; select takes two or more arguments, but the number of arguments must be even. Its format is:

```
(select <condition> <expr> {<condition> <expr>})
```

The first <condition> is evaluated. If it is TRUE, the following <expr> is evaluated. If FALSE, the next <condition> is evaluated. This series of evaluation continues until either a <condition> evaluates to TRUE, in which case the <expr> following it is evaluated, or no condition is left to evaluate, in which case it returns 0. For example:

```
(select 0 (+ 2 3) 1 (+ 3 4))
Value: 7
```

The while command allows repeated evaluation of command sequences. Its format is:

```
(while <condition> <expr>)
```

<condition> is evaluated; if it is TRUE, <expr> is evaluated. This is repeated while <condition> is TRUE. For example:

```
(macro loop (local i)
      (while (< i 5)
      (
      (print "%ld\n" i)
      (= i (+ i 1))
      ))
)
macro loop defined
loop
0
1
2
3
4
```

The for command also allows repeated evaluation of commands. Its format is:

```
(for <init-expr> <condition> <loop-expr> [<body-expr>])
```

<init-expr> is evaluated first, once only. Next, <condition> is evaluated, if it is TRUE, then both <body-expr>, if supplied, and <loop-expr> are evaluated in that order. This is repeated while <condition> evaluates to TRUE. Note that the last argument is optional. For example:

```
(macro loop
      (for (local i) (< i 5) (= i (+ i 1))
(print "%ld\n" i))
)
macro loop defined
loop
0
1
2
3
4
5
```

Macro Examples

Following are two examples to help explain the use of macros.

Example 1 shows a macro that places a string at a supplied address in memory:

```
(macro puts (local str addr)
      (= str (arg 1 "Enter string? "))
      (= addr (arg 2 "Enter address? "))
      (while (!= (@b str) 0)
      (
      (:=b addr (@b str))
            (= str (+ str 1))
            (= addr (+ addr 1))
      ))
)
macro puts defined
(puts "foo" 50000)
Value: 3
50000
050000 666F 6F00 0000 0000  f o o.....
```

Example 1

Example 2 shows macros to peek and poke long and byte values in memory:

```
Long Peek:  (macro @l (local addr)
                (= addr (arg 1 "address? "))
                (or (<< (@ (addr) 10) (@ (+ addr) 2))
            )
            macro @l defined

Byte Peek:  (macro @b (local addr)
                (= addr (arg 1 "address? "))
            (>> (@ addr) 8)
            )
            macro @b defined

Long Poke:  (macro :=l (local addr value)
                (= addr (arg 1 "address? "))
                (= value (arg 2 "value? "))
                (:= addr (>> (value 10))
                (:= (+ (addr 2) value)
            )
            macro :=l defined

Long Poke:  (macro @=l (local addr value)
                (= addr (arg 1 "address? "))
                (= value (arg 2 "value? "))
                (or
                      (<< (@= addr (>> (value 10)) 10)
                      (@= (+ addr 2) value))
            )
            macro @=l defined

Byte Poke:  (macro :=b (local i addr value)
                (= addr (arg 1 "address? "))
                (= value (arg2 "value? "))
                (= i (@ addr))
                (:= addr (or (<< (and value ff) 8) (and i ff)))
            )
            macro :=b defined
```

Example 2

Additional Macros

There are four macros that show how you can customize WACK to your own needs: **printname**, **printarg**, **stackframe** and **backtrace**. These four macros, using some of the macros defined above, make up a backtrace facility for programs written in the language BCPL. Even if these macros are not of any direct use to you, they should give you some idea of how macros can be put together.

The macro **stackframe** is designed to be used when a new routine is entered. It prints out the name of the current routine and the first four arguments passed to it. BCPL has an upward stack pointed to by the register A1. When a routine is entered, its first four arguments are on the stack at 0(A1), 4(A1), 8(A1), and 12(A1). Stored behind it on the stack are the base of the routine just entered, the address to return to when finished, and the previous stack pointer at -4(A1), -8(A1) and -12(A1), respectively. One further thing you need to know to understand these macros is that BCPL code contains routine names, truncated to seven characters, compiled just behind the base address of the routine to which they refer.

The **printname** macro lets you print the name of the routine you have just entered. It needs to be passed the base address of the current routine.

```
(macro printname (local i name)
       (= name (- (arg 1) 8))
       (for (= i 1) (< i 8) (= i (+ i 1))
       (print "%1c" (@b (+ name i))))
)
```

To print out an argument in a field width of eight, use the **printarg** macro. It needs to be passed the address where the argument is stored.

```
(macro printarg
      (print " %81x" (@1 (arg 1)))
)
```

The **stackframe** macro does all the work. It prints out a whole **stackframe** (the name of the routine and the first four arguments) given the stack pointer on entry to a routine. Notice too that the value which it returns is the stack pointer for the previous **stackframe**.

```
(macro stackframe (local sp i pstk name)
      (= sp (arg 1 "stack pointer? "))
      (= pstk (@1 (- sp #12))
      (= name (@1 (- sp #4))
      (printname name)
      (for (= i 0) (< i 4) (= i (+ i 1))
      (printarg (+ sp (* i 4)))
      (print "\n")
      (return pstk)
)
```

The **backtrace** macro takes one argument: the number of frames to display. Notice that the argument passed to the macro **stackframe** is A1 for the first stackframe, then the value returned by the previous call to stackframe.

```
(macro backtrace (local frames i j)
       (= frames (arg 1 "frames? "))
       (for (= i 0) (< i frames) (= i (+ i 1))
       (if (== i 0)
            (= j (stackframe !A1))
            (= j (stackframe j))
      )
      )
)
```

Finally, you can bind a key to the **backtrace** macro by typing the following command line:

```
(key '^b backtrace)
```

Suppose that you have hit a breakpoint at the beginning of a routine. You can do a stack backtrace by simply pressing CTRL-B. It prompts for the number of frames to be displayed, then goes ahead and does the backtrace. For example:

```
^b
Frames? 4
deplete       10      A48C        12        13
wrch           A     20B6C       456      A7C
writef     2876C         2         3     20F28
start      20F1C     2BB5A     2BB64         1
```

The above display contains the following information: the current routine is deplete; the first four arguments are hex 10, A48C, 12, and 13; it was called by wrch, which was called by writef, which was called by start.

KEY-MACROS

As stated before, an individual key can be bound to any type of symbol. However, a key can only be bound to one symbol at any one time. Pressing a key that is bound to a symbol has the same effect as typing the full symbol name. There are four WACK commands associated with key-macros: **key**, **remkey**, **is_key** and **keys**.

The **key** command defines key-macros. Its format is:

key ‹key-char› ‹symbol›

The key specified by the character ‹key-char› is bound to the symbol specified by the ‹symbol› argument. Once a key is bound to a symbol, pressing that key will have the same effect as typing the symbol's name. For example:

```
key '^a print
Key '^a' bound to 'print'
```

Typing CTRL-A produces:

```
? "Hello \n"
Hello
```

Notice that a '^' (caret) character is used to specify a control character. Likewise a '~' (tilde) character can be used before the numbers from 0 to 9 to specify the function keys F1 to F10, respectively:

```
key '~6 print
Key 'F7' bound to 'print'
```

The **remkey** command destroys key-macro definition. Its format is:

remkey ‹key-char›

remkey unbinds the key specified by the character ‹key-char› from the symbol to which it was bound.

The **is_key** command returns TRUE if the current macro being evaluated is a key-macro (invoked by a single key press). Otherwise it returns FALSE.

The **keys** command lists all the keys currently bound to symbols and the symbols to which they are bound.

BREAKPOINTS AND TRACING

Eight commands handle breakpoints and tracing: **set**, **halt**, **is_break**, **clear**, **reset**, **step**, **stepover** and **go**.

The format for the **set** command is:

set ‹address› [‹count› [‹macro›]]

The **set** command sets a breakpoint at the instruction specified by the ‹address› argument. The ‹count› argument specifies how many times the breakpoint must be encountered during execution before it is "triggered". (When a breakpoint is triggered the program is interrupted and control is returned to WACK.) If the ‹count› argument is not specified, the breakpoint will be triggered the first time it is encountered during execution. Once the breakpoint has been triggered, it is removed.

The ‹macro› option allows commands (primitives or macros) to be "bound" to breakpoints so that the command executes when the breakpoint is triggered.

The format of the **halt** command is:

halt ‹address› [‹count› [‹macro›]]

The **halt** command sets a breakpoint that is not removed when it is triggered but is reset to be in the same state as when it was generated. In all other respects **halt** behaves like the **set** command.

The **is_break** command returns TRUE if the current macro being evaluated was invoked by a breakpoint. Otherwise, it returns FALSE. The **is_break** command does not take any arguments.

The format of the **clear** command is:

clear ‹address›

If a breakpoint exists at the specified address, the **clear** command removes it.

The **reset** command takes no arguments and removes all breakpoints.

Tracing is achieved with the **step** command. Its format is:

step [‹count›]

The ‹count› argument specifies how many instructions are executed before control is returned to WACK. If no ‹count› argument is specified, then a single instruction is executed.

The **stepover** command takes no arguments. It sets a breakpoint at the next instruction and executes. This is useful during tracing to "step over" a JSR or BSR instruction rather than having to trace through the entire called routine.

The **go** command resumes execution at the current value of the program counter. The program counter can be updated by typing a command of the form:

```
(= !PC <value>)
```

When a breakpoint is hit, tracing is complete, or an internal exception occurs, WACK halts the program, explains briefly what exception has occurred, and displays the contents of the registers in the same manner as the **regs** command.

The **gos** primitive behaves exactly like **go** except that upon termination of the program being executed only the program itself is unloaded. The program's symbols remain in the symbol table. (The **unload** primitive, explained earlier, can be used to remove the symbols.)

The **resume** command is used to continue execution of a WACK-executed program that has made a call to Debug().

QUICK REFERENCE LIST OF COMMANDS

The commands are listed in the order that they are discussed in the text.

Input/Output:
quit
help
load
bindsymbols
unload
print

Examining Memory:
show_frame
current
where
size_frame
next_frame
back_frame
next_word
back_word
next_count
back_count
indirect
b_indirect
exdirect
@

Updating Memory:
assign_mem
:=
@=

Memory Access:
copy
find
fill

Allocating Memory:
allocmemory
freememory

System Interaction:
permit
forbid
enable
disable

Disassembly:
disassemble

Arithmetic:
+
-
*
/
<<
>>
xor

Symbol Control:
regs
addsymbol
remsymbol
remglobals
remmacros
boundp
=
getsymbol
getkey
symbolname
symboltype
symbolvalue
nextsymbol

Macros:
macro
arg
nargs
local
return
listmacro

Comparisons:
<
>
==
<=
>=
!=

Logicals:
and
or
not

Control Flow:
if
select
while
for

Additional Macros:
printname
printarg
stackframe
backtrace

Key Macros:
key
remkey
is_key
keys

Breakpoints and Tracing:
set
halt
is_break
clear
reset
step
stepover
go
gos
resume

ALPHABETICAL COMMAND LIST

Command	Format	Function
+	(+ ‹value1› ‹value2›)	Returns the sum of ‹value1› and ‹value2›.
-	(- ‹value1› [‹value2›])	Returns the negation of ‹value› if only one argument is supplied; otherwise returns the difference of ‹value1› - ‹value2›.
*	(* ‹value1› ‹value2›)	Returns the product of ‹value1› and ‹value2›.
/	(/ ‹value1› ‹value2›)	Returns the quotient of ‹value1› / ‹value2›.
<<	(<< ‹value› ‹bit-count›)	Returns ‹value› shifted left by ‹bit-count›.
>>	(>> ‹value› ‹bit-count›)	Returns ‹value› shifted right by ‹bit-count›.
==	(== ‹value› ‹value›)	Returns TRUE if its two arguments are equal, FALSE otherwise.
>	(> ‹value1› ‹value2›)	Returns TRUE if ‹value1› is greater than ‹value2›, FALSE otherwise.
<	(< ‹value1› ‹value2›)	Returns TRUE if ‹value1› is less than ‹value2›, FALSE otherwise.
>=	(>= ‹value1› ‹value2›)	Returns TRUE if ‹value1› is greater than or equal to ‹value2›, FALSE otherwise.
<=	(<= ‹value› ‹value›)	Returns TRUE if ‹value1› is less than or equal to ‹value2›, FALSE otherwise.
!=	(!= ‹value› ‹value›)	Returns TRUE if its two arguments are not equal, FALSE otherwise.
@	(@ ‹address›)	Returns the contents of the word at ‹address›.

:=	(:= <address> <value>)	Sets the contents of the word at <address> to <value>. Returns <value>.
@=	(@= <address> <value>)	Sets the contents of the word at <address> to <value> and returns the previous contents.
=	(= <name> <value>)	Assigns a value to an existing symbol, overwriting the old value of the symbol. Returns <value> of the symbol.
addsymbol	(addsymbol <symbol> <value>)	Defines a new global-variable symbol with name <symbol>, initially bound to <value>. Returns the value of the symbol.
allocmemory	(allocmemory <value>)	Allocates <value> bytes of memory and returns a pointer to the base of the region, or 0 if allocation failed.
and	(and <value> <value>)	Returns the result of applying a logical AND to its two arguments.
arg	(arg <arg position>) [<prompt-string>]	Returns the value of the macros nth argument, where <arg position> specifies n. If no such argument was supplied in the macro invocation, it asks for one interactively, using <prompt-string> as a prompt.
assign_mem *Bound to '='*	(assign_mem <value>)	Sets the contents of the current word to <value>.
backtrace	(backtrace <value>)	Displays a stack backtrace of <value> frames.
back_count *Bound to '-'*	(back_count <value>)	Moves the display frame back by <value> bytes.
back_frame *Bound to ','*	(back_frame)	Moves the display frame back by one frame size.
back_word *Bound to '<' & BACKSPACE*	(back_word)	Moves the display frame back by a single word.

b_indirect *Bound to '{'*	**(b_indirect)**	Moves display frame by indirecting off the BPTR assumed to be at the current longword.
bindsymbols	**(bindsymbols** ⟨filename⟩)	Loads and binds the symbols in the program specified by ⟨filename⟩ to a program already in memory.
boundp	**(boundp** ⟨symbol⟩)	Returns TRUE if ⟨symbol⟩ is currently bound, FALSE otherwise.
clear	**(clear** ⟨address⟩)	Clears the breakpoint at ⟨address⟩. Returns TRUE if there was a breakpoint, FALSE otherwise.
copy	**(copy** ⟨source⟩ ⟨finish⟩ ⟨destination⟩)	Copies the block of memory starting at address ⟨source⟩ and ending at address ⟨finish⟩ to address ⟨destination⟩.
current	**(current)**	Returns the address of the current frame.
disable	**(disable)**	Disables interrupts.
disassemble *Bound to ';'*	**(disassemble)**	Displays the current frame as disassembled 68000 mnemonics.
enable	**(enable)**	Enables interrupts.
exdirect *Bound to ']'*	**(exdirect)**	Returns the display frame to the previous indirected address.
fill .	**(fill** ⟨start⟩ ⟨finish⟩ ⟨value⟩)	Fills the block of memory starting at address ⟨source⟩ and ending at address ⟨finish⟩ with the word ⟨value⟩.
find	**(find** ⟨start⟩ ⟨finish⟩ ⟨value⟩)	Searches the block of memory starting at address ⟨source⟩ and ending at address ⟨finish⟩ for the first occurrence of ⟨value⟩. Returns the address of ⟨value⟩ or 0 if no match was found.

for	**(for** ‹init-expr› ‹condition› ‹loop-expr› [‹body-expr›]**)**	Evaluates ‹init-expr› first, once only. Next, if ‹condition› evaluates to TRUE, both ‹body-expr› and ‹loop-expr› are evaluated in that order; this is repeated while ‹condition› evaluates to TRUE.
forbid	**(forbid)**	Forbids multitasking.
freememory	**(freememory** ‹pointer›**)**	Frees the memory at the ‹pointer› previously allocated by **allocmemory**. Returns the number of bytes actually freed.
getkey	**(getkey** ‹key-char›**)**	Returns a pointer to the symbol node of the macro that is bound to key ‹key-char› or 0 if the key is not bound.
getsymbol	**(getsymbol** ‹symbol›**)**	Returns a pointer to the symbol node for ‹symbol› or 0 if the symbol is not in the symbol table.
go	**(go)**	Executes from the current value of the Program Counter. If executing a WACK-loaded program, both the program and its symbols are unloaded upon the program's termination.
gos	**(gos)**	Executes from the current value of the Program Counter. Differs from **go** in that the program's symbols are not unloaded after the program terminates.
halt	**(halt** ‹address› [‹count› [‹symbol›]]**)**	Sets a retriggerable breakpoint at ‹address›. If ‹count› is specified, **halt** determines how many times the breakpoint is reached before being triggered. (Defaults to 1 if ‹count› is not specified). ‹symbol›, if specified, is evaluated upon hitting the breakpoint.
help *Bound to '?' and ‹HELP›*	**(help)**	Displays a list of commands and their keyboard equivalents.

if	**(if** ‹condition› ‹then-expr› [‹else-expr›]**)**	Returns ‹then-expr› if ‹condition› evaluates to TRUE; otherwise it returns either ‹else-expr›, if there is one, or 0.
indirect *Bound to '['*	**(indirect)**	Moves the display frame by indirecting off the current longword.
is_break	**(is_break)**	Returns TRUE if a breakpoint macro is currently being evaluated, FALSE otherwise.
is_key	**(is_key)**	Returns TRUE if a key macro is currently being evaluated, FALSE otherwise.
key	**(key** ‹key-char› ‹symbol›**)**	Binds the key ‹key-char› to ‹symbol›.
keys	**(keys)**	Lists all currently bound keys and their bindings.
listmacro	**(listmacro** ‹macro-name›**)**	Lists the macro ‹macro-name› to the screen for inspection.
load	**(load** ‹filename› [‹argument›]**)**	Loads the program ‹filename› and its symbols, and specifies ‹arguments›, if present. However, when ‹filename› specifies a text file rather than a program, ‹filename› is treated as a macro file, and any definitions it contains are loaded.
local	**(local** {‹symbol›}**)**	Defines any number of local symbols within a macro. Each ‹symbol› specifies a single local symbol name; all locals have an initial value of 0.
macro	**(macro** ‹symbol› {‹expr›}**)**	Defines a new macro (or redefines an old one) binding ‹symbol› to ‹expr›.
nargs	**(nargs)**	Returns the number of arguments specified at the macro invocation.

nextsymbol	**(nextsymbol <symnode>)**	Returns a pointer to the next symbol or 0 if there are no more symbols.
next_count *Bound to '+'*	**(next_count <value>)**	Moves the display frame forward by <value> bytes.
next_frame *Bound to '.'*	**(next_frame)**	Moves the display frame forward by one frame size.
next_word *Bound to '>' & SPACE*	**(next_word)**	Moves the display frame forward by a single word.
not	**(not <value>)**	Returns the logical negation of <value>.
or	**(or <value> <value>)**	Returns the result of applying a logical OR to its two arguments.
permit	**(permit)**	Permits multitasking.
print	**(print <control> <arg 1> <arg 2>. . .)**	Converts, formats, and prints all its arguments on the screen under the control of <control>; similar in effect to printf in C.
printarg	**(printarg <address>)**	Prints out an argument in a field width of eight. It needs to be passed the address where the argument is stored.
printname	**(printname <address>)**	Prints the name of the routine you have just entered.
quit *Bound to CTRL_D*	**(quit)**	Quits WACK.
regs	**(regs)**	Displays the values of all registers.
remglobals	**(remglobals)**	Removes all bindings for global-variable symbols.
remkey	**(remkey [<key-char>])**	Unbinds the key(s) <key-char> from the symbol(s) it was bound to. Returns TRUE if all bindings were removed, FALSE otherwise.

remmacros	**(remmacros)**	Removes all bindings for macros, and returns the number of macros removed.
remsymbol	**(remsymbol** {<symbol>})	Removes any binding for given <symbol>s. Returns TRUE if all given <symbol>s were removed, FALSE otherwise.
reset	**(reset)**	Clears all breakpoints.
resume	**(resume)**	Resumes execution from the current value of the Program Counter.
return	**(return** [<value>])	Terminates the macro evaluation. The macro returns <value> if specified, 0 otherwise.
select	**(select** <condition> <expr> [<condition> <expr>])	Returns the first <expr> whose <condition> (the one directly preceding it) evaluates to TRUE. If there is none, it returns 0.
set *Bound to '^'*	**(set** <address> [<count> [<macro>]])	Sets a non-retriggerable breakpoint at <address>. <count>, if specified, determines how many times the breakpoint is reached before being triggered (defaults to 1 if <count> is not specified). <macro>, if specified, is evaluated upon hitting the breakpoint.
show_frame *Bound to <RETURN>*	**(show_frame)**	Displays the current frame.
size_frame	**(size_frame** <value>)	Alters the display frame size to <value> bytes (rounded up to an even number).
stackframe	**(stackframe)**	Prints out a whole stackframe given the stack pointer on entry to a routine.
step	**(step** <count>)	Executes <count> instruction (defaults to 1 if <count> is not specified).

stepover	**(stepover)**	Sets a breakpoint in the next instruction before executing normally.
symbolname	**(symbolname** ⟨symnode⟩**)**	Returns a pointer to the name of the symbol specified by ⟨symnode⟩.
symboltype	**(symboltype** ⟨symnode⟩**)**	Returns the type of the symbol specified by ⟨symnode⟩.
symbolvalue	**(symbolvalue** ⟨symnode⟩**)**	Returns the value of the symbol specified by ⟨symnode⟩.
unload	**(unload)**	Unloads a program brought in to memory by WACK and removes any symbols associated with it.
where *Bound to '@'*	**(where)**	Shows symbol name (offset) for current-frame address.
while	**(while** ⟨condition⟩ ⟨expr⟩**)**	Evaluates ⟨expr⟩ if ⟨condition⟩ evaluates to TRUE. This is repeated while ⟨condition⟩ evaluates to TRUE.
xor	**(xor** ⟨value⟩ ⟨value⟩**)**	Returns the result of applying a logical XOR to the two arguments.

Chapter 3. WACK Macro Files

This chapter lists the macro files included in the s:wack directory on the Software Toolkit disk. The directory contains four macro files: wack.macros, exec.macros, graphics.macros, and intuition.macros.

WACK.MACROS

Unless overridden by the COM <command file> option, WACK will automatically load the file wack.macros. The macros in this file provide WACK with additional commands, many of which are also used by the other three macro files, exec.macros, graphics.macros, and intuition.macros. This does not, however, prevent the user from adding his own macros and key bindings.

Command	Format	Function
@b	(@b <address>)	Returns the contents of the byte at <address>.
:=b	(:=b <address> <value>)	Sets the contents of the byte at <address> to <value>.
@l	(@l <address>)	Returns the contents of the longword at <address>.
:=l	(:=l <address> <value>)	Sets the contents of the longword at <address> to <value>.
@=l	(@=l <address> <value>)	Sets the contents of the longword at <address> to <value> and returns the previous contents.
strcmp	(strcmp <string1> <string2>)	Compares <string1> with <string2>. Returns 0 if the strings are identical, non-zero otherwise.
puts	(puts <string> <address>)	Places <string> in memory at <address>.
LTOB	(LTOB <address>)	Converts the BPTR at <address> to an APTR.
Node	(Node <address>)	Defines a symbolic EXEC Node structure at <address>. If <address> is omitted, then the structure is defined at the current address.
signb	(signb <byte>)	Prints the value of a signed byte.
BSTR	(BSTR <address>)	Prints the BCPL string (BSTR) at <address>.

EXEC.MACROS

The macros and key bindings in the exec.macros file were designed to facilitate the examination of EXEC structures and system lists. Please note that many of these macros require some of the macros found in wack.macros.

Command	Format	Function
ExecBase	(ExecBase)	Defines a symbolic EXEC ExecBase structure.
Library	(Library [<address>])	Defines a symbolic EXEC Library structure at <address>*.
List	(List [<address>])	Defines a symbolic EXEC List structure at <address>*.
MemChunk	(MemChunk [<address>])	Defines a symbolic EXEC MemChunk structure at <address>*.
MemHeader	(MemHeader [<address>])	Defines a symbolic EXEC MemHeader structure at <address>*.
Process	(Process [<address>])	Defines a symbolic EXEC Process structure at <address>*.
Resident	(Resident [<address>])	Defines a symbolic EXEC Resident structure at <address>*.
Task	(Task [<address>])	Defines a symbolic EXEC Task structure at <address>*.
ebase *Bound to F9*	(ebase)	Displays some important ExecBase members and their values.
devices *Bound to F3*	(devices)	Displays current device list.
libraries *Bound to F1*	(libraries)	Displays current library list.
resources *Bound to F4*	(resources)	Displays current resource list.
nodes	(nodes)	Displays the node list starting from the current address.

**If <address> is omitted, the structure is defined at the current address.*

memory *Bound to F5*	**(memory)**	Displays the current memory allocation list.
modules *Bound to F2*	**(modules)**	Displays system modules.
tasks *Bound to F6*	**(tasks)**	Displays all tasks.
showtask *Bound to F7*	**(showtask [<address>])**	Formats <address> (or current frame if <address> is omitted) as a task.
showprocess *Bound to F8*	**(showprocess [<address>])**	Formats <address> (or current frame if <address> is omitted) as a process.

GRAPHICS.MACROS

The graphics.macros package contains GRAPHICS macros, many of which require some of the macros found in wack.macros.

Command	Format	Function
GfxBase	**(GfxBase)**	Defines a symbolic GRAPHICS GfxBase structure.
Layer	**(Layer [<address>])**	Defines a symbolic Layer structure at <address>*.
Layer_Info	**(Layer_Info [<address>])**	Defines a symbolic Layer_Info structure at <address>*.
gfxbase	**(gfxbase)**	Displays some important GfxBase members and their values.
layer	**(layer [<address>])**	Formats <address>* as a Layer structure.
linfo	**(linfo [<address>])**	Formats <address>* as Layer_Info structure.

**If <address> is omitted, the structure is defined at the current address.*

INTUITION.MACROS

The intuition.macros package allows the programmer to easily examine INTUITION structures. Many of these macros require some of those found in wack.macros.

Command	Format	Function
IntuitionBase	(IntuitionBase)	Defines a symbolic IntuitionBase structure.
Screen	(Screen [<address>])	Defines a symbolic Intuition Screen structure at <address>*.
Window	(Window [<address>])	Defines a symbolic Intuition Window structure at <address>*.
Menu	(Menu [<address>])	Defines a symbolic Intuition Menu structure at <address>*.
MenuItem	(MenuItem [<address>])	Defines a symbolic Intuition MenuItem structure at <address>*.
Gadget	(Gadget [<address>])	Defines a symbolic Intuition Gadget structure at <address>*.
IntuiText	(IntuiText [<address>])	Defines a symbolic Intuition IntuiText structure at <address>*.
Image	(Image [<address>])	Defines a symbolic Intuition Image structure at <address>*.
ibase	(ibase)	Displays some important IntuitionBase members and their values.
screen	(screen [<address>])	Formats <address>* as a Screen structure.
window	(window [<address>])	Formats <address>* as a Window structure.
menu	(menu [<address>])	Formats <address>* as a Menu structure.

**If <address> is omitted, the structure is defined at the current address.*

menus	(**menus** [<address>])	Formats <address>* as a linked list of Menu structures (a menu strip).
menuitem	(**menuitem** [<address>])	Formats <address>* as a MenuItem structure.
menuitems	(**menuitems** [<address>])	Formats <address>* as a linked list of MenuItem structures.
gadget	(**gadget** [<address>])	Formats <address>* as a Gadget structure.
gadgets	(**gadgets** [<address>])	Formats <address>* as a linked list of Gadget structures.
itext	(**itext** [<address>])	Formats <address>* as an IntuiText structure.
image	(**image** [<address>])	Formats <address>* as an Image structure.

**If <address> is omitted, the structure is defined or formatted at the current address.*

Chapter 4. Tools

The Tools drawer contains six utilities:

CONTROL	Controls the parameters of a graphic print
ICON2C	Saves the Image data of a Workbench icon to an ASCII file
MERGEMEM	Merges the MemLists of RAM boards
MEMACS	Allows editing of multiple files
PALETTE	Changes the colors of a screen
PRINTFILES	Copies files to the printer

This chapter explains each of these utilities -- except for MEMACS which is covered in Chapter 5.

CONTROL

Format: CONTROL

Template: CONTROL

Purpose: To control the parameters of a graphic print.

Specification:

CONTROL lets you intercept graphic prints sent to your printer so that you can modify the parameters of the print. CONTROL waits for a call to do graphic printing, then it displays a Printer Control window to let you change the variables described below. These variables are the parameters to the DumpRPort command of the printer device.

(See the *Amiga ROM Kernel Manual:Libraries and Devices* for more information on these variables.)

CMap: the Color Map that tells the system which colors to use

Mode: one of eight possible display modes; Control gets the modes from the ViewPort, specifically ViewPort->Modes. The possible modes are:

PFBA	0x40	reverses dual playfield priority
DUALPF	0x400	dual playfield
HIRES	0x8000	high resolution
LACE	4	interlace
HAM	0x800	hold and modify
SPRITES	0x4000	sprites
VP_HIDE	0x2000	ViewPort is obscured by other ViewPorts
EXTRA_HALFBRITE	0x80	extra halfbright

SrcX: the x position at which the drawing area begins

SrcY: the y position at which the drawing area begins

SrcW: the width of the area to be printed (320 in low-resolution mode; 640 for high-resolution mode)

SrcH: the height of the area to be printed (200 in non-interlace mode; 400 in interlace mode)

DCol: describes the size of the area to print to as specified by the Special parameters (below)

DRow: describes the size of the area to print to as specified by the Special parameters (below)

The codes in the rectangular boxes are the flag bits for the io_Special parameter of the DumpRPort command.

Spec:	NP	(NOPRINT)	computes and returns print size; does not print
	TM	(TRUSTME)	does not reset on gfx prints
	NF	(NOFORMFEED)	does not eject paper on gfx prints
	D4	(DENSITY4 bit)	selects density 4 bit (obsolete)
	D2	(DENSITY2 bit)	selects density 2 bit (obsolete)
	D1	(DENSITY1 bit)	selects density 1 bit (obsolete)
	AS	(ASPECT)	ensures correct aspect ratio
	CE	(CENTER)	centers the image on the paper
	FR	(FRACROWS)	DestRows is a fraction of FULLROWS
	FC	(FRACCOLS)	DestCols is a fraction of FULLCOLS
	FR	(FULLROWS)	makes DestRows the maximum size possible
	FC	(FULLCOLS)	makes DestCols the maximum size possible
	MR	(MILROWS)	DestRows is specified in 1/1000 of an inch
	MC	(MILCOLS)	DestCols is specified in 1/1000 of an inch

To run CONTROL from the CLI, type:

```
1> run CONTROL
```

A small box displaying [CLI2] will appear, followed by another CLI prompt, type:

```
>1 GRAPHICDUMP
```

The Printer Control window will appear so that you can modify the variables. After you have set the variables, choose OK, to continue, or CANCEL. Your graphics will now be printed according to the new parameters.

To end CONTROL, type a BREAK command (if you are in the CLI) or Control-C. To run CONTROL from the Workbench, double-click on the CONTROL icon. Then, double-click on the Graphicdump icon.

ICON2C

Format: ICON2C

Template: ICON2C

Purpose: To save the image data of a Workbench icon to an ASCII file.

Specification:

ICON2C saves the image data of a Workbench icon to an ASCII file. The resulting file contains the following C source structures/data:

imageData1	the bit-planes in the icon that comprise the icon
image1	the intuition Image structure for the icon
imageData2	the data for the alternate image displayed when the icon is selected (if any)
image2	the intuition Image structure for the alternate image displayed when the icon is selected (if any)
diskObject	the Workbench DiskObject structure

Double-click on the ICON2C icon, and the ICON2C window appears. The window briefly explains the ICON2C tool, then prompts you to enter the icon name. The name should be entered without the .info suffix. For instance, to copy the image data of the clock icon into an ASCII file, type:

```
Enter icon name without ".info": Workbench:clock
```

Next you will be prompted to enter the filename for the output. Be sure to enter the complete path name for the file. The image data of the Workbench icon is then copied into the specified file.

To run ICON2C from the CLI, type:

```
1> ICON2C
```

The same information that appears in the ICON2C window will appear in the CLI window. Simply enter the icon name and filename as explained above.

You should convert flag values in the resulting C source file to OR'd labels. Fields commented in [] brackets are not traced. The user must supply these if the value is non-zero.

MERGEMEM

Format: MERGEMEM

Template: MERGEMEM

Purpose: To merge the MemLists of RAM boards.

Specification:

MERGEMEM attempts to merge the MemLists of sequentially configured RAM boards that have the same attributes as contiguous expansion RAM. Memory from separate RAM boards is usually kept in separate memory pools. MERGEMEM attempts to merge the separate memory pools into one large pool for allocation of larger contiguous memory blocks by programs. MERGEMEM only merges memory of the same attributes.

To run MERGEMEM from the Workbench, double-click on its icon.

If you run MERGEMEM from the CLI and it is not possible to merge the memory pools, you will receive a message listing the RAM configurations and stating that no merging is possible.

PALETTE

Format: PALETTE [<bitplanes> <screentype>]

Template: PALETTE "BITPLANES,SCREENTYPE"

Purpose: To change the colors of a screen.

Specification:

With PALETTE you can modify the colors of a screen. It differs from the color setting capabilities of Preferences in that Preferences is limited to the colors of the Workbench screen. However, color changes made with PALETTE are only temporary. They cannot be saved to disk.

By specifying values for the <bitplanes> and <screentype> options you can open a custom test screen. The values for <bitplanes> and <screentype> are as follows:

<bitplanes> specifies the depth of the test screen; 1 = 2 colors, 2 = 4 colors, 3 = 8 colors, 4 = 16 colors, and 5 = 32 colors

<screentype> specifies the resolution of the test screen; 0 = 320 x 200 pixels, 1 = 320 x 400 (interlaced screen), 2 = 640 x 200, and 3 = 640 x 400 (interlaced screen)

NOTE: The value for <bitplanes> is restricted to 4 or less if the value for <screentype> is equal to either 2 or 3.

To open PALETTE from the Workbench, double-click on its icon. The Palette Tool window will appear in the frontmost screen, usually your Workbench screen. To modify the colors of a different screen, you need to bring that screen to the front of the display before opening PALETTE. Open the desired screen, then slide it down so that the Workbench screen is also visible. Open the Tools window on the Workbench screen, and double-click on the PALETTE icon. When the Palette Tool window opens, it will be on the front screen.

The Palette Tool window contains several gadgets and sliders. Across the top of the window are color rectangles of the colors that can be modified. Beneath these rectangles are three color sliders: red, blue and green. Select a color rectangle to be modified, then drag the color sliders until you have the color you want. The color changes are reflected in the vertical box that runs alongside the color sliders. Repeat this process with each color rectangle until you have the screen configured as you want it.

Choose the OK box to implement the color change. Choose RESET to return the colors to their original states. Or choose CANCEL to quit PALETTE without making any changes. WARNING: If you open PALETTE on a screen other than the Workbench screen, you must first close PALETTE before exiting or closing the application that opened the new screen.

PRINTFILES

Format: PRINTFILES [-f] <filename> [[-f] <filename>] [[-f] <filename>] ...

Template: PRINTFILES "-f/S,FILENAME/A"

Purpose: To copy files to the printer.

Specification:

With PRINTFILES you can copy files to your printer. PRINTFILES also accepts multiple filenames, so you can designate a string of files to be printed. If PRINTFILES cannot find or open one of the files, it will skip it and go on to the next one.

The -f flag turns on the form feed mode which places a form feed between subsequent files and at the end of the file(s). To specify the form feed mode from the Workbench, use the Workbench menu's INFO item to bring up the PRINTFILES INFO window. Then add "FLAGS=formfeed" to the icon's TOOL TYPES.

To use PRINTFILES from the Workbench:

1) Select the icon of the first file you want to print.

2) Hold down the SHIFT key and select the icons of any additional files you want to print.

3) Hold down the SHIFT key and double-click on the PRINTFILES icon.

5. MEMACS

Format: MEMACS [<filename>] [goto <n>] [OPT W]

Template: MEMACS "FILENAME,GOTO/K,OPT/K"

Purpose: To allow the user to display and edit multiple files.

Specification:

MEMACS (which stands for MicroEmacs and is pronounced M-E-MACS) is a screen-oriented editor that allows you to edit multiple files at one time. The only restriction is that the entire body of each file must be able to fit into memory at one time, since MEMACS performs all of its operations on memory resident text.

The length of the lines you can edit is limited to 80 characters. Characters beyond the 80th character of the line are not lost; they simply do not show on the screen. The only way to see those characters is to break the line or to delete some of the displayed characters. When entering new characters, you can keep typing past the 80th character of a line, but what you type will not show on the screen.

NOTATIONAL CONVENTIONS AND SPECIAL TERMINOLOGY

Throughout this chapter certain key combinations/sequences will be referred to in the following manner:

^(char)	A caret (^) followed by a character is a "Control-key combination." This means that you should hold down the control key as you press the designated character key.
'(char)'	Whenever apostrophes enclose a character, it means that upper or lower case does not matter. It is the keycap itself the selects the function. (This notation is widely used in the summary at the end of this chapter.)
<ESC>	Represents the Esc (Escape) key on your Amiga keyboard.
<DEL>	Represents the Del (Delete) key on your Amiga keyboard.
<TAB>	Represents the Tab key on your Amiga keyboard.
<RETURN>	Represents the Return key on your Amiga keyboard.

There are some special terms associated with MEMACS that you should be familiar with:

Buffer: A memory area that MEMACS controls. There is always at least one buffer used by MEMACS, and it will contain zero or more characters of text.

Dot: The current cursor position.

Mark: A cursor position that you can specify. (Each buffer has its own dot and mark.) The menu item **Set-mark** allows you to "mark" the current cursor position. You can then move forward or backward in the file, adding or deleting text. Then, when you wish to return to the place that you "marked", you simply select the **Swap-dot&mark** command.

You can also set a mark to indicate the beginning of a block of text that you want to duplicate, move, or delete. The "block" will encompass all the characters starting with the mark and continuing to the current cursor position.

Kill: Kill commands remove text from the screen and save it in a kill buffer. This text can be retrieved and put back into your document by using the **Yank** command. As you issue successive Kill commands (without selecting Yank in between), each block of text that you kill will be added to the existing text in the kill buffer.

Yank: The Yank command copies the contents of the current kill buffer to the line just above the one in which the cursor is positioned. You can copy a block of text from one buffer to another by killing that block, then, without moving the cursor, immediately Yanking it back into the same buffer. Move the cursor to a new position, and choose Yank again. The text you want to copy will still be in the kill buffer and will remain there until you mark and kill another block of text.

Window: A MEMACS window is somewhat different than an Intuition window in Workbench. In MEMACS, the screen can be split into multiple slices so that you can edit and display more than one buffer or two or more portions of the same buffer. Each "slice" is a MEMACS window.

Read a File: When you ask MEMACS to read a file (using the menu command **Read-file**), the contents of the current buffer are replaced with the contents of the file you want to read.

Visit a File: If you want to access a new file without replacing the contents of the current buffer, you can ask MEMACS to visit a file (using the **Visit-file** command). MEMACS assigns a new buffer to the file you are visiting.

Select Buffer: You can switch back and forth between the buffers you are working with by choosing the **Select-buffer** command and specifying the name of the buffer you wish to use. MEMACS sometimes assigns a shorter name to a buffer than the file name to which it corresponds. Be careful to specify the correct buffer name when you want to switch back and forth.

Modified Buffers: When you make any changes to a buffer, even if you only hit <RETURN> then delete it, MEMACS remembers and will mark that buffer as a modified buffer.

You can see which buffers have been modified by using the **List-buffers** command. Any modified buffers are signified with an asterisk (*). If you try to exit MEMACS without saving any changes, a prompt will tell you that modified buffers exist and will ask you if you really want to quit. Once you save a buffer, the modified status is removed.

OPENING MEMACS

When you open MEMACS, a new screen appears. At the bottom of this screen are the words "MicroEMACS -- main". This line displays the name of the buffer that is currently in use. In this case, it is the "main" buffer. Remember, a buffer contains zero or more characters of text.

Usually you will invoke MEMACS with a filename specified. If the file exists, it will be read into a buffer. Otherwise, the file will be created when you save your work.

You can have several buffers in use at one time, and you can show one or more on the screen at the same time. Menu options let you switch back and forth between them. At all times, what you see on screen is what is actually in the buffer.

If the contents of a buffer have been either read from or written to a file, that buffer will be associated with that file. In this case, the bottom line of the screen will display the name of the buffer along with the file name with which it is associated.

MEMACS has two modes of operations: normal and command. When MEMACS is in normal mode, you can:

- *move the cursor using the cursor keys
- *move the cursor to the edge of the window by holding down the SHIFT key and pressing the appropriate cursor key
- *move the cursor by clicking the left mouse button in the desired place on the screen
- *insert characters at the current cursor position simply by typing them
- *delete the character at the current cursor position by pressing <DEL>
- *delete the character to the left of the cursor by pressing <BACKSPACE>
- *perform other special functions as explained in the menu section and command summaries that follow

When MEMACS is in command mode, the cursor jumps to the bottommost line of the display, and the program asks you for certain additional information. The command mode is entered through various menu items which are explained later.

MOUSE COMMANDS

You can also use the mouse to interface with MEMACS. If the MEMACS window is inactive, clicking the left mouse button will activate the window. You can then move the cursor by moving the mouse's pointer to the spot where you want the cursor to be and clicking the left mouse button. You can also use the mouse to switch between buffers. However, to do this both buffers must be visible on the screen.

MEMACS MENUS

MEMACS offers the following main menu items:

Project	--	system and file-oriented items
Edit	--	file editing commands
Window	--	controls the characteristics of the MEMACS windows
Move	--	controls the placement of the cursor
Line	--	line-oriented operations
Word	--	word-oriented operations
Search	--	search and search/replace options
Extras	--	controls the numerical value of arguments, and lets you execute a series of operations as though it was a single special command

This section will explain each of these menus and their commands. Each of the commands also has a keyboard shortcut. The shortcuts appear in the menus to the right of the command and in this text along the right-hand margin.

The Project Menu

The commands in the Project menu, except for Visit-file, affect the buffer associated with the current cursor position.

Rename ^XF

Changes the name of the file associated with the current buffer. This command is useful if you are saving versions of a program or text file as you go along. You can perform a Save command for the first version, modify a few things, Rename the file associated with this buffer, and then save the new version.

When you select Rename, MEMACS prompts:

```
New file name:
```

If you simply press <RETURN> without specifying a file name, the buffer becomes disassociated with any file name. You must specify a name here if you want the buffer to be appropriately associated with a file.

Read-file ^X^R

Replaces the contents of the current buffer with the contents of a file. When you select Read-file, MEMACS moves the cursor to the bottom line of the display and requests:

```
Read File:
```

Enter the complete path of the file, including the volume name, directory, and file, then press <RETURN>. The file is read into the current buffer, overwriting the data that was stored there.

If you do not want to read a file, simply press <RETURN> without specifying a file name. MEMACS will ignore the request and return you to normal mode.

Visit-file ^X^V

Lets you work with additional files, aside from the first file you open. You must already be editing something before you can visit another file. This command is useful for programmers who are creating a program and want to extract pieces from or refer to other programs.

When you issue this command, MEMACS moves the cursor to the bottom line and asks:

Visit File:

Type the complete path of the file, and hit <RETURN>. MEMACS will read the file into a buffer, if it is not already there. If the file you want to visit is on a different disk, AmigaDOS will display a requester asking you to insert that particular disk into any drive. If the file is already in a buffer, MEMACS will switch you to that buffer automatically.

Insert-file ^X^I

Inserts the contents of a file into the current buffer. When you issue this command, MEMACS moves the cursor to the bottom line and asks:

Insert File:

Enter the complete path name of the file to use, and hit <RETURN>. MEMACS will read it into the current buffer at a point one line above the current cursor position.

Save-file ^X^S

Writes the contents of the current buffer to the file name associated with that buffer. The file name associated with the buffer was determined when the contents of an existing file were read to the file (Read-file) or when the file associated with the current buffer was renamed (Rename).

If there is no file name specified on the status line, MEMACS tells you "No File Name" and refuses to perform the save.

After a successful Save, MEMACS uses the bottom line of the screen to tell you how many lines it has written out to the designated file name.

Save-file-as **^X^W**

Allows you to specify the name of a file to associate with a buffer. When you issue this command, MEMACS prompts:

Write File:

MEMACS is requesting the name of the file in which it should save the current contents of the buffer. If you provide a complete path and press <RETURN>, the buffer will be written out to that file. (If you don't provide a name and press <RETURN>, you are returned to normal mode.) On the status line for the buffer, the following notation will appear:

File: <filename>

From now on, that file will be used to save the current contents of this buffer when you issue a Save command.

Save-mod **^X^M**

Writes the contents of all modified buffers to the disk. Use this item with caution to be sure that you don't accidentally modify a buffer associated with a file you have visited but don't intend to change.

Save-exit **^X^F**

Saves all modified buffers then exits MEMACS. It is simply a combination of the Save and Quit items. Again, use this item with caution (see **Save-mod**).

New-Cli **^-**

Brings up an entirely new CLI window called "Spawn Window". You can issue as many CLI commands in the spawn window as you want without interfering with MEMACS. To return to MEMACS, use the ENDCLI command. The spawn window disappears, and MEMACS is restored to its previous state.

Cli-Command **^X!**

Allows you to execute an AmigaDOS command while you are still in MEMACS. It is similar to issuing a RUN command while in the CLI.

When you select this menu item, MEMACS moves the cursor to the bottom of the screen and provides you with a prompt (!). You can then type a command for AmigaDOS to process on this line. MEMACS temporarily suspends operation, and AmigaDOS executes your command. The output of the command appears in a temporary buffer called spawn.output.

Quit **^C**

Exits MEMACS. If one or more of the buffers has been modified since you last saved it to a file, MEMACS prompts:

Modified buffers exist, do you really want to exit? [y/n]?

MEMACS is giving you a last chance to save your work. If you don't want to exit, simply press <RETURN>. If you do want to quit, you must press the 'y' key then hit <RETURN>.

Before quitting, you can check which buffers MEMACS is referring to by selecting List-buffers in the Edit menu. MEMACS lists the names associated with each buffer and shows an asterisk by each buffer that has been modified since you last saved it to disk.

There are circumstances under which you will not want to save all buffers back to the original files. For example, let's say you were writing a program and copying pieces from other existing programs as you went along. Some of the files you visited may have been accidentally modified or may have been visited on a write-protected disk.

If you are simply using an old program as temporary source material, you will not want to destroy the original program. When you are finished writing the new program, save your new material and exit MEMACS without saving the modified buffers of the source program.

Two alternate keyboard commands for the Quit command are ^X^C and <ESC>^C.

The Edit Menu

The commands in the Edit menu affect the editing of your buffers and their associated files.

Kill-region ^W

Deletes blocks of text from the current buffer and saves it in a kill buffer, a special buffer for text that has been deleted from buffers by using Kill commands. (Text can be pulled back into the document by using the **Yank** command, described below.)

If a block of text has been "marked" using the Set-mark command (explained below) and the cursor has been positioned away from the mark, the area between those two points is considered a **block** and can be deleted by selecting Kill-region.

You can also use Kill-region to copy a block from one section of the buffer to another. Simply mark the block, select Kill-region, then **without moving the cursor**, immediately select Yank. The block will be restored to its original position, but there will also be a copy of the block in the kill buffer.

If you repeatedly select Kill-region on different areas of text, without performing a Yank, each successive kill segment is **appended** to the kill buffer. When you perform the first Yank, it marks the end of the kill buffer.

Yank ^Y

Copies the contents of the kill buffer to the line immediately above the current cursor location in the current buffer. Yank reverses the action of Kill-region, but it does not change the contents of the kill buffer. Therefore, you can repeatedly move the cursor to another buffer, select Yank, and copy the contents of the kill buffer. The next time you kill a block of text, however, the contents of the kill buffer will be replaced with the new material, and the old contents will be lost.

Kill-region and Yank are often used together to move text from one buffer to another.

Set-mark **^@**

Marks the cursor position in a buffer. When you select Set-mark, the position of the cursor is marked in the current buffer. From then on, any other position of the cursor is referred to as the dot. You can move back and forth between the mark and the dot by selecting the Swap-dot&mark command in the Move menu.

You can use Set-mark to mark the beginning of a block of text that you want to duplicate or move somewhere else in the buffer. Set the mark on the first character you want to include in the block. As you move the cursor through the file, you are essentially blocking out a portion of text.

An alternate keyboard shortcut for Set Mark is **<ESC>-**.

Copy-region **<ESC>w**

Copies the contents of the marked region to the kill buffer. This new text replaces any previous contents of the kill buffer.

Upper-region **^X^U**

Changes the text of the entire marked region, the area between the mark and the current cursor position (dot), to upper case.

Lower-region **^X^L**

Changes the text of the entire marked region to lower case.

List-buffers **^X^B**

Splits the current buffer's window and provides you with a list of the buffers that MEMACS is currently maintaining. The list has 4 columns. For example:

C	Size	Buffer	File
*	17260	emacs.doc	df1:docfiles/emacs.doc

C is an abbreviation of "Changed" and will display an asterisk if the buffer has been modified since it was last saved to a file.

Size shows how many characters there are in a particular buffer.

Buffer shows the name given to this particular buffer. If you have read in a file, this will usually be the name of the file itself, minus the full path. For example, if the file you are editing is df1:docfiles/emacs.doc, then its buffer name will be emacs.doc.

File shows the name of the file, including the full path. This shows you where MEMACS will write this file if you choose Save-file or Save-exit while your cursor is in that buffer.

When you choose List-buffers, the status line at the bottom of the screen displays "MEMACS -- [List]". Even though List-buffers brings up a window display, it is not listed as an available buffer. If you edit the List-buffers window, it can be made to act just like any other buffer. If, for example, you open a file in the List-buffers window, the name of the buffer will continue to be [List], and the name of the file you have opened will become associated with the List-buffers window.

If you should leave the List-buffers window on the screen but use a different window to modify the listed buffers, the List-buffers display will not be continuously changed to reflect the current changes. To get current information, you must select List-buffers again.

Select-buffer **^Xb**

Lets you select which buffer you wish to edit in the currently selected window, the window where your cursor is positioned. When you choose Select-buffer, MEMACS moves the cursor to the bottom line and asks:

Use buffer:

You must provide a name that is the same as one of those shown in the List-buffers listing. If you specify one of the available names, that buffer replaces the contents of the currently selected window.

If you specify a name that is not in the List-buffers listing, you are telling MEMACS to create a new buffer with that name. In this case, there is no file name associated with the new buffer and you will have to rename the file or select Save-as-file when you are prepared to save the buffer's contents to a file.

If you simply press <RETURN>, the command is ignored.

Insert-buffer **<ESC>^Y**

Inserts the contents of a named buffer into the current buffer at the line above the current cursor position. When you select Insert-buffer, MEMACS asks:

Insert buffer:

You must type the name of the buffer to insert, then press <RETURN>.

Kill-buffer **^Xk**

Deletes the contents of a chosen buffer. MEMACS can only edit a file if the entire file fits in the available memory. To make room in the system's memory, you can use Kill-buffer to delete the contents of one or more buffers. This command returns the buffer's memory to the memory manager for reuse.

When you choose Kill-buffer, MEMACS asks:

Kill buffer:

You must then enter the name of the buffer you wish to delete. You cannot kill a buffer if its contents are currently displayed.

Justify-buffer **^XJ**

Removes all blank spaces and tabs from the left-hand edge of all the lines in the current buffer. The text is rearranged so that it aligns with the current margins.

Redisplay **^L**

Causes a complete redrawing of the entire screen.

Quote-char **^Q**

Lets you "Quote" a character and make it part of the text file. Some keyboard selections have been assigned as MEMACS control characters (for instance, the menu command shortcuts). If you try to insert such a selection into your text, MEMACS will react as if you you had chosen a menu item.

For example, Control-L (^L) tells MEMACS to redraw the display, but ^L is also useful as a printing control to insert a formfeed character. By selecting Quote-char, the next character you type will be taken "literally" by MEMACS and will be inserted into the text file, instead of being treated as a menu command. You can also use Quote-char to insert a <RETURN> key into the text or to insert any other control character that may be needed during a macro command. Even ^Q can be inserted by typing it twice.

To designate a Control-key combination, use the caret (^) followed by the character, as used in this manual. As MEMACS manipulates the buffer, the combination of the caret and the character is treated as a single character, both by the cursor keys and the character counter. An alternate keyboard shortcut for Quote char is **^Xq**.

Indent **^J**

Moves the cursor to the next line, automatically indenting the same amount of spaces as the previous line.

Transpose **^T**

Swaps the positions of two adjacent characters. Place the cursor on the rightmost of the two characters.

Cancel **^G**

Ends an ongoing menu command, such as a query search and replace.

The Window Menu

A window in MEMACS is not the same as a window in Intuition (the Workbench's display system). MEMACS splits the screen into multiple slices, allowing you to edit a separate file (buffer) in each MEMACS window. The Window menu lets you control how you view your buffers on the screen.

One-window **^X1**

Makes the current buffer a single, full-sized window on the MEMACS screen. All other buffers remain invisible, allowing you maximum space to work on the current buffer.

Split-window **^X2**

Splits the current window in half, positioning the current buffer identically in both windows. This lets you edit two segments of the buffer at the same time. Any changes made in either window affect the entire buffer. This is convenient when you want to see what you wrote in an earlier part of your document while working on a later section.

Next-window **^Xn**

Moves the cursor "down" to the next window and makes that window available for editing.

Prev-window **^Xp**

Moves the cursor "up" to the next window and makes that window available for editing.

The Next-window and Prev-window commands wrap-around. If you move the cursor down as far as it will go, the cursor will automatically move up to the topmost window. Selecting Prev-window when the cursor is in the topmost window will move the cursor to the bottom-most window.

Expand-window **^Xz**

Adds a line to the current window and simultaneously deletes a line from the adjacent window.

Shrink-window **^X^Z**

Deletes a line from the current window and simultaneously adds a line to the adjacent window.

Next-w-page **<ESC>^V**

Displays the next page of the next window. For instance, if you have split a window and are working in the top one, selecting Next-w-page will move the contents of the bottom window (the one you are not working in) to the next page. This does not make the window available for editing; it just lets you view the contents.

Prev-w-page **^Xv**

Displays the next page of the previous window. If only one window is displayed, it displays the next page of that window.

The Move Menu

The commands in the Move menu let you move the cursor rapidly through the current buffer.

Top-of-buffer **<ESC><**

Moves the cursor to the top line of the current buffer.

End-of-buffer **<ESC>>**

Moves the cursor to the bottom line of the current buffer.

Top-of-window **<ESC>,**

Moves the cursor to the top of the current window.

End-of-window **<ESC>.**

Moves the cursor to the bottom of the current window.

Goto-line **^X^G**

Moves the cursor to a specific line number. When you select Goto-line, MEMACS moves the cursor to the bottom of the screen and asks:

```
goto-line:
```

Enter a line number, press <RETURN>, and MEMACS moves the cursor directly to that line. If you specify a line number larger than the total number of lines in the buffer, MEMACS moves the cursor to the last line of the buffer.

Swap-dot&mark **^X^X**

Places a mark at the current cursor position and moves the cursor to where the mark had been set. If you have not yet set a mark in the window, MEMACS replies, "No mark in this window." This command lets you move quickly to and from a preset location in your buffer. Selecting this item again, restores the cursor to where it was before you selected Swap-dot&mark the first time.

Next-page **^V**

Moves the text within the window toward the end of the buffer by one full window, less one line. The cursor is repositioned so as to stay on the screen.

Prev-page **<ESC>v**

Moves the text within the window toward the beginning of the buffer by one full window, less one line. The cursor is repositioned so as to stay on the screen.

Next-word **<ESC>f**

Moves the cursor forward to the next non-alphanumeric character after the current word.

Previous-word **<ESC>b**

Moves the cursor back to the first letter of the previous word.

Scroll-up **^Z**

Moves the text within the window towards the end of the buffer by a single line.

Scroll-down **<ESC>z**

Moves the text within the window towards the beginning of the buffer by a single line.

The Line Menu

The commands in the Line menu let you move the cursor within or between lines and let you perform operations involving entire lines.

Open-line ^O

Splits the line the cursor is in, forcing the character at the cursor rests to become the first character of the following line. This command leaves the cursor in the original line so that you can type new characters beginning at the current cursor position.

If you select Open-line by mistake, immediately pressing the <DEL> key will close up the line.

Kill-line ^X^D

Deletes the line in which the cursor is located and places the text in the kill buffer. If you have not selected Yank since the last Kill command, the text will be appended to the existing text in the kill buffer.

Kill-to-eol ^K

Deletes the text between the current cursor position and the end of the line. This text enters the kill buffer and will be appended to the existing text if a Yank has not been recently performed. The text can be restored to your file by immediately selecting Yank.

Start-of-line ^A

Moves the cursor to the leftmost position on a line.

End-of-line ^E

Moves the cursor to the rightmost position on a line. If you have typed more than 80 characters on a line, a dollar sign ($) appears at the right hand edge of the line. Moving to the end of the line places the cursor logically on the rightmost character even though you cannot see it. Physically the cursor is positioned over the dollar sign. If you use the left cursor key to move the cursor to the left, it will take as many key presses as there are unseen characters before the cursor actually begins to move.

Next-line **^N**

Moves the cursor down one line.

Previous-line **^P**

Moves the cursor up one line.

Line-to-top **<ESC>!**

Moves the line containing the cursor to the top of the window.

Delete-blanks **^X^O**

Deletes blank lines, proceeding forward from the current cursor position, until MEMACS gets to the next line on which text exists.

Show-Line# **^X=**

Displays information on the present cursor position. For example:

```
Line 17 Column 1 (2%)
```

In this example, the cursor is on the 17th line of text, in the first column. The percentage shows that the cursor is in a position 2% of the way from the top of the buffer. In other words, if the cursor was on the last character of text, the percentage would be equal to 100.

The Word Menu

The Word menu contains word-associated operations.

delete-forw **<ESC>d**

Deletes the character on which the cursor is positioned and all remaining characters to the right until the next non-alphanumeric character is found, (i.e. a blank space, tab, or punctuation mark).

For instance, if the cursor is positioned on the "s" in the word wordsuffix, choosing delete-forw will delete "suffix" from the word. If the cursor is positioned on a blank space, it must be moved forward to the start of a word to delete that word.

delete-back **<ESC>h**

Deletes the character at the cursor and all remaining characters to the left of the cursor until it finds the first character of a word.

An alternate form of this command is **<ESC><DEL>**.

Upper-word **<ESC>u**

Changes a word to upper case, starting at the character at the cursor and proceeding to the last character of the word.

Lower-word **<ESC>l**

Changes a word to lower case, starting at the character at the cursor and proceeding to the last character of the word.

Cap-word **<ESC>c**

Changes the character where the cursor is positioned to upper case. It also changes the characters to the right of the cursor, through to the end of the word, to lower case.

Switch-case **<ESC>^**

Changes the case of a word, starting at the current cursor position and proceeding to the right until it reaches the end of the word. If a word is upper case it changes it to lower case, and vice versa.

The Search Menu

The Search menu allows you to search through the current buffer for specific text strings. The case (upper or lower) of the string is not significant in the search itself. However, if you are using a text substitution (search and replace), the text will be replaced in the same case as that of the replacement string.

Search-forward ^S

Searches through the text starting at the cursor and moving forward to the end of the buffer. When you issue this command, MEMACS moves the cursor to the bottom line of the screen and asks:

```
Search:
```

Enter the string of characters that you want MEMACS to search for, and press <RETURN>. If the string is found, MEMACS positions the cursor immediately following the last character of the string.

If MEMACS cannot find the string, it replies "Not found."

An alternate form of this command is ^Xs.

Search-backward ^R

Searches through the text from the cursor backwards to the beginning of the buffer. This command operates in the same manner as Search-forward.

An alternate form of this command is ^Xr.

Search-replace **<ESC>r**

Operates the same way as Search-forward, except that it allows you to replace the string with different text. When MEMACS finds the first occurrence of a specified string, it asks:

Replace:

You must enter the string of characters that should replace the found string. Remember, the characters will appear in the same case as you type them. When you press <RETURN>, MEMACS will automatically forward-search the rest of the file and replace the search-string with the replacement-string. After MEMACS completes this command, it reports:

Replaced (xx) occurrences

(xx) stands for the number of times the string was replaced.

Query-s-r **<ESC>q**

Operates the same way as Search-replace, except that it allows you to choose whether or not to replace each occurrence of the string. When you select Query-s-r, MEMACS prompts:

Query replace:

As it finds a matching string, it always asks:

Change string? [y/n/c/^G]?

The options are: y (yes); n (no); c (changes all occurrences of the string); and ^G (abort). This gives you a chance to control the replacement process. After MEMACS completes this command, it reports:

Replaced (xx) occurrences

Fence-match **<ESC>^F**

Finds the next occurrence of a character that is the same as the one at the current cursor position. For instance, if the cursor is resting on an asterisk (*), choosing Fence-match will move the cursor to the next occurrence of an asterisk in the text.

The Extras Menu

The Extras menu contains commands to let you tell MEMACS how to operate. Many of these operational commands require that you specify a numeric argument before selecting the command itself.

This menu also includes several macro commands. A macro command is actually a sequence of commands or other keystrokes that are executed by selecting the menu item **Execute-macro**.

Set-arg ^U

Lets you specify a numeric argument for the operational commands. When you issue this command, MEMACS responds by moving to the bottom line and prompting:

```
Arg: 4_
```

If you select Set-arg again, MEMACS multiplies the argument value by 4.

If you press a numeric key (0-9), MEMACS accepts an integer argument. If you press a minus sign first, MEMACS accepts a negative integer argument, starting at -1.

Examples: (Each started by a single press of ^U)

```
Arg: -1          (pressed "-" as the first key)
Arg: -23         (pressed "- 2 3" as a 3-key sequence)
Arg: 12          (pressed "3 ^U" as a 2-key sequence)
```

MEMACS accepts the argument value as a key for whatever you do next. To add 12 blank lines at the cursor position, specify an argument of 12, then press <RETURN>. To add 20 minus signs, select an argument number of 20, do **not press <RETURN>**, and press the minus sign on the keyboard. (Note: Don't use the keypad's minus sign; it is mapped to a different value.)

To set one of the MEMACS operational parameters (described below), select the value of the argument, **do not press <RETURN>**, then select the appropriate menu item. MEMACS will use the argument to set the value.

Set **<ESC>s**

Allows you to choose various MEMACS parameters. When you choose Set, MEMACS prompts:

Set:

You can then enter one of the following:

Screen	places the MEMACS display in a Workbench window or back onto a custom screen
Interlace	turns the interlace mode on or off
Mode	results in a second prompt "Mode:"; you can enter cmode (for editing c programs) or wrap* (to enable automatic wordwrap when the text reaches a set cursor position). Cmode provides automatic fence matching. Use +mode or -mode to add or subtract a mode.
Left*	determines the left margin
Right*	determines the right margin
Tab*	sets the increment for tab spacing
Indent*	used in cmode to determine how far to indent each level of nesting
Case	turns case sensitive searches on or off; default is off
Backup	turns on or off MEMACS' backup function. Your options are ON (renames the current file <filename>.bak and saves that backup file to the T: directory, SAFE (this option checks to see if a file already exists for the buffer; if so, it will not overwrite the existing file), and OFF (this is the default option; MEMACS does not perform any backup)

*Each of these entries results in a prompt for a numerical argument, unless the numeric argument is given along with the entry.

Start-macro **^X(**

Tells MEMACS to start recording any subsequent keystrokes or menu selections. This is a macro command and is used in conjunction with the Stop-macro and Execute-macro commands.

Stop-macro **^X)**

Tells MEMACS to stop recording keystrokes.

Execute-macro **^Xe**

Repeats keystrokes and menu selections that were entered between Start-macro and Stop-macro. They are repeated as if you had freshly entered the entire sequence.

Set-key **^X^K**

Allows you to redefine all of the function keys, the Shifted function keys, the Help key, or any key on the numeric keypad as keyboard macros. This means that if you select one of these redefined keys while recording macro commands, the new key definition will be recorded in the command. One definition having as many as 80 keystrokes can be recorded for each of these keys.

NOTE: If you want to insert the Set-mark command into any of the keyboard macro definitions, you cannott use the menu shortcut of ^@. This does not function correctly when used in a macro command. Instead, you must use the alternate form of Set-mark, <ESC>-. This alternate form is acceptable in macro commands.

When you choose Set-key, MEMACS asks:

key to define:

Press one of the 10 function keys, the Help key, or a numeric keypad key. MEMACS responds:

def: [commands]:

[commands] is a display of the current commands bound to that key. Enter the new string of characters (up to 80) that you want to have MEMACS respond to when this key is pressed. Hitting <RETURN> terminates the entry.

Remember that when entering commands involving function keys, for example <ESC>< (go to top of buffer), you must use Quote character [^Q] to properly insert the function key keystroke into the definition.

The chart below contains the default values of the function keys when used in macro commands:

KEY	DEFAULT VALUE	KEY SEQUENCE
fkey 1	clone line	^A^K^Y^M^Y
fkey 2	delete line	^X^D
fkey 3	execute keyboard macro	^Xe
fkey 4	next screen	^V
fkey 5	previous screen	<ESC>v

fkey 6	split window	^X2
fkey 7	one window	^X1
fkey 8	scroll window up	^Z
fkey 9	scroll window down	<ESC>Z
fkey 10	save file and exit	^X^F
help	insert line and indent	^J
keypad enter	insert line and indent	^J

The numeric, period, and minus keys on the numeric keypad default to their normal values (i.e. keypad 1 defaults to 1, keypad 2 defaults to 2, etc.).

Reset-keys **<ESC>k**

Returns any keys defined by Set-keys to their original default state.

Execute-file **<ESC>e**

Allows you to execute a program file within MEMACS. When you select this command, MEMACS prompts:

File:

Enter the name of the file you wish to access. This file is executed as a file of MEMACS commands.

Execute-line **^[^[**

Sets MEMACS to the command mode. When you choose Execute-line, MEMACS asks:

execute-line:

You can then enter any menu command and its parameters by simply typing it at the prompt. You must use the exact format used in the menus, including hyphens, or you will receive an alert and "command error" message. For instance, you cannot type:

execute-line: insert file <filename>

You must type:

execute-line: insert-file <filename>

An alternate shortcut for execute-line is **<ESC><ESC>**.

COMMANDS NOT INSTALLED IN MENUS

The following commands have not been installed in menus and are only accessible through the keyboard.

Describe Key **<ESC>^D**

Tells you if any functions are bound to a key or key-sequence. When you select <ESC>^D, MEMACS prompts for the key to describe. If you enter a key sequence, such as ^L or <ESC>k, MEMACS will respond with the corresponding function. In this case, Redisplay and Reset-keys, respectively.

Bind Key **<ESC>^B**

Allows you to bind a key to a function. When MEMACS prompts for the key to bind, enter the function (following the format used in the menu items) then the key or key sequence. To check if the key was bound properly, use Describe key (<ESC>^D).

Unbind Key **<ESC>^U**

Allows you to return a bound key to an unbound state. When MEMACS prompts for the key to unbind, enter the key or key sequence. MEMACS will then reply "Key is not bound."

Echo **<ESC>^E**

Displays the string typed in the command line. This command is usually used when creating or editing executable MEMACS script files.

Move to Edge of Window **<SHIFT> + Cursor Key**

By holding down the Shift key and a cursor arrow key, MEMACS will move to the top, bottom, left, or right edge of the screen. This is subject to the amount of text available.

Delete the Next Character **^D**

Deletes the character at the current cursor position. This is the same as hitting the <DELETE> key.

Delete the Previous Character **^H**

Deletes the character to the left of the current cursor position. This is the same as hitting the <BACKSPACE> key.

Move to Next Line **^M**

Inserts a newline character after the current cursor position and moves the cursor to the start of the new line.

Move Cursor by x number of Characters **^F (forward)**
^B (backward)

Allows you to move the cursor forward or backward a specified number of spaces. The default value of this command is one character. However, you can establish a higher value by using ^U to set the argument value. Then select ^F or ^B to move that number of characters.

ADDING MEMACS STARTUP COMMANDS

When MEMACS is opened, it reads the contents of a file called emacs_pro to see if there are any commands that it should automatically execute. Emacs_pro does not exist; you have to create it. MEMACS first looks in the current directory for emacs_pro. If it is not there, it looks in the disk's s:emacs directory.

You can create more than one emacs_pro file if you wish. You can create a global file that would execute a series of commands each time MEMACS is opened. And, you can create more specified local files with startup commands particular to a certain file that you may use frequently. In the case of both local and global emacs_pro files, the local startup command files override the global file.

FUNCTIONAL LIST OF COMMANDS

Operations:

Replace buffer with new file (Read-file)	^X^R
Open an additional file (Visit-file)	^X^V
Insert a file into current buffer (Insert-file)	^X^I
Rename the buffer	^XF
Display a list of buffers	^X^B
Select a buffer to edit	^Xb
Insert contents of buffer into current buffer	<ESC>^Y
Write buffer to new file (Save-as-file)	^X^W
Save buffer to existing buffer	^X^S
Save all modified buffers	^X^M
Save all modified buffers and exit	^X^F
Set numeric argument value	^U
Set MEMACS parameters	<ESC>s
Show the current cursor position (Show-line#)	^X=
Run one CLI command	^X!
Open a CLI window	^-
Cancel a menu command	^G
Quit MEMACS	^C
	^X^C
	<ESC>^C

Moving Cursor:

Swap dot and mark	^X^X
Move forward x number of characters	^F
Move backward x number of characters	^B
Move to start of line	^A
Move to end of line	^E
Go to a specified line	^X^G
Move to next line	^N
Move to previous line	^P
Move to next page	^V
Move to previous page	<ESC>v
Move to start of buffer	<ESC><
Move to end of buffer	<ESC>>
Move to next word	<ESC>f
Move to previous word	<ESC>b

Windows:

Make current buffer a full-sized window (One-Window)	^X1
Split current window	^X2
Expand window	^Xz
Shrink window	^X^Z
Move cursor to top of window	<ESC>,
Move cursor to end of window	<ESC>.
Move current line to top of window	<ESC>!
Scroll window up one line	^Z
Scroll window down one line	<ESC>z
Move cursor to next window	^Xn
Move cursor to previous window	^Xp
Redraw the screen	^L

Text:

Set a mark in the text	^@
	<ESC>-
Quote a character	^Q
	^Xq
Justify text in buffer	^XJ
Transpose characters	^T
Make a word upper case	<ESC>u
Make a word lower case	<ESC>l
Make a region upper case	^X^U
Make a region lower case	^X^L
Change the case of a word	<ESC>^
Capitalize a word's first letter	<ESC>c
Insert a tab	^I
	<TAB>
Insert line, cursor is moved to next line	^M
	<RETURN>
Insert line, cursor is moved to next line and indented same number of spaces as previous line	^J
Split the line (Open-line)	^O

Delete and Copy:

Delete next character	^D
	<DEL>
Delete previous character	^H
	<BACKSPACE>
Delete next word	<ESC>d
Delete previous word	<ESC>h
	<ESC><DEL>
Delete text from cursor to end of line	^K
Delete entire line	^X^D
Delete region between dot and mark	^W
Delete blank lines	^X^O
Delete entire buffer	^Xk
Copy contents of marked region into kill buffer	<ESC>w
Copy contents of kill buffer into current buffer (YANK)	^Y

Search:

Search forward	^S
	^Xs
Search backward	^R
	^Xr
Search forward and replace	<ESC>r
Search forward, query and replace	<ESC>q
Fence match	<ESC>^F

Macro Commands:

Start macro commands	^X(
Stop macro commands	^X)
Execute macro commands	^Xe
Execute line	^[^[
	<ESC><ESC>
Execute file	<ESC>e
Define function keys (Set-key)	^X^K
Reset keys	<ESC>k
Describe key	<ESC>^D
Bind key	<ESC>^B
Unbind key	<ESC>^U
Echo string	<ESC>^E

ALPHABETICAL LIST OF COMMANDS

Remember: When a key is shown enclosed in apostrophes it means that the case of the key does not matter; it is the keycap itself which controls the function.

Control-Key Combinations

Set-mark	^'@'
New-CLI	^'-'
Execute-line	^[^[
Start-of-line	^'A'
Move backward x number of characters	^'B'
Quit MEMACS	^'C'
Delete next character	^'D'
Move to end of line	^'E'
Move forward x number of characters	^'F'
Cancel a menu command	^'G'
Delete previous character	^'H'
Insert a tab	^'I'
Insert line, cursor is moved to next line and indented same number of spaces as previous line	^'J'
Kill-to-eol	^'K'
Redisplay	^'L'
Insert line, cursor is moved to next line	^'M'
Next-line	^'N'
Open-line	^'O'
Previous-line	^'P'
Quote a character	^'Q'
Search-backward	^'R'
Search-forward	^'S'
Transpose	^'T'
Set argument value	^'U'
Next-page	^'V'
Kill-region	^'W'
Yank	^'Y'
Scroll window up	^'Z'

**Used in conjunction with Set-arg (^U)*

Control-X/Control-Key Combinations

List-buffers	^X^B
Quit MEMACS	^X^C
Kill-line	^X^D
Save-exit	^X^F
Goto-line	^X^G
Insert-file	^X^I
Define function keys (Set-key)	^X^K
Make a region lower case	^X^L
Save all modified buffers	^X^M
Delete-blanks	^X^O
Read-file	^X^R
Save-file	^X^S
Make a region upper case	^X^U
Visit-file	^X^V
Save-file-as	^X^W
Swap-dot&mark	^X^X
Shrink-window	^X^Z

Control-X/Key Combinations

CLI-Command	^X!
Start macro commands	^X(
Stop macro commands	^X)
Show-Line#	^X=
One-window	^X1
Split-window	^X2
Select-buffer	^X'b'
Execute-macro	^X'e'
Rename buffer	^X'F'
Justify text	^X'J'
Kill-buffer	^X'k'
Next-window	^X'n'
Previous-window	^X'p'
Quote character	^X'q'
Search-backward	^X'r'
Search-forward	^X's'
Next page of previous window	^X'v'
Expand-window	^X'z'

Escape-Key Combinations

Line-to-top of window	<ESC>!
Switch-case	<ESC>^
Set mark	<ESC>-
Move to top of window	<ESC>,
Move to end of window	<ESC>.
Top-of-buffer	<ESC><
End-of-buffer	<ESC>>
Bind key	<ESC>^B
Previous-word	<ESC>b
Quit MEMACS	<ESC>^C
Capitalize a word	<ESC>c
Describe key	<ESC>^D
Delete next word	<ESC>d
Echo	<ESC>^E
Execute-file	<ESC>e
Fence-match	<ESC>^F
Next-word	<ESC>f
Delete previous word	<ESC>h
Reset-keys	<ESC>k
Make a word lower case	<ESC>l
Query, search and replace	<ESC>q
Search-replace	<ESC>r
Set	<ESC>s
Unbind key	<ESC>^U
Make a word upper case	<ESC>u
Next page of next window	<ESC>^V
Previous-page	<ESC>v
Copy-region	<ESC>w
Insert-buffer	<ESC>^Y
Scroll window down one line	<ESC>z
Execute-line	<ESC><ESC>
Delete previous word	<ESC><DEL>

Function Keys

Delete next character	<DEL>
Delete previous character	<BACKSPACE>
Insert line, cursor is moved to next line	<RETURN>
Insert a tab	<TAB>

Chapter 6. The Workbench Drawer

The Workbench drawer contains three utilities:

CANCEL!	Prevents the appearance of autorequesters and alerts
ICONMERGE	Merges two single-image icons or splits a double-image icon
SETFONT	Changes the default type font used by the Workbench

CANCEL!

Format: CANCEL! [off]

Template: CANCEL! "OFF/S"

Purpose: To disable the appearance of autorequesters and alerts.

Specification:

CANCEL! automatically cancels any autorequesters or alerts that the system tries to display. Typical requesters display messages such as, "Disk Full," "Printer not Ready," or "Software Error -- Task Held." When one of these requesters appears, your work is interrupted and is not resumed until you click in one of the requester's options. Using CANCEL! prevents such interruptions, and is especially useful if you are running a program and you want to leave your Amiga, if you are running a remote terminal from a serial port, or if you are trying to recover a disk that is full of read errors.

The [off] option allows you to "turn off" CANCEL! and allows requesters to be displayed.

ICONMERGE

Format: ICONMERGE [<icon1> <icon2> <icon3> [OPT S]]

Template: ICONMERGE "ICON1,ICON2,ICON3,OPT/S"

Purpose: To merge the images of two single-image icons or to split the image of one double-image icon.

Specification:

ICONMERGE lets you merge the images of two single-image Workbench icons. It also lets you split a double-image icon into two single-image icons. A double-image icon is an icon that changes its shape when it is selected.

To run ICONMERGE from the Workbench, double-click on its icon. The ICONMERGE window appears and prompts you to enter either an m, to merge two single-image icons, or an s, to split a double-image icon. For example, to merge the CONTROL icon with the PRINTFILES icon. Enter an m at the prompt, and you'll get the following response:

```
------------------------------------------------------------------------------
      MERGE: Icon1, Icon2 (merge)-> Icon3

      Enter all icon filenames without ".info."
        Icon1:
------------------------------------------------------------------------------
```

At the prompt after Icon1:, enter "CONTROL" and hit RETURN. A second prompt will appear for Icon2:, enter "PRINTFILES". For Icon3:, enter "TEST". ICONMERGE will merge the two icons and the window will quickly disappear.

To see your new icon, you must close and re-open the Tools window. The new TEST icon will be on top of the CONTROL icon. Simply drag the TEST icon to an empty space in the window. Notice that TEST icon is an exact copy of the CONTROL icon (Icon1). When you click on the test icon, the PRINTFILES image (Icon2) will appear to show that the icon is selected.

The ICONMERGE icon is a double-image icon. To split that icon, double-click on the ICONMERGE icon to open the ICONMERGE window. Enter an s at the merge/split prompt, and the following display will appear:

```
------------------------------------------------------------------------------
      SPLIT: Icon1 (split)-> Icon2, Icon3

      Enter all icon filenames without ".info.":
        Icon1:
------------------------------------------------------------------------------
```

Enter ICONMERGE at the Icon1: prompt, ICON for Icon2:, and MERGE for Icon3:. After the ICONMERGE window disappears, close the Tools window, then re-open it. The two new icons, ICON and MERGE will be behind the original ICONMERGE icon. Drag them to an empty space in the window (or select Clean-up from the Workbench's Special menu). ICON will reflect the unselected appearance of the ICONMERGE icon. MERGE will reflect the highlighted appearance of ICONMERGE.

When you select either of these new icons, they are simply highlighted. They don't change form like a double-image icon would.

A Few Things to Remember:

1) Icon names must be entered without the .info suffix.
2) When merging icons, be sure they are the same size.

SETFONT

Format: SETFONT <font name> <size> [<styles>] [<flags>]

Template: SETFONT "FONTNAME/A,SIZE/A,STYLES,FLAGS"

Purpose: To change the default type font used by the Workbench and CLI.

Specification:

When you change the default font, the new font is displayed in the Workbench title bar and in the CLI windows. The type size of the icon names may change, but the font remains the same. The text in the window title bars is not changed.

The <fonts> available are the fonts in the Workbench's fonts directory:

FONT	SIZE
Ruby	8, 12 and 15
Diamond	12 and 20
Opal	9 and 12
Sapphire	14 and 19
Garnet	9 and 12
Emerald	17 and 20
Topaz	8, 9, and 11

The <style> options are E (extended), I (italic), B (bold face), and U (underline). The styles only affect the text in the Workbench title bar. The flags are M (in memory only), R (reversed), T (tall), W (wide), and P (proportional).

Font changes made with SETFONT are not saved to disk. When you reset your Workbench, the font will default to topaz.

Examples:

1> SETFONT TOPAZ 11

changes the default font to topaz, 11 point.

1> SETFONT OPAL 9BI

changes the default font to opal, 9 point, boldface and italicized.

Chapter 7. Debug

There are four utilities in the Debug drawer:

MEMLIST	Lists the addresses and sizes of memory blocks
MEMMUNG	Tests for illegal uses of memory
WATCHMEM	Inspects low memory for illegal writes
WEDGE	Monitors calls to any library function

WEDGE has several icons associated with it, therefore it has its own drawer in the Debug window.

MEMLIST

Format: MEMLIST

Template: MEMLIST

Purpose: To list the addresses and sizes of both free memory chunks and allocated memory blocks.

Specification:

To run MEMLIST from the Workbench, double-click on the MEMLIST icon. A MEMLIST window will open to display the output. You can also redirect the output to a file.

To prevent changes in the memory list, MEMLIST prompts to let it be re-executed without reloading. You must type a 'q' to quit, even if you have redirected the output to a file.

MEMLISTs are copied to an 8K buffer. Since MEMLIST executes in a FORBIDden state, it is not written to standard output until the MEMLIST list traversal is complete.

MEMMUNG

Format: MEMMUNG

Template: MEMMUNG

Purpose: To test for illegal use of memory.

Specification;

YOU MUST BE USING WORKBENCH VERSION 1.3 IN ORDER FOR MEMMUNG TO WORK PROPERLY.

When a block of memory is freed by a task, it is returned to the free memory pool, and it becomes illegal for any task to access that memory. However, every program is not perfect. Some programs have bugs that try to access the memory.

When MEMMUNG is running, any call to FreeMem() will be intercepted. First, the block will be cleared to $4e41, then the normal free will take place. Since this value is both odd and a Trap #1 instruction, any attempt by a task to illegally reference the free memory will probably cause a Guru.

Location 0 is set to $1e8b4e41 sto that any program illegally accessing a NULL pointer will change behavior.

It is possible to use Forbid(), free memory, then use the memory again. MEMMUNG will allow this.

To quit MEMMUNG, either run it again or send it a break by typing Control-C.

Example:

1> run MEMMUNG

If your software crashes while MEMMUNG is active, your software has a use-memory-after-free bug.

WATCHMEM

Format: WATCHMEM [<file>|<window>] [OPT N] [<interval>]

Template: WATCHMEM "TO,OPT/K,INTERVAL"

Purpose: To detect illegal writes to low memory.

Specification:

WATCHMEM inspects low memory for illegal writes. If any illegal writes are detected, WATCHMEM will intercept them and signal you by an alert. If you specify the [<file>|<window>] option, WATCHMEM will instead write to a window, the serial device, or a designated file. [OPT N] will disable the interception of the illegal write.

The [<interval>] option lets you specify the interval at which WATCHMEM inspects low memory. The default for this option is 2000 bytes.

To stop WATCHMEM, send it a BREAK signal by simultaneously pressing the Control and the C keys.

To activate WATCHMEM from the Workbench, double-click on its icon. When run from the Workbench, WATCHMEM inspects low memory at an interval of 2000 bytes and automatically corrects any illegal writes.

Examples:

```
>1 WATCHMEM
```

checks low memory for any illegal writes and signals you with an alert.

```
>1 WATCHMEM SER: OPT N
```

runs WATCHMEM as a separate process and sends the output to the serial device. The illegal write is not corrected because the N option was specified.

(WATCHMEM was inspired by the MemWatch program written by John Toebes VIII.)

WEDGE

Format: WEDGE <library> <offset> [REGS PTRS] [OPT FKLNPRSXZ U T=<tasklist> B=<baud> C=<comment> D=<n>]

Template: WEDGE "LIBRARY,OFFSET,REGS/K,PTRS/K,OPT/S,T=/K,B=/K,C=/K,D=/K"

Purpose: To monitor calls to any library function.

Specification:

WEDGE allows you to monitor the library function calls of system and application tasks. This is useful for both debugging and the optimization of software. WEDGE reports the name and address of the task calling the function, the Forbid/Disable status, the register contents, data pointed to by registers, stack, and the function return value.

You can restrict reporting to a list of tasks, and you can also exclude a list of tasks from reporting. When in serial mode, Debug (ROMWACK) can be invoked either before or after the WEDGEd function.

EXPLANATION OF WEDGE ARGUMENTS

In this section, the CLI argument is listed followed by the Workbench TOOL TYPE enclosed in parentheses. If an example is included to illustrate an argument, the example shows both the CLI argument and the corresponding TOOL TYPE.

Required Arguments

library (LIBRARY=) Specifies any run-time library (without the ".library" suffix).

Example: exec (LIBRARY=exec)

offset (OFFSET=) Specifies the library base offset of the function being WEDGEd. It must be expressed as a negative offset in hexadecimal form.

Example: 0xff28 (OFFSET=0xff28)

Optional Argument Pair

NOTE: REGS and PTRS must be used in conjunction with each other. If you supply one option, you MUST supply the other. They cannot be used alone.

REGS (REGS=) Hexadecimal word (format 0xHHHH) representing the registers that should be reported. Each bit set in the word represents a 68000 register, with the bits from left to right, representing d0 through d7 then a0 through a7.

Example: To monitor d0, d1, a0 and a6, the bit pattern would be:

```
        d0 d1     a0       a6
        \ /       \       /
Binary:  1100 0000 1000 0010
```

Convert the binary nibbles to hexadecimal:

Binary<>Hex Conversion Table

0000=0	0100=4	1000=8	1100=C
0001=1	0101=5	1001=9	1101=D
0010=2	0110=6	1010=A	1110=E
0011=3	0111=7	1011=B	1111=F

This example converted to hex is

HEX C 0 8 2

The regs argument would be:

0xC082 (REGS=0xC082)

PTRS (PTRS=) The hex word (format 0xHHHH) representing the monitored registers that point to the data you want reported. This is especially useful for monitoring text strings passed as function arguments. The first 16 bytes of the data will be reported in both hex and ASCII. The hex word is formed the same way as for the REGS option.

Example: If d1 and a0 are pointers:
0x4080 (PTRS=0x4080)

F	(FORBID=TRUE)	Causes WEDGE to Forbid() before calling the WEDGEd function. WEDGE will not call Permit() until Result is returned. This flag is only meaningful if Result reporting is in effect. When you are monitoring tasks that are calling the same functions, this flag can help to synchronize Result reports with other WEDGE output by attempting to prevent multitasking until the function returns. Functions that Wait() and local output will break this Forbid().
K	(KILL=TRUE)	Kills this WEDGE. Use KILL=TRUE for a kill-only icon. Use KILL=FALSE for an install-only icon. If a KILL TOOL TYPE is not present, the icon will toggle WEDGE in and out.)
L	(LOCAL=TRUE)	Selects studio reporting. In this mode, WEDGE can only report the function calls of non-FORBIDden processes with available stack of at least 1800 bytes. (Only 300 bytes are needed for serial or parallel output.) In addition, function calls made by the Local output handler will not be reported. (The Local output handler is usually a window, but it may be another handler if CLI output redirection was used.) In all output modes, WEDGE will bypass reporting if the caller's available stack is too low for safe execution of the output function. All unreported calls are tallied, and a count is presented at the next report. The available stack safety feature may be turned off (at your own risk) with the UNSAFE TOOL TYPE or with the CLI 'U' flag.
N	(NOISY=TRUE)	Notifies user of WEDGE installation and removal.
P	(PARALLEL=TRUE)	Selects parallel reporting, instead of serial. By default, all debugging output is directed to the serial port and may be received by a terminal, serial printer or a second computer running terminal software. Optionally, the output can be directed to the parallel port or displayed locally.

R (RESULT=TRUE) Monitors the return value of reported functions. An Id number is assigned to each reported function call. That call's Return value report is tagged with the same Id number. This allows you to match calls and return values, even if multitasking causes other WEDGE reports to be output in between them.

S (STACK=TRUE) Monitors stack limits and pointer.

X (EXCLUDE=TRUE) Excludes tasks in the tasklist from reporting.

U (UNSAFE=TRUE) Reports regardless of the available stack. Use this option at your own risk. When WEDGE is run from the CLI, the U flag may not be grouped with other flags (there must be a space before it).

Z (ZAP=TRUE) When used in conjunction with the K option, Z lets you force the unloading of a WEDGE that is still in use. Generally this is a very dangerous thing to do. But, occasionally, it is necessary because a WEDGE that is unWEDGEd with kill will not be unloaded if there is still an "occupant" in the WEDGE. This can occur if you have WEDGEd Wait() with Result reporting, and a task calls Wait(0), which will never Return. In this case, you can use the Z option to force the unloading of the Wait() WEDGE. Note: A new WEDGE may not be installed in a function if an old one has not been unloaded.

```
Usage -- CLI:  Opt KZ
         TOOL TYPES:    KILL=TRUE
                        ZAP=TRUE
```

B=rate (BAUD=RATE) Sets the serial hardware baud rate. This option is ignored if Parallel or Local output is requested. The baud rate is determined by the last value stored in the serper register. When you boot your system, the hardware is set to 9600 baud. If you use the serial.device or SER: before using WEDGE, the serper (serial period) register may contain a different rate. Use this option to reset the baud rate for WEDGE. If run from CLI, the B option without a specified rate will set 9600 baud. If you also use the D option to enter ROMWACK, you must use 9600 baud since ROMWACK forces 9600. Use quotes if you are passing this parameter to a script.

C=text (COMMENT="text") Allows you to include a comment with each report. The comment is generally the name and register argument descriptions for the function being WEDGEd. From CLI, quotes must be used around the entire C= string if the comment contains spaces. If entered from the Workbench, everything to the right of the = sign should be in quotes. The WEDGE comments are displayed when you do a WEDGE List. If a WEDGE has no comment, its offset and library will be listed.

```
Usage -- CLI:          "c=AvailMem d1=Type"
         TOOL TYPE:    COMMENT="AvailMem d1=Type"
```

D=n (DEBUG=n) Causes a call to Debug() on the nth reported call of the WEDGEd function. (This option is only valid in a serial WEDGE.) By default, Debug() invokes the system's built-in 9600 baud serial debugger ROMWACK. Since ROMWACK forces 9600 baud, you should only use this option in a 9600 baud WEDGE (unless you have a resident serial debugger that works at other baud rates).

If Result reporting (OPT R) is in effect, the call to Debug() will be made AFTER the WEDGEd function has been called and its result reported. If Result reporting is not on, Debug() will be called after the normal WEDGE report and immediately before WEDGE actually calls the function. This allows you to enter the debugger either before or after the WEDGEd function has been called. In both cases, you will enter the debugger with all of the function caller's registers intact, except for a6 which will contain ExecBase. The first two values on the stack will be the return address for Debug() and the caller's a6.

When you exit the debugger, you will be prompted "Debug again <y or n>?" at the serial terminal. By entering 'y', you can Debug successive calls to the WEDGEd function. Note that Debug will not be invoked if the caller is running on the system stack because the caller may be an interrupt.

When run from the CLI, the D option used without an argument is equivalen to "d=1."

```
Usage -- CLI:          "d=6" or Opt D
         TOOL TYPE:    DEBUG=1
```

T=tasklist
(TASKS="tasklist")

Limits reporting to the tasks named in Tasklist. This is useful for monitoring the function calls of a particular task or group of tasks. If the X Option (EXCLUDE) is used, the tasks in the tasklist are instead excluded from reporting. This is useful for screening out any Amiga OS tasks that make heavy use of the function being monitored.

The tasklist should be in quotes, and multiple task names should be separated by a vertical bar. WEDGE will compare the tasklist names against both Task node names and the command name of any CLI invoked command which calls the WEDGEd function.

WEDGE also recognizes two special task names:

System: Matches any code running on system stack (interrupts, etc.).

All: If this is the only task name in the list, the list is ignored. This allows you to disable the TASKS TOOL TYPE without removing it from the .info file.

```
Usage -- CLI:          "T=System|CON|wedge"
         TOOL TYPE:    TASKS="System|CON|wedge"
```

RUNNING WEDGE FROM THE WORKBENCH

There are eight icons in the Wedge drawer. The Wedge (function) icon is the actual WEDGE tool; the other icons (FunctionName.w) either contain scripts for wedging common system functions or sample projects for starting similar WEDGEs from the Workbench. New scripts and icons can be easily created using the offset and register lists in the back of the Addison-Wesley *ROM Kernel Reference Manual: Exec.* (DOS offsets are incorrect. See "Creating Your Own WEDGE Scripts and Icons," later in this chapter.)

When WEDGE is run directly from the Workbench, the TOOL TYPES entries in the icon's INFO window determine which library and offset to monitor and which options are enabled/disabled. To change an entry or add an option that is not listed, use the Workbench INFO function. To bring up the INFO window, select a Wedge icon, then select the INFO option in the Workbench's first menu (the Workbench menu). Use the scroll arrows, the add/delete gadgets, and the keyboard to add, delete or modify the icon's TOOL TYPE entries.

The acceptable format for TOOL TYPES entries was included in the explanation of each option. The TOOL TYPES default to:

AmigaDOS argument	TOOL TYPE Default
REGS	REGS=0x0000
PTRS	PTRS=0x0000
L	LOCAL=FALSE
P	PARALLEL=FALSE
S	STACK=FALSE
X	EXCLUDE=FALSE
"T=tasklist"	TASKS=ALL

There are no default values for the library and offset arguements. **These values MUST be supplied.**

SPECIAL WEDGE FUNCTIONS

The following functions can be specified with a single CLI argument, a TOOL TYPES entry, or simply by double-clicking on the corresponding icon in the WEDGE window.

CLI Command	TOOL TYPES	Icon	Function
WEDGE help	HELP=TRUE	Wedge.Help	Lists WEDGE arguments, TOOL TYPES, and notes.
WEDGE killall	KILLALL=TRUE	Wedge.KillAll	Signals all WEDGEs to remove themselves. If no WEDGEs are currently installed, this option will remove and deallocate the resident "WedgeMaster" Message Port which holds some global information for the WEDGEs.
WEDGE list	LIST=TRUE	Wedge.List	Lists all WEDGEs currently installed.

WEDGE LIMITATIONS

Due to the need for WEDGE to be fast and small and to call as few library functions as possible, there are certain arbitrary limitations.

1. The Exec functions RawIOInit, RawDoFmt, RawMayGetChar, and RawPutChar() may not be WEDGEd because they are called indirectly by the WEDGE itself. If these functions were WEDGEd, a recursive loop would occur. WEDGE caches pointers to Forbid and Permit. Installed WEDGEs call these functions using these pointers, so these functions may be WEDGEd.

2. To help prevent such recursive loops in Local mode, function calls made by the Local output handler are not reported. Unless CLI output redirection has been used, the Local handler is a CON: window. All function calls made by the standard Amiga Exec devices are automatically screened out in Local mode, since our devices are not Processes. However, be warned that if you redirect Local output to a third party handler or device, a recursive loop (and big crash) can occur.

3. There are size limits for several WEDGE arguments:

tasklist	31 task names; 319 characters
library	39 characters
comment	79 characters

4. The maximum size for KILLALL and LIST is 127 WEDGEs.

5. The CLI command names specified in the tasklist option must be 16 characters or less. If you want to monitor or exclude a command with a larger name, rename it. There is no limit on the size of normal Exec Task names in the tasklist.

WARNINGS

1. In some cases, the caller of a WEDGEd function may have so little available stack that the WEDGE code to save the caller's registers and to check his stack may overrun it. If this happens, severe crashes can occur. Use the tasklist to screen out tasks with tiny stacks.

2. WEDGE contains a number of safety features to prevent recursion. However, the possibility of recursive crashes still exists, most notably in Local mode. See the note on Local recursion in "WEDGE Limitations."

3. When creating masks for REGS and PTRS, do not indiscriminately specify registers as pointers. Only specify a register as a pointer if the address of a string, structure, buffer, etc. is actually passed in that register. If a register is specified as a pointer, its contents will be used as an address and the data at that address will be read. If the register actually contains flags or other non-address data, you could end up reading registers which are reset by a read. This could cause a crash.

4 If you WEDGE common Exec functions, such as AllocMem() and Signal(), without excluding low level OS tasks such as input.device and trackdisk.device, the system will slow to a crawl. It is strongly suggested that you do not write to disks while an intensive WEDGE is running. (It would take forever anyway.)

SAMPLE WEDGE COMMANDS AND RESULTING OUTPUT

NOTE: Normally you would set up a ".w" script and/or icon to install a WEDGE.

Example 1.

WEDGE exec.library AvailMem function (oxff28) with Stack reporting (OPT S). Only report on calls made by Workbench or the CLI command AVAIL.

Enter:

```
1> run Toolkit:Debug/Wedge/WEDGE exec 0xff28 OPT S "C=AvailMem d1=Type"
"t=Workbench|Avail"
```

Sample WEDGE Output:

```
-------------------------------------------------------------------------------
AvailMem d1=Type
COMMAND: Avail    PROCESS: Initial CLI ($203798) [F]
Pre-wedge a7=$232D7C  Task Stack:  Lower=$203854, Upper=$203E94
Command Stack: Base at startup=$232DF8, Size=10000

AvailMem d1=Type
PROCESS:  Workbench ($20FB40)
Pre-wedge a7=$210F14   Task Stack: Lower=$20FBFC, Upper=$210F84
-------------------------------------------------------------------------------
```

When a CLI command calls the WEDGEd function, the COMMAND name is reported. In addition, if Stack reporting has been requested, both the PROCESS and the separate COMMAND stack are reported. Also note the "[F]" in one of the reports. This means the task calling the function had a Forbid() in effect. A Disable() would have been reported as "[D]", both as "[FD]".

Example 2

WEDGE dos.library Open function (offset 0xffe2), report registers d1 and d2 (0x6000). Report what d1 points at (0x4000), and report both Result (OPT R) and Stack (OPT S).

Enter:

```
1> run Toolkit:Debug/Wedge/WEDGE dos 0xffe2 0x6000 0x4000 OPT RS "C=Open d1=Name d2=Mode"
```

Sample WEDGE Output when Workbench calls Open (".info",MODE_NEWFILE):

```
-------------------------------------------------------------------------------
Open d1=Name d2=Mode
PROCESS: Workbench ($20FB40)
d1 $00FECD60 -> $2E696E66 6F005359 533A5379 7374656D .info.SYS:System
d2 $000003EE
Pre-wedge a7=$210DFC   Task Stack: Lower=$20FBFC, Upper=$210F04
Result ID=6
Result: $86EB9  (ID=6)
-------------------------------------------------------------------------------
```

In the ASCII dump at the right, all unprintable values, such as the 0 terminating the ".info" filename, are printed as a period ("."). All output lines except the "PROCESS" line are optional and are requested via CLI arguments or TOOL TYPES.

The OpenWindow.w, Open.w, and AvailMem.w icons have been provided for wedging common system functions. OpenWindow.w installs a WEDGE into $FF34 of the intuition.library; Open.w installs a WEDGE into $FFE2 of the dos.library; and AvailMem.w installs a WEDGE into $FF28 of the exec.library. These projects can be run through the Workbench. To change any of the parameters, change the TOOL TYPES entries in the icon's Info window.

The icons are set up for Local output. The scripts are set up for default (serial) output with no baud rate specified. To use serial scripts for Local or Parallel debugging, pass the appropriate flag when you execute the script. In order for Execute to find the scripts, you must cd where they are or copy them to your s: directory. Here are some examples of passing options to the AvailMem.w script:

```
execute AvailMem.w prs  (prs = Parallel, Result reports, Stack reports)
execute AvailMem.w l    (l = Local)
execute AvailMem.w dr   (dr = Debug on first call, Result reports)
```

CREATING YOUR OWN WEDGE SCRIPTS AND ICONS

To create additional WEDGE icons, duplicate an existing icon, then go into the icon's INFO window to modify the TOOL TYPES arguments.

Each icon is attached to a ".w" script file, so you should use ED or MEMACS to modify the script file attached to your icon. If you do not, the script file will not install the desired WEDGE from CLI since it will be a duplicate of the script attached to the icon you copied.

Offsets for all system library functions may be found in the back of the *Addison-Wesley ROM Kernel Reference Manual: Exec*. The manual contains several representations of each offset. Use the hex form that starts with "0xf". Note that the list of dos.library offsets is incorrect. A correct list starts with 0xffe2. An incorrect dos list starts with 0x0000. If your list starts with 0x0000, find the dos function you want to WEDGE, and use the offset of the 5th function after it in the list. (Equivalent to subtracting 0x1e and using low word of the result.)

Chapter 8. IFF

The IFF drawer of the Software Toolkit window contains two utilities for displaying and saving graphics using the IFF ILBM (InterLeaved bitplane BitMap image) format.

DISPLAY	Displays graphics saved in IFF ILBM format
SCREENSAVE	Saves graphics in IFF ILBM format

DISPLAY

Format: DISPLAY <filename> [<time>]

Template: DISPLAY "FILENAME/A,TIME"

Purpose: To display graphics saved in IFF ILBM format.

Specification:

DISPLAY displays graphics saved using the IFF ILBM format, such as files saved with SCREENSAVE. When run from the CLI, you must specify the name of the file in which the graphic is contained. The value given for the <time> option specifies the number of seconds the graphic will be displayed.

When the graphic is displayed the pointer becomes a dot. Hitting the TAB key cycles the colors at the given time rate. Clicking a mouse button briefly displays the screen's title bar and close gadget. To close the picture, click on the hidden close gadget in the upper-left corner of the screen.

To run DISPLAY from the Workbench, click once on the DISPLAY icon, then while holding down the SHIFT key, double-click on the icon of the file to display.

SCREENSAVE

Format: SCREENSAVE [<filename>]

Template: SCREENSAVE "FILENAME"

Purpose: To save an Intuition screen using IFF ILBM format.

Specification:

SCREENSAVE enables you to save the frontmost Intuition screen to a file in IFF ILBM format. This file can then be printed or manipulated using one of the various IFF drawing programs.

To run SCREENSAVE from the Workbench, double-click on its icon. SCREENSAVE opens a window and prompts:

```
----------------------------------------
SCREENSAVE -- C. Scheppner CBM 10/86
Saves the front screen as an IFF ILBM file.
A CAMG chunk is saved (for HAM pics, etc.)

Enter filename for save:
----------------------------------------
```

Enter a path to the file where you want the screen to be saved, and hit RETURN.

Next you see:

```
----------------------------------------
Click here and press RETURN when ready:
Front screen will be saved in 10 seconds
----------------------------------------
```

The 10 second delay gives you time to arrange and/or bring to the front the screen that you want to save. When those ten seconds are up, the disk drive light will come on, indicating that the screen is being saved. Do not move the screen until the disk drive light goes out. When the save is finished, return to the SCREENSAVE window.

```
----------------------------------------
Saving ...
Screen saved
Icon saved
Done

PRESS RETURN TO EXIT
----------------------------------------
```

Press RETURN to exit.

Chapter 9. Other

The Other drawer contains six utilities that let you test your printer and also provide various graphic displays of system features and functions:

CARDS	Displays the index cards in an ASCII card file, such as those stored in the 68000.cards file
FREEMAP	Displays chip memory usage
KEYTOY1000	Displays the current global keymap for the Amiga 1000
KEYTOY2000	Displays the current global keymap for the Amiga 500 and Amiga 2000
PERFMON	Displays graphic representation of system-wide performance
PRINTERTEST	Tests a printer's features

CARDS

Format: CARDS <cardfile>

Template: CARDS "CARDFILE/A"

Purpose: To display the index cards in an ASCII card file.

Specification:

CARDS allows you to look at index cards stored in a specially formatted ASCII cardfile. There is a cardfile called 68000.cards in the Other window. This file contains index cards describing the format, sizes and results of each 68000 machine language instruction.

To look at these cards, double-click on the 68000.cards icon. You can also use the CD command to make the directory containing the cards the current directory (in this case, Software Toolkit's Other directory), then use the command "CARDS 68000.cards."

When the 68000.cards display appears, there is a highlighted index line at the bottom of the display:

```
.0123456789.ABCDEFGHIJKLMNOPQRSTUVWXYZ.          <>
```

The periods ('.') in the index line represent ranges of special and punctuation ASCII characters. To display the first card that starts with a letter or number, click on the appropriate letter/number in the index line. To move through the cards one by one, use the < (backward) and > (forward) gadgets on the right side of the index line.

Initially, the cards are sorted according to the category of the first line, in this case alphabetically. However, you can sort the cards by any category by selecting the corresponding line description on the left side of the display. A '>' character marks the selected line by which the cards are sorted. Generally, it is best to sort the 68000.cards cardfile on the first line which is the mnemonic name of each 68000 instruction.

You can also move through the cards by using the keyboard. Hitting any alphanumeric key moves you to the first card starting with the letter or number. Other keyboard shortcuts are:

< or ,	move backwards through the cards one at a time
> or .	move forward through the cards one at a time
Shift-U	move the sort line up
Shift-D	move the sort line down

To close the CARDS window and exit the program, hit Ctrl-C.

CREATING ADDITIONAL CARD FILES

Additional card files can be created with any ASCII text editor. The start of the card file must be a card template. The rules for this template are explained below and are illustrated in the example.

Template Example

```
1 --      @TEMPLATE
2 --      @N=50
3 --       @@@@@@@@@@@|@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ @@@
4 --       ------------------------CLIENTS------------------------
5 --       Name
           Company
           Phone
6 --      @ENDTEMPLATE
```

1. The first line of the card file must start with the @TEMPLATE keyword.

2. The second line is the @N=MaxCards line. This line specifies the maximum number of cards allowed in the file. It is optional, and if not specified the default value is 640 cards. In the above example, 50 is the maximum number of cards permitted.

3. The third line is a series of '@' symbols to specify the width of the card. One @ symbol is equal to one character space in the card. You need to insert a vertical bar (|) to designate the break between the line titles (on the left) and the individual card entries.

4. The fourth line contains the text for the card title line.

5. The next several lines specify the left-hand line titles. (Remember, the line titles cannot exceed the number of @ symbols before the vertical bar in the third line.)

6. Finally, the template is concluded with the @ENDTEMPLATE keyword.

After the template is created, you can enter the text for each card. Simply, start each entry with a @ symbol on the first line. (This should be the only entry on the first line.) For example:

```
@
Smith, P.J.
Hypertronics, Inc.
215-555-3434

@
Jones, Mandy
Widgets, Ltd.
408-555-6767
```

You could continue on in this fashion with another 48 card entries.

FREEMAP

Format: FREEMAP

Template: FREEMAP

Purpose: To give a graphic display of chip memory usage.

Specification:

FREEMAP gives a graphic display of chip memory usage, showing the blocks used and the free holes. Double-click on the FREEMAP icon, and a "map" appears. Each pixel in the map area represents a 64 byte block. If all the bytes in a block are free, the pixel is dark. If a byte is not free, the pixel is light.

The amount of free chip RAM and fast RAM is listed at the bottom of the window.

To quite FREEMAP, choose Please from the Quit menu. Selecting the INFO menu, brings up a brief explanation of FREEMAP. To restore the FREEMAP display after viewing Info, choose Redisplay.

KEYTOY1000

Format: KEYTOY1000

Template: KEYTOY1000

Purpose: To display the current global keymap for the A1000 keyboard.

Specification:

Double-clicking on the KEYTOY1000 icon results in a display of the global keymap for the keyboard accompanying the Amiga 1000 computer.

The initial display reflects the characters that are output when each key is pressed alone. To see the characters that are output when qualifier keys are pressed simultaneously with a character key, either click on the corresponding gadget in the KEYTOY display, or hit the key on the keyboard. The qualifier keys are Control, both Shifts, and both Alts.

The following list is a guide to interpreting the KEYTOY display:

* all KEYTOY keys labeled in blue are not used by KEYTOY (That's why CTRL, SHIFT and ALT appear in blue in the initial display. After you select one or more of these keys, the label changes color.)
* any key labeled in red is a dead key, except for the two AMIGA keys and the CAPS LOCK key
* any key labeled in bold-italic is deadable
* any key labeled with $$ is defined by a character string longer than one character
* a character preceded by ^ or ~ is a control character
* if a key is blank, it is undefined for the current choice of qualifier(s)

KEYTOY2000

Format: KEYTOY2000

Template: KEYTOY2000

Purpose: To display the current global keyboard for the Amiga 500 and the Amiga 2000.

Specification:

KEYTOY2000 displays the global keyboard for the Amiga 500 and the Amiga 2000. The display follows the same rules as described for the KEYTOY1000.

PERFMON

Format: PERFMON

Template: PERFMON

Purpose: To provide a graphic display of system-wide performance statistics.

Specification:

PERFMON provides a graphic display of the system-wide performance statistics and updates them approximately once a second.

Double-click on the PERFMON icon, and the Performance Monitor window appears. You can choose the time interval for the performance update from the PM Menu. The options are: .5, 1 and 2 seconds.

The top portion of the window shows the amount of CPU time currently in use (calibrated for a 68000). The black center line indicates 100% CPU utilization.

The bottom portion of the window shows the amount of chip RAM currently in use. The bottom black line indicates 100% chip memory utilization.

The amount of fast memory available is displayed at the bottom of the window.

PRINTERTEST

Format: PRINTERTEST

Template: PRINTERTEST

Purpose: To test the printer.

Specification:

The PRINTERTEST program allows you to test the capabilities and features of printers used with Amiga computers.

Double-click on the PRINTERTEST icon and a window appears. The window is actually made up of three custom screens. The top screen is the Lo Res HAM Screen for printer test is the top screen. This is a low-resolution Hold and Modify screen that is in the black-and-white test. The second screen, Lo Res LACE Screen for printer test, is a low-resolution interlaced color test screen that is used for the color test. The final screen is a high-resolution prompt screen which allows you to communicate with the program.

Prompts instruct you on how to proceed. Initially, PRINTERTEST requests the name of the printer which will be displayed. For example:

```
------------------------------------------------------------------------------
    Test CBM_MPS1000? (y,n)[n]
------------------------------------------------------------------------------
```

Continue through this list until the printer attached to your computer is listed, then choose "yes."

The test consists of four parts: Alpha test, Reset test, Black & White test, and a Color test. You can stop after any of the tests.

Alpha Test

This test is designed to test how the printer will react to various Printer Device Command Functions (For a list of the commands, see the Printer Device chapter of the *Amiga ROM Kernel Manual: Libraries and Devices.*) These functions include type styles and sizes, colors, superscripts and subscripts, justification, tabs, margins, etc.

Reset Test

After the Alpha test is completed, the prompt screen asks the user to select any key to begin the Reset Test. This test simply checks the reset command function.

Black & White Test

When the Reset test is finished, PRINTERTEST prompts:

```
--------------------------------------------------------------------------------
    B&W graphics test?  (y,n)
--------------------------------------------------------------------------------
```

If you wish to continue with the test, select yes. If your printer has graphics capability, PRINTERTEST will do a black and white dump of the custom Hold and Modify screen (the top screen) to demonstrate the printer's ability to print gray-scale graphics.

Color Test

Finally, if your printer has color capability in addition to graphics, PRINTERTEST will dump the low-resolution interlaced color screen (the middle screen). Compare the screen colors to the resulting print out.

Chapter 10. C (The Command Directory)

The C drawer in the Software Toolkit window does not contain any icons, but it does represent the additional AmigaDOS CLI commands available in the C directory of the Software Toolkit disk. With these commands, you can extend your AmigaDOS capabilities so that you can perform extensive debugging of programs, convert and link binary files to loadable files, repair corrupt disk blocks, keep close tabs on your system's memory, and much more.

The AmigaDOS commands included on Software Toolkit are:

ABSLOAD	ADDMEM
ALINK	ATOM
DISKED	DRIP
FRAGS	MOREROWS
PRINTA	READ
ROMWACK	SETPARALLEL
SNOOP	STRIPA
STRIPC	WACK

Some of these commands are documented in other sources, such as the *Amiga ROM Kernel Manual* and *The AmigaDOS Manual*. But for your convenience, the commands will be fully explained in this manual. (WACK was explained in Chapter 2.)

ABSLOAD

Format: ABSLOAD [-D] [-M] [-X] [-S] [-B <Ksize>] [-O <sfile>] [-T<addr>] [<lfile>]

Template: ABSLOAD "-D/S,-M/S,-X/S,-S/S,-B/K,-O/K,-T/K"

Purpose: To convert Amiga loader format to Motorola srecords.

Specification:

ABSLOAD converts Amiga Binary object files from the linker ALINK and creates a Motorola srecord file suitable for downloading to an Amiga machine. The starting address of the code is assumed to be the first code hunk found in the input file.

The options available are:

[-D] Writes debugging information to standard error. This option is primarily useful if the linker is creating a file that ABSLOAD cannot understand. Much of this information is similar to that from PRINTA. The first line printed is the version of ABSLOAD. (Default is not to write debugging information.)

[-M] Writes map of hunk addresses to standard output. The map is a list of lines with three fields: decimal hunk number, hexadecimal hunk base byte address, and hexadecimal byte length of hunk. (Default is not to produce a map.)

[-X] Excludes the hunk map from memory image. This option is used to save memory (e.g. ROMs), but it requires that you use the -M option in order to do any debugging.

Note: When using the resulting srecord file, the starting address of the code is not the same as the starting address in memory unless the -X option is used. When you download the file with the Amiga debugger's LOAD command, the starting address will be printed. Make sure you use the starting address for your GO command.

[-S] Suppresses same length. Writes the srecords with the last srecord having the exact length specified. (Default is to zero pad the last srecord to have the same length as the others.)

[-B <Ksize>] Specifies the decimal number of kilobytes ABSLOAD is to use to build its memory image of the file. (Default for the Sun is 1/2 Megabyte. Default for the Amiga is 16 Kbytes.)

[-O <sfile>] Writes srecords to <sfile>. (Default is file 'a.srec.')

[-T <addr>] Specifies byte address in hexadecimal for ABSLOAD to begin allocating memory for the absolute binary form. The address must be long word aligned. (Default is hex 20000.)

[<lfile>] Specifies the file, created by ALINK, that is to be converted. If no file is specified, the standard input is used.

A brief synopsis of the options, along with an error message, is printed if the correct syntax is not followed.

For debugging purposes: If you did not use the -M option and the -X option, and you need to know the address of a hunk, it can be found at your memory starting address plus four times the hunk number. Hunk numbers start from zero.

With the map produced from ALINK and the map produced from ABSLOAD, you should be able to determine the location of all externals. Note that PRINTA's and ALINK's maps number hunks starting at 1. ABSLOAD numbers hunks starting at 0.

See Also: ALINK, PRINTA, ASSEM

Bugs: Does not handle overlays.

ADDMEM

Format: ADDMEM <lower-hex-address> <upper-hex-address> [NOCLEAR]

Template: ADDMEM "LOWER-HEX-ADDRESS,UPPER-HEX-ADDRESS,NOCLEAR/S"

Purpose: To link external, non-autoconfigured RAM added to the Amiga's system free list.

Specification:

ADDMEM is used to add external RAM to the Amiga. The memory is tested and linked to the system free list. If the NOCLEAR option is specified, the memory is not cleared.

ADDMEM uses Exec's AddMemList() function, and the external memory is given the attributes MEMF_PUBLIC|MEMF_FAST. The priority specified is zero.

Example:

1> ADDMEM F00000 F7FFFF

adds one-half megabyte of memory.

1> ADDMEM F00000 F40000 noclear

adds 256K of memory, without clearing.

ALINK

Format: ALINK [FROM <objectfiles>] [WITH <files>] [TO <file>] [MAP <file>] [LIBRARY <files>] [VER <file>] [XREF <file>] [width <n>]

Template: ALINK "FROM=ROOT,WITH/K,TO/K,LIBRARY=LIB/K,MAP/K,VER/K,XREF/K, WIDTH/K"

Purpose: To link Amiga binary object files into loadable files.

Specification:

ALINK is a linkage editor to convert Amiga binary object files to Amiga load files.

In the above format, <file> stands for a single file name, while <files> means zero or more file names that must be separated by a comma, plus, or space. If file names are separated by spaces, the parameter must be enclosed in quotes.

[FROM <objectfiles>] Specifies the object files to be used as the primary binary input. The contents of these files are always copied to the load file to form part of the overlay root.

[WITH <files>] Specifies files containing the linker parameters (e.g. normal command lines). Usually only one file will be used, but, for completeness, a list of files can be given. Parameters on the command line will override those in WITH files. Lines can be continued by putting an asterisk (*) at the end of them. WITH files are frequently used when the command line gets too long.

[TO <file>] Specifies the destination for the load file. If this parameter is not given, the second pass is omitted.

[MAP <file>] Writes a map of addresses within hunks to the <file> specified.

[XREF <file>] Specifies the destination of the cross reference output.

[VER <file>] Specifies the destination of messages from ALINK. If a file is not specified, all messages are sent to the standard output.

[LIBRARY <files>] Specifies the files to be scanned as the library. Only referenced code segments in these files will be included.

[WIDTH <n>] Specifies the output width (<n> characters) to be used when producing the link map and cross reference table.

Example:

```
1> ALINK a.obj,b.obj,c.obj LIBRARY amiga.lib MAP demo.map TO demo.ld
```

Converts and links binary files a, b and c to load file demo.

See Also: ASSEM, ABSLOAD, PRINTA

Bugs: The MAP output only gives the first eight characters of a symbol's name.

For more information on ALINK, see the *AmigaDOS Developer's Manual*.

ATOM

Format: ATOM <infile> <outfile> [-I]
or
ATOM <infile> <outfile> [-C[C|D|B]] [-F[C|D|B]] [-P[C|D|B]]

Template: ATOM "INFILE,OUTFILE,-I/K"
or
ATOM "INFILE,OUTFILE,-C/K,-F/K,-P/K"

Purpose: To specify the portions of a program that should be loaded into CHIP memory and/or FAST memory.

Specification:

ATOM is used to specify to the loader the portions of a program to be loaded into CHIP memory, FAST memory, and PUBLIC memory. Some system structures (like Images) must be stored in CHIP memory to work. Others can take advantage of the increased speed of FAST memory, while others can work in either. Normally, the loader tries to use FAST memory first, if no other type (or PUBLIC) is specified.

You can compile or assemble a program, then pass the individual object modules through ATOM before they are linked to form a load file. ATOM will flag the desired hunks (CODE, DATA and BSS) and place them in the specified type of memory (CHIP, FAST, or PUBLIC).

When using ATOM, <infile> represents an object file that has just been compiled, assembled or ATOMed. <outfile> represents the destination for the converted file. The options are as follows:

[-I] operate interactively

[-C] change memory to CHIP
[-F] change memory to FAST
[-P] change memory to PUBLIC
[C] change CODE hunks
[D] change DATA hunks
[B] change BSS hunks

ATOM ERROR MESSAGES

Error Bad Args:

1. an option does not start with a "-".
2. wrong number of parameters.
3. "-" not followed by I, C, F or P.
4. -x supplied in addition to -I.

Error Bad Infile = file not found.

Error Bad Outfile = file cannot be created.

Error Bad Type ## = ATOM has detected a hunk type that it does not recognize; the object file may be corrupt.

Error Empty Input = input file does not contain any data.

Error Read Externals = external reference or definition if of an undefined type; object file may be corrupt.

Error Premature End of File = an end of file condition (out of data) was detected while ATOM was still expecting input. Object file may be corrupt.

Error This Utility can only be used on files that have NOT been passed through ALINK = the input file you specified has already been processed by the linker; external symbols have been removed and hunks coagulated. You need to run ATOM on the object files produced by the C compiler or Macro Assembler BEFORE they are linked.

Examples:

1> ATOM infile.obj outfile.obj -pc -cdb

All code hunks go into Public RAM; data and BSS hunks go into chip RAM.

1> ATOM infile.obj outfile.obj -c

All hunks in the object file are loaded into chip memory.

1> ATOM myfile.o myfile.set.o -cd

Sets all data hunks to load into chip memory.

DISKED

Format: DISKED <device>

Template: DISKED "DEVICE/A"

Purpose: To inspect and patch disk blocks.

Specification:

You can use DISKED to inspect and patch disk blocks; for instance, to recover information from a corrupt floppy disk. However, because DISKED writes directly to the disk, you should use it carefully. A disk does not have to be inserted to be examined by DISKED.

Only use DISKED with reference to the layout of an AmigaDOS disk. (For a description of this layout refer to *The AmigaDOS Manual*.) DISKED understands the AmigaDOS layout. For instance, the R (Root Block) command prints the key of the root block. The G (Get Block) command followed by the key number reads the block into memory, and the I (Information) command prints out the information contained in the first and last locations. This information indicates the type of block, the name, the hash links, and so on. If you specify a name after an H (Hash) command, DISKED gives you an offset on a directory page that stores as the first key any headers with names that has to the name you supplied. If you then type the number that DISKED returns followed by a slash (/), DISKED displays the key of that header page. You can then read that key with further G commands, and so on.

All DISKED commands are single characters, sometimes followed by arguments. The following is a complete list of the available commands.

COMMAND	FUNCTION
B n	Sets logical block number base to n
C n	Displays n characters from the current offset
G [n]	Gets block n from the disk (default is the current block number)
H name	Calculates the hash value of the name
	NOTE: If you specify a name after an H command, DISKED gives you an offset on a directory page that stores, as the first key, headers with names that hash to the name you specified. If you then type the number that DISKED returns followed by a slash (/), DISKED displays the key of that header page. You can then read on with further G commands, etc.
I	Displays block information
K	Checks block checksum (and corrects it if wrong)
L [lwb upb]	Locates words that match value under mask (lwb and upb restrict the search)
M n	Sets mask to n (for L and N commands)

N [lwb upb]	Locates words that do not match value under mask
P n	Puts block in memory to block n on disk (default is the current block number)
Q	Quit (do not write to disk)
R	Displays block number of root block
S char	Sets display style: char = C (characters), S (string), O (octal), X (hexadecimal), or D (decimal)
T lwb upb	Types range of offsets in block
V n	Sets value for L and N commands
W	Windup (P & Q)
X	Inverts write-protect state
Y n	Sets cylinder base to n
Z	Zero all words of buffer
number	Sets current word offset in block; equals the display values set in program
?	Displays current values (number of cylinders, blocks per track, number of blocks, block size, etc.)
/[n]	Displays word at current offset or updates value to n
'chars'	Puts characters at current offset
"chars"	Puts string at current offset

To indicate octal, start numbers with #; for hexadecimal, start numbers with #X. You can also include BCPL language string escapes in strings (*N and so forth).

Example 1:

Because DISKED can be somewhat difficult to learn, here is a sample session. You can try this out on a copy of your Workbench disk. It will go through the most useful of the DISKED commands. The block numbers in the sample session may not match your results exactly. Therefore, you will need to fill in the block numbers DISKED displays on your system.

It is suggested to follow along in the *AmigaDOS Technical Reference Manual.*

NOTE: In this example, everything after a ";" is a comment and should not be typed.

```
HARDCOPY v1.0 recorded on 20-Jun-88 19:14:52 to file "ram:temp"
To end the HARDCOPY session and close the file, type HARDCOPY END

RAM-DISK:> disked df0:
AmigaDOS Disk Editor version 3.4
Write protect mode set, cylinder base set to 0
# r                       ;Find the root block
 Root Block is block 880
# g 880                   ;Get the root block
 Block 880 read (cyl 40, sur 0, sec 0)
# i                       ;Display info about current block
 Short file
 Header key:       0
 Highest seq num:  0
 Data size:        72
 First data block: 0
 Checksum:         -173384115
 Secondary type:   Root block
 Parent Directory: 0
 Hash Chain:       0
 Filename:         "Workbench 1.3"
 Modified:         Monday 20-Jun-88 18:52:18
 Created:          Monday 20-Jun-88 11:45:57
 Bitmap is Invalid
# hc                      ;Get the hash value for the "C:" directory
 Hash value = 14
# 14                      ;Go to (longword) offset 14
# /                       ;Look at what is at the current offset
 889
# g 889                   ;Go to the "C:" directory
 Block 889 read (cyl 40, sur 0, sec 9)
# hEd                     ;Look up the hash of "Ed"
 Hash value = 13
# heD                     ;Case is ignored for hashing
 Hash value = 13
```

```
# 13
# /
 910
# g 910
 Block 910 read (cyl 41, sur 0, sec 8)
# i                     ;Get the info
 Short file
 Header key:       910
 Highest seq num:  41
 Data size:        0
 First data block: 911
 Checksum:         -59184531
 Secondary type:   File
 Extension block : 0
 Parent Directory: 889
 Hash Chain:       1346
 Filename:         "Ed"
 Created:          Monday 20-Jun-88 10:55:40

      ;When two or more names have the same hash value,
      ;they are chained together.  In this case "Hash Chain"
      ;points to the next one in line.

# g 1346                ;Go to next in hash chain
 Block 1346 read (cyl 61, sur 0, sec 4)
# i
 Short file
 Header key:       1346
 Highest seq num:  6
 Data size:        0
 First data block: 1347
 Checksum:         435379822
 Secondary type:   File
 Extension block : 0
 Parent Directory: 889
 Hash Chain:       0
 Filename:         "Binddrivers"
 Created:          Monday 20-Jun-88 19:13:34

# 82                    ;Go to offset 82, where the comment is stored
# sx                    ;Set display to hex
# /
 00000000              ;No comment yet!
# /#X01410000          ;Set a comment as the BCPL string "A"
# /
 01410000              ;Check it, looks ok!
# k
*** warning - checksum incorrect
***         - value in block was -38147475
***         - corrected to       -59184531
```

Example 1:

Because DISKED can be somewhat difficult to learn, here is a sample session. You can try this out on a copy of your Workbench disk. It will go through the most useful of the DISKED commands. The block numbers in the sample session may not match your results exactly. Therefore, you will need to fill in the block numbers DISKED displays on your system.

It is suggested to follow along in the *AmigaDOS Technical Reference Manual.*

NOTE: In this example, everything after a ";" is a comment and should not be typed.

```
HARDCOPY v1.0 recorded on 20-Jun-88 19:14:52 to file "ram:temp"
To end the HARDCOPY session and close the file, type HARDCOPY END

RAM-DISK:> disked df0:
AmigaDOS Disk Editor version 3.4
Write protect mode set, cylinder base set to 0
# r                       ;Find the root block
 Root Block is block 880
# g 880                   ;Get the root block
 Block 880 read (cyl 40, sur 0, sec 0)
# i                       ;Display info about current block
 Short file
 Header key:       0
 Highest seq num:  0
 Data size:        72
 First data block: 0
 Checksum:         -173384115
 Secondary type:   Root block
 Parent Directory: 0
 Hash Chain:       0
 Filename:         "Workbench 1.3"
 Modified:         Monday 20-Jun-88 18:52:18
 Created:          Monday 20-Jun-88 11:45:57
 Bitmap is Invalid
# hc                      ;Get the hash value for the "C:" directory
 Hash value = 14
# 14                      ;Go to (longword) offset 14
# /                       ;Look at what is at the current offset
 889
# g 889                   ;Go to the "C:" directory
 Block 889 read (cyl 40, sur 0, sec 9)
# hEd                     ;Look up the hash of "Ed"
 Hash value = 13
# heD                     ;Case is ignored for hashing
 Hash value = 13
```

```
# 13
# /
 910
# g 910
 Block 910 read (cyl 41, sur 0, sec 8)
# i                     ;Get the info
 Short file
 Header key:       910
 Highest seq num:  41
 Data size:        0
 First data block: 911
 Checksum:         -59184531
 Secondary type:   File
 Extension block : 0
 Parent Directory: 889
 Hash Chain:       1346
 Filename:         "Ed"
 Created:          Monday 20-Jun-88 10:55:40

      ;When two or more names have the same hash value,
      ;they are chained together.  In this case "Hash Chain"
      ;points to the next one in line.

# g 1346                ;Go to next in hash chain
 Block 1346 read (cyl 61, sur 0, sec 4)
# i
 Short file
 Header key:       1346
 Highest seq num:  6
 Data size:        0
 First data block: 1347
 Checksum:         435379822
 Secondary type:   File
 Extension block : 0
 Parent Directory: 889
 Hash Chain:       0
 Filename:         "Binddrivers"
 Created:          Monday 20-Jun-88 19:13:34

# 82                    ;Go to offset 82, where the comment is stored
# sx                    ;Set display to hex
# /
 00000000              ;No comment yet!
# /#X01410000          ;Set a comment as the BCPL string "A"
# /
 01410000              ;Check it, looks ok!
# k
*** warning - checksum incorrect
***         - value in block was -38147475
***         - corrected to       -59184531
```

```
# x
 Write protect mode unset
# p
 Block 910 updated (cyl 41, sur 0, sec 8)
# q                      ;Quit

RAM-DISK:> list df0:c/ed
df0:c/ed                   19564 --p-rwed Today     10:55:40
: A
```

Example 2:

Consider deleting a file that, due to hardware errors, makes the filing system restart process fail.

1. Locate the directory page that holds the reference to the file. Do this by searching the directory structure from the root block, using the hash codes.

2. Locate the slot that references the file. This is either the directory block or a header block on the same hash chain. This slot should contain the key of the file's header block.

3. Set the slot to zero by typing the slot offset, a slash, then zero (<offset>/0).

4. Correct the checksum with the K command.

5. Disable the write protection of the disk with the X command.

6. Write the update block to the disk with the P command or the W (Windup) command.

The blocks that the file used in error will become available again after the RESTART process has successfully scanned the disk.

DRIP

Format: DRIP [<threshold>]

Template: DRIP "THRESHOLD"

Purpose: To reveal unrestrained memory allocation or deallocation.

Specification:

DRIP is used in program development to report unrestrained memory allocation or deallocation. For instance, if a program is suspected of failing to free memory which it has acquired, DRIP can be used to reveal any memory left allocated after the program has been terminated. DRIP should be executed initially to allow it to install itself. (It creates its own message port).

If you specify a value for the <threshold> option, DRIP will only report a loss or gain if it is greater than that value. The <threshold> value should be given in decimal form. If it is not supplied, DRIP assumes the threshold to be zero.

Example:

```
1> DRIP
```

Initialize DRIP; the threshold is zero.

```
1> myprog
```

Run the suspected program. After the program is terminated, run DRIP again.

```
1> DRIP 1024
=============>> drip:  lost 2203 bytes
```

According to DRIP, 2203 bytes of memory were not freed by myprog.

FRAGS

Format: FRAGS [full]

Template: FRAGS "FULL/S"

Purpose: To display system free memory size distribution.

Specification:

FRAGS displays Amiga system free memory size distribution. System free memory is maintained as a linked list of free regions of varying sizes. As memory is dynamically allocated, some larger blocks of free memory may be fragmented (thus the command name FRAGS).

By default, FRAGS shows the number and sizes of the remaining free regions. If the full option is specified, FRAGS instead shows the fragments for each separate allocation pool.

Notes:

1. The FRAGS program itself must load into memory (thus, frag memory).

2. Large blocks are expressed as a sum of their parts. For example, the count for 72 would appear as a 64 and an 8.

Example:

```
1> FRAGS

Free memory size distribution:

Chip Memory, $420

   Size         Count
    8:            6
   16:           12
   32:            8
   64:            5
  128:            5
  256:            2
  512:            1
 1024:            1
 2048:            1
 4096:            0
 8192:            0
16384:            0
32768:            0
65536:            0
131072:           0
262144:           0
```

MOREROWS

Format: MOREROWS [ROWS <number>] [COLUMNS <number>]

Template: MOREROWS "ROWS/K,COLUMNS/K"

Purpose: To increase the width and/or height of the Workbench screen.

Specification:

NOTE: MOREROWS IS UNSUPPORTED AND MAY NOT WORK WITH FUTURE OPERATING SYSTEMS.

MOREROWS increases the width and/or height of the Workbench screen by the amounts specified in the command line.

[ROWS <number>] adds a specified number of rows to the Workbench screen. The legal values for <number> are 0 to 63 for a non-interlaced screen, and 0 to 126 for an interlaced screen.

[COLUMNS <number>] adds a specified number of columns to the Workbench screen. The legal values for <number> are 0 to 64.

After running MOREROWS, you must run Preferences. Selecting the SAVE gadget in the Preferences screen stores the changes specified by MOREROWS (and any other changes made to the Preferences screen). Rebooting your machine will implement the screen dimension changes.

Typing MOREROWS without any arguments results in a display of the current row and column size changes.

Example:

```
1> MOREROWS ROWS 10 COLUMNS 5

1> MOREROWS

Current RowSizeChange 10, ColumnSizeChange 5
```

PRINTA

Format: PRINTA <file> [NOHUNK]

Template: PRINTA "FROM/A,NOHUNK/S"

Purpose: To print the contents of Amiga binary format files.

Specification:

PRINTA prints Amiga binary object files from the assembler (ASSEM) or the linker (ALINK). To understand the meaning of the output of PRINTA, you should read Chapter 2, "AmigaDOS Binary File Structure," of the *AmigaDOS Technical Reference Manual*.

Entering NOHUNK after your file name causes PRINTA to suppress printing the actual contents of code and data hunks.

Example:

```
1> PRINTA startup.obj
HUNK.CODE
23F80004 00000004 23CF0000 00004EB9 00000000 42802E79 00000000
4E75202F 000460F2
HUNK.RELOC32 HUNK=1 [00000018] [0000000A] [00000004]
HUNK.EXT [1] [_exit...]Val=0000001E
HUNK.EXT [129] [_main...] [00000010]
T.END
HUNK.BSS [00000003]
HUNK.EXT [1] [_errno...] Val=00000008
HUNK.EXT [1] [_SysBase] Val=00000004
T.END
```

Examine this print out in parts:

```
HUNK.CODE
23F80004 00000004 23CF0000 00004EB9 00000000 42802E79 00000000
4E75202F 000460F2
```

The file "startup.obj" has a CODE hunk with the data listed in hexadecimal.

```
HUNK.RELOC32 HUNK=1 [00000018] [0000000A] [00000004]
```

The CODE has three values relative to itself (HUNK=1) at offsets 18, A and 4 from the start of the hunk.

```
HUNK.EXT [1] [_exit...]Val=0000001E
```

The CODE has an external reference to itself ([1]) called _exit which is 1E bytes from the start of the hunk.

```
HUNK.EXT [129] [_main...] [00000010]
```

The CODE has an external reference to _main in some other hunk (which must be linked in later before the code will be loadable).

```
HUNK.BSS [00000003]
```

Following the CODE hunk is a BSS (uninitialized data area) hunk. The BSS is 3 long words long (12 bytes).

```
HUNK.EXT [1] [_errno...] Val=00000008
HUNK.EXT [1] [_SysBase] Val=00000004
```

The symbol _errno, which is 8 bytes from the base of the BSS hunk, is referenced by the CODE (hunk [1]). The symbol _SysBase, which is 4 bytes from the base of the BSS hunk, is also referenced by the CODE (hunk [1]).

See Also: ASSEM, ALINK, METACC, ABSLOAD

READ

Format: READ <filename> [SERIAL]

Template: READ "TO/A,SERIAL/S"

Purpose: To read data from the parallel port or serial line and store it in a file.

Specification:

READ listens to the serial or parallel port and expects a stream of hexadecimal characters. (Default is the parallel port; you must use the SERIAL option for READ to listen to the serial port.) Each hex pair is stored as a byte in memory. READ recognizes the ASCII digits 0-9 as well as the capital letters A through F. READ ignores spaces, new lines, and tabs. You must send an ASCII hex digit for every nibble, and you must have an even number of nibbles. When the stream is complete, READ writes the bytes from memory to the disk file you have specified.

You can use READ to transfer either binary or text files as long as the file has been converted into hex characters. (The TYPE command has an H option to produce files of this type. However, EDIT must be used to remove the ASCII listing on the right.)

WARNING: READ overwrites the file specified by the TO filename.

Example:

```
1> READ TO df0:test
```

Reads to the file df0:test from the parallel port.

ROMWACK

Format: ROMWACK

Template: ROMWACK

Purpose: To run the Amiga's ROM-resident debugger.

Specification:

ROMWACK is a small, ROM-resident debugger primarily used for system data structure examination. All communication is through the RS-232-C serial data port at 9600 baud. When ROMWACK is run, the Amiga will be frozen, but ROMWACK does not disturb the system beyond using a small amount of supervisor stack (memory between $200 and $400) and the serial port. Once enabled, interrupts continue, but multitasking is halted.

When ROMWACK is run, a register frame is printed. The frame includes the current processor state as well as the system context from which ROMWACK was involved. All numbers are treated as hexadecimal.

Several key commands are used by ROMWACK:

Relative positioning

.	forward a frame
,	backward a frame
>	forward a word
<	backward a word
+n	forward n bytes
-n	backward n bytes
<RETURN>	redisplay current frame
<SPACE>	forward a word
<BACKSPACE>	backward a word

Execution Control

go	execute from current address
resume	resume execution at current PC address
^D	resume execution at current PC address
^I (<TAB>)	single instruction step
boot	system cold reset
ig	system cold reset

Breakpoints

set	set breakpoint at current address
clear	clear breakpoint at current address
show	show all breakpoint addresses
reset	clear all breakpoints

Other

<hex-address>	set current address to <hex-address>
=	modify current memory word
alter	perform a repeated =; <RETURN> to exit
limit	set current address as the upper bound of find
find <pattern>	find <pattern> in memory, beginning with the current address and ending with the address set by limit; <pattern> may be from one to four bytes in length
fill <pattern>	fill memory with <pattern> from the current address to the upper bound address as set by limit

Example:

```
1> ROMWACK
PC: FC08B8  SR: 0010  USP: 01DB54  SSP: 07FFA  SCPT: 0000  TASK: 01C7D0
DR: 00000000 0000F803 0001DC68 00000000 0000F803 00000008 0001C748 00000000
AR: 000001E8 0001AC30 0000F803 00FF0282 0001DC68 0001DBBC 00000676
SF: 00FF 395A 0001 DC68 00FE F8E4 00FE E$C$ 00FF 3470 00FE FA4A 0002 6B98 0001
<RETURN>
FC08B8 4E75 007C 2000 518F 40D7 2F7C 00FC 08B8 N u.. |  .. Q.. @.. / |....^H..
:20
FC08B8 4E75 007C 2000 518F 40D7 2F7C 00FC 08B8 N u.. |  .. Q.. @.. / |....^H..
FC08C8 0002 3F7c 0020 0006 4ED5 0CAF 00FC 08AA ..^B? |..  ..^F N..^L......^H..
:4
FC08B8 4E75 007C  N u..  |
>
FC08BA 007C 2000 .. |   ..
+24
FC08DE 00FC 08BA ....^H..
```

See Also: For a more detailed explanation of ROMWACK, see the ***Amiga ROM Kernel Manual.***

SETPARALLEL

Format: SETPARALLEL <index> <value>

Template: SETPARALLEL "INDEX/A,VALUE/A"

Purpose: To allow the CLI user to dynamically change any particular parallel port parameter.

Specification:

SETPARALLEL allows the user to change the parameters for the parallel port.

<index> is a decimal number that specifies the function SETPARALLEL is to perform, as follows:

QUERY — Print current parameter values and indexes without changing them.

FLAG — Set ParFlags to <value>. Valid values are:

PARF_SHARED	$20	ParFlags non-exclusive access mask
PARF_RAD_BOOGIE	$08	ParFlags (not yet implemented)
PARF_EOFMODE	$02	ParFlags EOF mode enabled mask

EOF — Set PTermArray to <value>. Value is interpreted as a long word and thus occupies the first 4 bytes of the 8 byte array. The latter 7 bytes are propagated with the low order byte (<value> & $FF). PTermArray is only used when EOF mode is selected. Thereafter, each input character will then be compared against those in the array. If a match if found, the read is terminated.

<value> is the number to which to set the <index> parameter. <value> is decimal unless it starts with X or x, in which case the number is interpreted as hexadecimal. See <index> for determining <value>.

Example:

```
1> SETPARALLEL EOF x51040303

1> SETPARALLEL QUERY
ACTION      FIELD-NAME                      HEXADEC          DECIMAL
flag        io_ParFlags =                        22               34
eof   io_PTermArray.PTermArray0 =  51040303        1359217411
      io_PTermArray.PTermArray1 =  03030303          50529027

      io_Status =                        06               6
```

SHOWLOCKS

Format: SHOWLOCKS <volumename:>

Template: SHOWLOCKS "VOLUME"

Purpose: To print out the access mode and path for every DOS file lock on a particular volume.

Specification:

SHOWLOCKS prints out the access mode and path for every DOS file lock on a particular volume. This is useful for discovering programs that may not clean up properly and, consequently, leave unwanted locks in the system.

Example:

Lock Monitor, Copyright 1988 Charles McManis, prints out outstanding locks on a given volume. (Lock #0 is the one created by this program)

```
1> SHOWLOCKS a:

------------------------
--Number----Access--Path
------------------------
  Lock #0  : -2   'a:'
  Lock #1  : -1   '(exclusive lock):'
  Lock #2  : -1   '(exclusive lock):'
  Lock #3  : -2   'a:devs/printers/Nec_Pinwriter'
  Lock #4  :  0   'a:devs/printers'
  Lock #5  : -2   'a:S'
  Lock #6  : -2   'a:System'
```

SNOOP

Format: SNOOP

Template: SNOOP

Purpose: To report memory allocation and deallocation.

Specification:

SNOOP displays any memory allocations and deallocations made by Amiga tasks or processes. It does so by intercepting calls to the EXEC functions ALLOCMEM and FREEMEM. It then reports any actions to the user through a terminal connected to the serial port.

StripA

Format: StripA [FROM] <sourcefile> [TO] <destinationfile>

Template: StripA "FROM,TO/A"

Purpose: To remove debug and symbol hunks from binary load files.

Specification:

The StripA program is a utility which removes all HUNK_SYMBOLS and HUNK_DEBUG hunks from Amiga binary load files. These hunks, while useful for debugging, are not loaded by the loader. Stripping them from a program can often reduce the size of a program file considerably.

<sourcefile> specifies the name of the binary load file to be stripped.
<destinationfile> specifies the name StripA is to give to the stripped version of the file.

Example:

```
1> StripA FROM myprog.nostrip TO myprog
```

strips the HUNK_SYMBOL and HUNK-DEBUG hunks from the binary load file myprog.nostrip. The stripped version of the file is called myprog.

StripC

Format: StripC <sourcefile> <destinationfile>

Template: StripC "SOURCEFILE,DESTINATIONFILE"

Purpose: To remove comments and extraneous white space characters.

Specification:

StripC (short for Strip Copy) removes comments and white space characters from C language and 680xx assembly language source files. Stripping source files can significantly reduce the amount of time it takes to compile or assemble programs. In particular, stripped .h and .i files, included in many source files for a program, can dramatically decrease compilation/assembly.

<sourcefile> should end in .h (C source) or .i (assembly). If not, StripC will do a straight, unaltered copy of the source file. <destinationfile> can be any legal name, but it must be different from <sourcefile>.

Example:

```
1> StripC intuition.h intuition.strip
```

Index

Several of the Index entries are followed by text in parenthesis. This text represents the directory or program associated with the entry.

Page numbers refer to the primary page where the command or function is explained. The List Of Commands that are included at the end of Chapter 2 (WACK) and Chapter 5 (MEMACS) are not referenced in this index.

Commodore Business Machines, Inc.
1200 Wilson Drive · West Chester, PA 19380
Part No. 363117-01

Commodore Business Machines, Limited
3470 Pharmacy Avenue · Agincourt, Ontario, M1W3G3
Printed in USA 11/88