Macintosh®

MacsBug 6.1 Reference

APPLE COMPUTER, INC.

This manual and the software described in it are copyrighted, with all rights reserved. Under the copyright laws, this manual or the software may not be copied, in whole or part, without written consent of Apple, except in the normal use of the software or to make a backup copy of the software. The same proprietary and copyright notices must be affixed to any permitted copies as were affixed to the original. This exception does not allow copies to be made for others, whether or not sold, but all of the material purchased (with all backup copies) may be sold, given, or loaned to another person. Under the law, copying includes translating into another language or format.

You may use the software on any computer owned by you, but extra copies cannot be made for this purpose.

© Apple Computer, Inc., 1989
20525 Mariani Avenue
Cupertino, CA 95014
(408) 996-1010

Apple, the Apple logo, HyperCard, ImageWriter, LaserWriter, and Macintosh are registered trademarks of Apple Computer, Inc.

Apple Desktop Bus, Extended Discipline, Finder, MPW, MultiFinder, RAMDump, ResEdit and SADE are trademarks of Apple Computer, Inc.

ITC Garamond and ITC Zapf Dingbats are registered trademarks of International
Typeface Corporation.

Microsoft is a registered trademark of Microsoft Corporation.

PostScript is a registered trademark of Adobe Systems Incorporated.

Simultaneously published in the United States and Canada.

Contents

Preface About This Manual

Contents

Overview

Welcome to MacsBug 6.1, Apple's assembly-language debugger for Macintosh® programmers. If you have written, or are trying to write, a program for the Macintosh, you'll find MacsBug a powerful debugger with many unique capabilities. If you aren't actually writing a program, but have a good basic understanding of *Inside Macintosh*, you'll find MacsBug a helpful tool for learning more about the Macintosh. (In fact, MacsBug was used frequently in the writing of *Inside Macintosh* to determine how particular routines actually worked.)

- Chapter 1 provides an overview of MacsBug. This includes a description of the hardware and software configurations MacsBug works with, what kind of debugger MacsBug is, and the files on the MacsBug disk.
- Chapter 2 introduces the MacsBug commands and describes how they fit into various debugging strategies.
- Chapter 3 provides a complete specification of the MacsBug command language, including command syntax, operation, and examples.
- Appendix A contains a summary of all MacsBug commands.
- Appendix B lists the error messages returned by MacsBug.
- Appendix C describes MacsBug internals for advanced programmers.
- Appendix D details how you can call MacsBug from within your program.
- Appendix E explains how to write your own customized debugging commands.
- Appendix F provides tips, shortcuts, and interesting facts about MacsBug.
- Appendix G covers procedure name definition for advanced programmers.

Notation conventions

The following notation conventions are used to describe MacsBug commands:

literal	Plain text indicates a word that must appear in the command exactly as shown. Special symbols (-, §, &, and so on) must also be entered exactly as shown.
variable	Items in italics can be replaced by anything that matches their definition.
[optional]	Square brackets mean that the enclosed elements are optional.
repeated...	An ellipsis (...) indicates that the preceding item can be repeated one or more times.
either | or	A vertical bar (|) indicates an either/or choice.
(grouping)	Parentheses indicate grouping and are used when both items of a choice can be specified and repeated; that is, (param1 | param2 ...).
{ Return }	Brackets are used in examples to indicate that the specified key should be pressed. They are also used to enclose comments.

Command names and filenames are not sensitive to case.

Aids to understanding

Look for these visual cues throughout the manual:

▲ **Warning** Warnings like this indicate potential problems. ▲

△ **Important** Text set off in this manner presents important information. △

◆ *Note:* Text set off in this manner presents notes, reminders, and hints.

Chapter 1 **MacsBug Overview**

Contents

About MacsBug

MacsBug is a Motorola 68000-family assembly-language debugger customized for the entire Macintosh® family of computers. First introduced in 1981, MacsBug has continued to evolve along with the Macintosh.

MacsBug 6.1 runs on the Macintosh Plus, Macintosh SE, and Macintosh II, and supports all members of the 68000 family. It handles the MC68881 floating-point coprocessor and the MC68851 Memory Management Unit (MMU). It also supports external displays on the Macintosh Plus and Macintosh SE, as well as various screen sizes and bit depths on Macintosh II displays. There's no need to customize MacsBug for particular configurations since it determines the attributes of the machine at system startup.

MacsBug 6.1 works with all versions of Macintosh system software, and is compatible with MultiFinder™.

MacsBug 6.1 does not work with the 64K ROMs, nor does it run on the Macintosh XL.

Macintosh debugging

MacsBug uses as little of the Macintosh system software as possible. This lets systems programmers debug their software without having to worry about the debugger using the code they're debugging. But MacsBug isn't only a systems-level debugger. The high degree of interaction between a Macintosh application and the system also makes MacsBug a powerful tool for debugging applications.

MacsBug is an assembly-language debugger. If you're writing programs in a high-level language like C or Pascal, you'll more often want to use the Symbolic Application Debugging Environment (SADE™). SADE lets you debug your program at the source-code level, which means you don't need to know assembly language or map object code back to your program's source-level instructions. If you need to, SADE lets you monitor program execution at the machine level as well.

SADE does have its limitations, however, and high-level programmers will find that MacsBug picks up where SADE leaves off. Specifically:

- SADE uses the Macintosh system software extensively, and in the case of a severe crash may not be operable. MacsBug lets you examine the remains to try to determine what went wrong.
- If RAM is severely limited, you may not be able to run SADE. MacsBug is lean and mean.

MacsBug is loaded at system startup and sits quietly in RAM until it's invoked. Unlike debuggers that expect a target program to work with, MacsBug lets you look at practically anything running on the Macintosh—toolbox and operating-system routines, applications, desk accessories, and so on.

You can suspend program execution at any point, either manually (by pressing the interrupt switch or a key that you define) or programmatically (by calling special traps from within your program). And since MacsBug needs so little of the system to operate, it can be used even in the case of fatal system errors. Whenever the System Error Handler is called, or when a 68000 exception occurs, MacsBug takes control and lets you look around.

Once MacsBug has been invoked, you can enter commands to

- Display and set memory and registers.
- Disassemble memory.
- Set execution breakpoints.
- Step and trace through both RAM and ROM.
- Monitor system traps.
- Display and check the system and application heaps.

The next chapter introduces the MacsBug features and how they fit into various debugging strategies. Chapter 3 provides a complete specification of the MacsBug command language, including command syntax, operation, and examples.

MacsBug files

The MacsBug 6.1 release disk contains the following files and folders:

■ Read Me First	Read this file first; it contains information about the files on the release disk.
■ MacsBug	Copy MacsBug into the System Folder on your boot disk.
■ Debugger Prefs	This file contains macros, templates, and other resources used by MacsBug. Copy this file into the System Folder as well. (If your space is limited, you can omit this file.)
■ Resources	This folder contains additional resources that you can paste into the Debugger Prefs file (using ResEdit™) to provide additional functionality. It also contains another folder, MPW .r Files, that provides the Macintosh Programmer's Workshop (MPW™) text files used to create the MacsBug resources. You can use these files as models for creating your own resources and add them to Debugger Prefs with the Rez tool.
■ dcmds	This folder contains files that you can use to create your own customized debugging commands, as well as samples of such commands. See Appendix E for details.

Chapter 2 **Debugging With MacsBug**

Contents

Getting started

MacsBug is installed at system startup and resides in RAM until shutdown. In order to be recognized at boot time, the MacsBug file must be in the System Folder on the startup disk. If you want the resources contained in the Debugger Prefs file to be loaded, this file must also be in the System Folder. (See "MacsBug resources" later in this chapter for details on editing and adding resources.)

To prevent MacsBug installation indefinitely, you can rename the MacsBug file, or move the file from the System Folder. To override MacsBug installation for a single session only, simply hold down the mouse button during startup.

After a successful installation, the message "MacsBug installed" is displayed below the "Welcome to Macintosh" message. The startup application (typically the Finder™) is then launched.

The simplest way to invoke MacsBug is by pressing the interrupt switch; this generates an NMI exception and suspends program execution. MacsBug takes control and displays a screen like that shown in Figure 2-1.

■ **Figure 2-1** MacsBug display

«INCLUDE fig 1.mac2»

◆ *Note:* Another way to invoke MacsBug is to define an 'FKEY' resource containing the two instructions needed—Debugger ($A9FF) and RTS ($4E75).

To see the application screen again, press the tilde (~) key or the Esc (Escape) key. To return to the MacsBug display, press any character key.

If you have multiple screens, MacsBug uses the "Welcome to Macintosh" screen by default. You'll probably want your application on the larger screen and MacsBug on the smaller screen. To select a different screen for the MacsBug display, press the Option key while clicking on the Monitor icon from the Control Panel, drag the Macintosh icon to the desired screen, and reboot.

The MacsBug display is in black and white only by default. If you prefer to debug in color, see the section "MacsBug resources" later in this chapter.

At the bottom of the MacsBug display is the command line, indicated by a flashing bar cursor. MacsBug accepts the standard editing keys (Delete, Left Arrow, Right Arrow), as well as several special functions:

Command–Left Arrow	Move cursor left one word.
Command–Right Arrow	Move cursor right one word.
Command-Delete	Delete the word to the left of the cursor.
Command-V	Restore previous command line(s) for editing.

Multiple commands, separated by semicolons, can be entered on the command line. To execute the command(s) on the command line, press Return or Enter. Pressing Return without entering a command repeats the last command.

You can use either the Return key or the Space bar as a toggle to pause and resume execution of a command. To cancel the execution of a command, press any other key. (Note, however, that execution cannot then be resumed.)

Thorough on-line help information that includes the syntax of all commands can be displayed with the HELP command.

The largest area of the screen is the output region. MacsBug output falls into three categories, indicated by three levels of indentation:

- The reason for the break. MacsBug tells which 68000 exception, Macintosh system error, or user-specified break caused MacsBug to be invoked.
- Messages. For each command you enter, MacsBug gives a message either confirming execution or explaining a failure.
- Command output.

Output scrolls up (and eventually off) the screen as new commands are executed. You can use the Up Arrow and Down Arrow keys to examine text that has scrolled off the top of the display. This feature is enabled by a buffer whose size—initially 2K—can be modified to suit your needs. (See "MacsBug resources" later in this chapter for details.) An 8K buffer will hold about four pages of output.

If you scroll back to examine some text and then enter another command, the new output is displayed starting from where you are (rather than at the end of the buffer). The rationale behind this behavior is that you'll more often want to see the new output along with the output at which you were just looking.

The LOG command lets you save all MacsBug output to either a file or an ImageWriter® printer.

Immediately above the command line is the PC (program counter) region; it shows the address of the next instruction to be executed, along with the disassembly of that instruction. In the case of a fatal error, it shows the last instruction executed (in other words, the instruction that caused the crash). You can change the number of lines displayed—it's two lines by default—in the PC region; see "MacsBug resources" later in this chapter for details.

The area on the left side of the screen, known as the status region, displays information about the system. At the top is the address contained in the stack pointer (register A7), followed by the bytes at the top of the stack. The number of bytes displayed varies with the screen size and the format of the display. The SHOW command lets you specify the display in word, long word, and ASCII format; it also lets you specify other areas of memory for display.

Below the stack data is the name of the current heap; by default it's the application heap. You can change the current heap with the HX (Heap Exchange) command. The HZ (Heap Zones) command tells you all known heap zones and works with MultiFinder.

The rest of the status region shows the contents of the CPU registers. Several commands give additional register information. The TF (Total Floating-Point) and TM (Total MMU) commands show the contents of the floating-point register and MMU registers respectively. The TD (Total Display) command displays the CPU registers in the output region. Since the CPU registers are constantly updated and displayed in the status region, the TD command is useful for remembering register values between commands.

There are several ways to leave MacsBug. The simplest way is with the G (Go) command; program execution resumes at the current program counter. If MacsBug was invoked due to an unexpected error condition, it may not be possible to resume program execution. Depending on the severity of the error condition, it may be necessary to relaunch the application (EA command), relaunch the shell (ES command), restart the system (RS command), or reboot the machine (RB command).

Specifying things

Most of the MacsBug operations—setting breakpoints, displaying memory, disassembling code—need an actual address to work with. To make life easier, MacsBug provides a number of different alternatives to specifying hard addresses.

Whenever possible, MacsBug accepts and returns symbols in place of addresses. Procedure names are the most common example of this. Most compilers for the Macintosh have the option of embedding character names after the code generated for each procedure or function. (Compiler writers will want to see Appendix G for details on procedure name definition.) If you've used this option, you can specify a procedure name and offset whenever MacsBug wants an address. Conversely, MacsBug returns addresses as offsets from procedures whenever it can. For instance, if the instruction shown in the PC is part of a valid procedure, the PC window gives the name and offset of that instruction.

You can disassemble any of your application's procedures with the IR command.

MacsBug provides a very handy feature for displaying and selecting procedure names. When you press Command-:, a menu showing all procedure names is displayed. You can qualify the names displayed by typing the first few letters in the name. You can then use the arrow keys to move up and down the list. When the name you want is highlighted, press Return and the selected name is inserted into the command line at the insertion point.

If you've qualified the list and want to move back to the previous level of qualification, press the Delete key. To dismiss the menu without making a selection, press the Esc key.

The WH (Where) command provides mapping between symbols and addresses. When given an address or a symbol name, MacsBug gives you the other item.

To translate a symbol name into an address, MacsBug must search the current heap. Since this search process can be slow, MacsBug provides the SX (Symbol Exchange) command for disabling the use of symbol names.

◆ *Note:* Advanced programmers may find themselves dealing with multiple files (code segments, for instance) having the same symbol names. The RN (Resource Number) command lets you restrict symbol matching to a file with a given reference number.

MacsBug supports both 24-bit and full 32-bit addressing modes.

MacsBug also allows the creation of macros. Macros are simple text string substitutions and can be used to create command name aliases, reference global variables, and name common expressions. Macros are expanded before the command line is executed and can thus contain anything you can type in a command line.

You can create macros on the fly with the MC command or include them in a resource file. (See the MC command for details.) The MCD command lists the macros known to MacsBug, and the MCC command clears one or all macros.

How did I get here?

When your program crashes unexpectedly, you'll start with several clues. MacsBug tells you what 68000 exception or system error ID caused the crash. The PC region gives you the instruction that caused the crash. The location of the instruction, whether in ROM or at an offset from a procedure, is also given. You can examine the code immediately preceding the crash by using the IP command.

One approach is to examine the stack for the procedure call history. If your procedures use the LINK A6 procedure prolog, the SC6 command returns the calling history. If they don't use LINK A6, or if you are in a part of the ROM that doesn't use A6 links, you'll need to use the SC7 command. This command finds possible return addresses on the stack. You can use these addresses to examine the stack yourself. You can also use the addresses in other MacsBug commands. Be aware, though, that the SC7 command will almost certainly include old or invalid values (in other words, addresses not in the current calling chain), since local stack variables can change the stack top without changing the contents.

Another way to find out where a program has been is by recording the A-trap calls it makes, using the ATR (A-Trap Record) command. When recording has been turned on, MacsBug records all trap calls in a circular buffer. When the buffer is full, the oldest calls will be overwritten by new calls. You can define the size of the buffer and thereby the number of traps recorded. (See the ATR command for details.) You may want to consider always enabling trap recording; the performance cost isn't very great. To see the information recorded, use the ATP (A-Trap Playback) command.

In the same way that trap recording lets you build a trap history in a buffer, the ATT command lets you direct that history to the screen or to a log file. Tracing to the screen is useful if you have two screens. MacsBug can take over one screen and display the history as your program executes on the other screen. In cases where the program crashes so badly that MacsBug cannot be invoked, you'll still have a trap history available.

Enabling logging with the LOG command and tracing to a file is useful if you want to record a large number of calls and can't afford to dedicate the memory for the trap recording buffer. Another benefit of log files is that you can use your editor to help examine the data.

If you turn on logging after entering MacsBug, the reason for the break is lost. You can use the HOW command to redisplay the reason.

Controlling program execution

MacsBug provides a set of commands that let you control and watch the execution of your program. Two commands let you execute instructions one at a time. The S (Step) command executes a single 68000 instruction, stops at the next instruction, and returns to MacsBug. The contents of the program counter—in other words, the next instruction to be executed—are disassembled and displayed. You can also step through a specified number of instructions, or until a condition is met (for instance, when a register contains a particular value).

When the S command reaches a subroutine or an A-trap call, it steps right in. Particularly with ROM routines, which are often very long and typically not of interest, you'll probably want to use the SO (Step Over) command instead. The SO command works exactly like the S command except that it treats A-trap calls and subroutines as a single instruction, stopping at the first instruction after the A-trap or subroutine returns. (With traps having the auto-pop bit set, MacsBug returns to the address on the top of the stack at the time of the trap call.)

While stepping through code, MacsBug decodes conditional statements (DBcc, Bcc, and Scc instructions) to determine whether branches will be taken or will fall through. This information is shown to the right of the PC information.

If you've stepped into a procedure with the S command and want to get out, you can use the MR (Magic Return) command to move to the end of the procedure. The MR command needs to know where the return address is; for this reason, it's a good idea to use the LINK A6 prolog for your procedures.

If you're stepping through your program and find you want to move past some code, you can use the GT (Go Till) command to resume execution until a specified address is reached.

Stopping at a particular place

Once you've narrowed down the location of a bug, you may want MacsBug to stop when a particular point in your program is reached. There are several ways of doing this.

The ATB (A-Trap Break) command lets you specify a break when A-traps are encountered. You can specify individual traps or a range of traps, as well as conditions that must be met. For instance, you could specify a break when the HFSDispatch trap is encountered and the value of register D0 is 6 (which is the routine selector for the DirCreate routine). You can also specify commands to be executed once MacsBug has been invoked, making life a little easier.

Another way to stop program execution is to set a breakpoint at a specified address, using the BR command. The address can be given as an actual address, or as an offset from a procedure name. This information will have been found by disassembling or stepping through your code. The BR command also lets you specify commands to be executed when the breakpoint is reached. You can specify multiple breakpoints; MacsBug stores this information in a table, which you can see at any time with the BRD command. Breakpoints remain set until you clear them with the BRC command.

You can also set breakpoints by using partial name matching with the BRM command. You pass BRM a sequence of characters; it sets breakpoints on all names that contain those characters. The BRM command is especially useful with C++ debugging; you might, for instance, wish to break on all methods of a given class.

The BR command can be useful in working with A-traps as well as with your own code. With some ROM routines, the actual trap is often preceded by glue code that sets up the parameters. Whereas the ATB command stops right before the trap is made, the BR command can be used to stop at the point where your program calls the routine, letting you examine what goes on with the glue code.

An advantage of using breakpoints is that they don't require changes to your source code and can be used after the application has been built. However, breakpoints cannot be set in a procedure until the segment containing that procedure is loaded and the address determined. One way around this problem is to specify a break from within your procedure by using the traps Debugger ($A9FF) and DebugStr ($ABFF). Debugger is a system trap that invokes MacsBug and displays the message "User break at <addr>." DebugStr additionally lets you supply a custom message for display, as well as MacsBug commands for execution. (For a description of how to declare and use these traps, see Appendix D.)

The DX (Debugger Exchange) command lets you disable breaks from the Debugger and DebugStr traps without having to go in and remove them from your program.

Watching for memory to change

Several commands let you determine when and where a particular area of memory is being changed. One common problem is when a program inadvertently changes the contents of a memory location. You can detect when a range of memory changes by using the SS (Step Spy) command. This command checksums a given range and then executes instructions one at a time until the checksum changes. The SS command can slow down a program considerably, so MacsBug treats a long word as a special case and optimizes for speed. If you suspect a certain range of memory is being altered, you usually don't need to check the whole range but can check just a long word within the range. If you must check a long range, you'll probably want to use a hardware emulator. (You can also use the SS command as a way of slowing down certain routines—those that draw to the screen, for instance—so you can actually watch how they work.)

A variation on the SS command, the ATSS (A-Trap Step Spy) command lets you checksum a memory range before specified A-traps are executed.

The CS (Checksum) command lets you monitor whether a range of memory has changed. The first time you execute the CS command, you specify a range and MacsBug computes a checksum. Subsequent CS commands compute the checksum and compare it with the previous value.

Displaying and setting memory

The DB (Display Byte), DW (Display Word), DL (Display Long), and DP (Display Page) commands display respectively a byte, word, long word, and page (128 bytes) of memory. With the DM (Display Memory) command, you can specify a number of bytes to be displayed. Often you'll want to look at the contents of a data structure consisting of fields of various different sizes. The DM command lets you specify templates for displaying memory in a structured format.

The TMP (Template) command lists the names of all templates known to MacsBug. See the description of this command for instructions on defining your own templates.

The SB (Set Byte), SW (Set Word), and SL (Set Long) commands let you set bytes, words, and long words in memory. The SM (Set Memory) command lets you assign values of varying size; the size of the assignment is determined by the value.

Checking the heap

Several commands let you examine and monitor heap zones. The HD (Heap Dump) command displays information about all blocks in the current heap. To get a summary of the heap allocation, use the HT (Heap Totals) command.

One of the more common bugs is dereferencing a handle to a block that has moved, potentially corrupting the heap. Two commands are useful in detecting this problem. The HC (Heap Check) command checks the current heap and reports any errors. If the problem is reproducible, the ATHC (A-Trap Heap Check) command can be used to check the heap before trap calls.

Exercising your program

It's possible to simulate a worst-case memory situation to exercise your application. The HS (Heap Scramble) command moves all relocatable blocks whenever they might be moved; in other words, whenever the NewPtr, NewHandle, ReallocHandle, SetPtrSize, or SetHandleSize trap is called. (With SetPtrSize and SetHandleSize, the heap is scrambled only if the block size is being increased.)

The DSC command turns on the Extended Discipline™ utility. This program examines parameters before A-traps are called and checks results after the calls complete. If Extended Discipline detects an error, MacsBug is invoked. (See the Extended Discipline manual for details.)

The dot address

MacsBug provides a way of saving and specifying addresses between successive commands; it's so useful that it deserves a separate section.

MacsBug maintains a variable, known as "dot," that contains the last address of interest from certain commands. The period character (.) refers to this address and can be used in any command that expects an address. The commands that set the "dot address" are ones that are often followed by another command using the same address.

Dot is used primarily as a shorthand notation between one command and the next. For instance, you might type `WH name` to find a particular procedure. The WH command sets dot to the address returned, letting you then type `IL .` to disassemble code, or `BR .` to set a breakpoint at the start of the procedure.

Dot can also be used as a placeholder. For instance, the DM (Display Memory) command displays memory starting from a specified address and sets dot to that address. You can resume execution, reenter MacsBug later, and type `DM .` to display the same memory. This technique is useful for watching for an area of memory to change.

The commands that set dot are as follows:

- The commands for displaying memory—DM, DP, DB, DW, and DL—all set dot to the address of the first byte displayed.
- The commands for setting memory—SM, SB, SW, and SL—set dot to the address of the first byte changed. These commands also set the last command to the DM command. This means that after setting memory, you can simply press Return to display the memory just set.
- The WH (Where) command sets dot to the address of the procedure or trap located.
- The F (Find) command sets dot to the first byte of the string that was found.
- The IL (Disassemble From Address), IP (Disassemble Around Address), and ID (Disassemble One Line) commands set dot to the address of the first instruction disassembled.
- The dot address is also used in connection with heap commands. Any command that scans the heap—HD and HC, for instance—can receive a heap error. If the error concerns a particular block (as opposed to the entire heap), MacsBug sets dot to the address of the block header. Typing `DM .` will display the block in question. MacsBug also scans resource maps while examining resource blocks in the heap. Resource map errors set dot to the address of the bad map.

MacsBug resources

The MacsBug release disk includes a variety of resources that configure MacsBug and perform useful functions. The Debugger Prefs file contains the following standard resources:

Type	**ID**	**Contents**
'mxbc'	0	RGB specifications for foreground and background colors
'mxbi'	0	Default configurations: number of traps to record with the ATR command, number of lines to display in the PC area, size of the history buffer.
'mxbm'	100,101	Standard macros; see the MC command for details.
'mxwt'	100	Standard templates; see the TMP command for details.

If you want these resources to be loaded, Debugger Prefs must be in the System Folder on the startup disk.

You can use ResEdit to edit these resources. Debugger Prefs contains a 'tmpl' resource with templates for all of the MacsBug resources. If you have ResEdit version 1.2 or later, these templates are automatically used when you edit a resource. If you have an earlier version of ResEdit, just copy the 'tmpl' resource from Debugger Prefs and paste it into ResEdit.

The Resources folder contains additional resources that you can paste into Debugger Prefs (using ResEdit) to increase functionality:

Name	**Type**	**ID**	**Contents**
C++	'C++ '	0	Code to properly interpret C++ names. (Do not edit this resource.)
KCHR	'KCHR'	0	The key map to use for the MacsBug keyboard. This resource is used to debug international software. (Do not edit this resource.)
Floating Point	'mxbm'	102	Macros to support A-Trap breaks at FP68K calls
HFS Calls	'mxbm'	103	Macros to support A-Trap breaks at HFSDispatch calls
List Manager	'mxbm'	104	Macros to support A-Trap breaks at Pack0 calls

The Resources folder also contains another folder, MPW .r files, that provides the text files used to create all of the MacsBug resources. If you like, you can use these files as models for creating your own resources and add them to Debugger Prefs with the Rez tool.

Chapter 3 **MacsBug Commands**

Contents

Command syntax

MacsBug commands have the following format:

COMMAND required parameters [optional parameters]

Parameters can be numbers, text literals, symbols, or expressions combining these elements. MacsBug provides full command line evaluation, so any parameter can be entered as an expression. The general form of an expression is

value1 [operator value2]

Parentheses can be used to control the order of evaluation. Expressions always evaluate to a 32-bit value unless `.W` or `.B` follows the specified value (in which case the word or byte is sign-extended to 32 bits). Expressions can evaluate to either a numeric or a Boolean value depending on the operators used. The operation of certain commands varies depending on the type of expression. For instance, the BR (Breakpoint) command will break after *n* times if the given expression is numeric, or when a certain condition is met if the expression is Boolean.

Values

Depending on the command, there are a variety of different ways to specify values:

registers	All 68000-family registers use their Motorola names. MMU 64-bit registers and floating-point registers are not allowed in expressions.
numbers	Numbers are hexadecimal by default, but can be preceded by the dollar sign character ($) in the case of conflicts with registers An and Dn. Numbers are decimal if preceded by the number sign character (#). The unary operators "+" and "–" can precede any number, but must appear before the "#" or "$" character.
symbols	Symbols are found by searching the heap, and evaluate to an address. (See Appendix G for details on procedure name definition.) Partial name matching is supported. If you enter `BR My`, for instance, the first symbol starting with *My* is used as the breakpoint address.
traps	A-traps are specified by trap number in the range A000 to ABFF or by trap name. Trap names can be preceded by the dagger character (†) in the case of conflicts with symbol names.

strings	Strings are sequences of characters surrounded by single (') or double (") quotation marks. There is no padding to word or long word boundaries; each character in the string is 1 byte.
.	The period character (.) specifies the dot address; see Chapter 2 for details on using this character.
:	The colon character (:) indicates the address of the start of the procedure shown in the program counter window. This character is not valid if no procedure name exists for PC.

Operators

The operators allowed in expressions, listed in order of precedence from highest to lowest, are given below. Groupings within the table show operators of the same precedence.

(	)	Grouping
@ (prefix)	^ (postfix)	Address indirection
!	NOT	Bitwise or Boolean NOT
*		Multiplication
/	MOD	Division
+		Addition
-		Subtraction
=	==	Equal
<>	!=	Not equal
<		Less than
<=		Less than or equal
>		Greater than
>=		Greater than or equal
&	AND	Bitwise or Boolean AND
||	OR	Bitwise or Boolean OR
XOR		Bitwise or Boolean XOR

◆ *Note*: `@addr` is the same as `addr^`. `Addr^.B` or `addr^.W` fetch only a byte or word from *addr*; the value is then sign-extended to 32 bits.

Command descriptions

This section contains descriptions of all MacsBug commands, arranged alphabetically. For each command, the parameters are given and the operation of the command discussed. Where appropriate, examples are provided. A list of the entire set of commands can be found in Appendix A.

ATB — A-Trap Break

Syntax ATB[A] [*trap* [*trap*]] [*n* | *expr*] ['; *cmds*']

Description The ATB command sets a breakpoint at the specified A-trap(s). Traps can be specified by either trap number or trap name. Appending the letter A to the ATB command tells MacsBug to break only when the given trap is called from the application heap. (Note that this means the current application heap at the time the ATB command was entered.) Specifying two traps indicates a range of traps; MacsBug breaks at every trap encountered within this range. If no traps are specified, a default range of A000 through ABFF is used.

If *n* is specified, MacsBug breaks only after a given trap has been encountered *n* times. If *expr* is specified, MacsBug breaks only when a given trap has been encountered and *expr* is TRUE. If neither *n* nor *expr* is given, MacsBug breaks each time the trap is encountered. You can also supply one or more commands to be executed once the break conditions are satisfied; each command must be preceded by a semicolon and enclosed in quotation marks.

You can set multiple trap breaks with different break conditions or commands. MacsBug checks the table until an entry satisfies the break conditions. The break commands for this entry are executed. Later entries in the table (that also satisfy the break conditions) are ignored.

Be aware that MacsBug stores the information for breakpoints, step commands, and A-trap commands in a single table. New entries are entered at the end of the table. It's possible to receive the error message "Entry will not fit in the table" while entering an ATB command if step commands, BR commands, and other A-trap commands have already filled this table.

Examples

```
ATB                           {break on all traps}
ATB GetNextEvent              {break on GetNextEvent trap}
ATB A000 A010                 {break on traps Open through Allocate}
ATB HFSDispatch D0.W = 6      {break on HFSDispatch when register D0=6 (DirCreate)}
ATB SizeWindow  ';DM (SP+6)^ WindowRecord'  {break on SizeWindow, then display }
                                            { from the contents of the long word }
                                            { 6 bytes above the stack pointer }
                                            { using the WindowRecord template}
```

For a display of the trap table after having set these actions, see the ATD command.

See also ATC, ATD

ATC — A-Trap Clear

Syntax ATC [*trap* [*trap*]]

Description The ATC command clears all actions on the specified traps; in other words, it cancels the ATB, ATT, ATHC, and ATSS commands. Traps can be specified by either trap number or trap name. Specifying two traps indicates a range of traps; MacsBug cancels actions for all traps within this range. If no traps are specified, all trap actions for all traps are cleared.

The ATC command comes in handy when you want to set an action for most, but not all, of the traps in a particular range. For instance, you may think you want to break at all toolbox traps, but soon find that you can do without a break at every call to GetNextEvent. One way around this is to set two ranges around GetNextEvent with the ATB command. An easier way is to set the action on the whole range and use the ATC command to exclude the GetNextEvent trap. Be aware, however, that MacsBug accomplishes this by doing the dirty work for you, itself setting two new ranges around GetNextEvent. This means that, even though you are ostensibly clearing a trap action, you are actually creating an additional entry in the A-trap table and could conceivably receive the error message "Entry will not fit in the table."

Example

Assume the trap table (displayed by using the ATD command) looks like this:

```
 A-Trap actions from System or Application
  Trap Range  Action    Cur/Max or Expression     Commands
  A000  ABFF  Break     00000000 / 00000001
```

After you enter the command

```
ATC GetNextEvent
```

the trap table looks like this:

```
 A-Trap actions from System or Application
  Trap Range  Action    Cur/Max or Expression     Commands
  A000  A96F  Break     00000000 / 00000001
  A971  ABFF  Break     00000000 / 00000001
```

See also ATB, ATD, ATHC, ATSS, ATT

ATD — A-Trap Display

Syntax ATD

Description The ATD command displays the A-trap table(s), which list all actions set with the ATB, ATT, ATHC, and ATSS commands. Two A-trap tables may be displayed, depending on which actions have been set. One table lists all actions restricted to the application heap (using the A parameter), and another lists actions that apply to either the system heap or the application heap.

Both tables have the same format. The trap range for the action is shown in the first column, and the type of action is shown in the second column. If a number of iterations (*n*) was specified with the action, it's shown in the third column, preceded by the actual number of iterations so far. If a logical expression was entered instead, it's shown in the third column. The fourth column shows any commands that were specified for execution upon breaking into MacsBug.

Example

In this example, the following trap breaks were set previously with the ATB command:

```
ATB
ATB GetNextEvent
ATB A000 A010
ATB HFSDispatch D0.W = 6
ATB SizeWindow  ';DM (SP+6)^ WindowRecord'
```

The trap table displayed by the ATD command is given below. Note that traps are represented by trap number; you can determine the corresponding trap name by using the WH command.

```
ATD
 A-Trap actions from System or Application
  Trap Range  Action    Cur/Max or Expression       Commands
  A000  ABFF  Break     00000000 / 00000001
     A970     Break     00000000 / 00000001
  A000  A010  Break     00000000 / 00000001
     A060     Break     D0.W = 6
     A91D     Break     00000000 / 00000001         ;DM (SP+6)^ WindowRecord
```

See also ATB, ATC, ATHC, ATSS, ATT, WH

ATHC — A-Trap Heap Check

Syntax ATHC[A] [*trap* [*trap*]] [*n* | *expr*]

Description The ATHC command checks the consistency of the heap before executing the specified A-trap(s). If the heap is found to have been corrupted, MacsBug is invoked and an error is displayed; see the HC command for a list of possible errors.

Traps can be specified by either trap number or trap name. Appending the letter A to the ATHC command tells MacsBug to check only when the given trap is called from the application heap. (Note that this means the current application heap at the time the ATHC command was entered.) Specifying two traps indicates a range of traps; MacsBug checks for every trap encountered within this range. If no traps are specified, a default range of A000 through ABFF is used.

△ **Important** If you don't specify a trap range, be aware that the Memory Manager makes trap calls—StripAddress, for instance—while moving heap blocks around. In such cases, ATHC will report an error because the heap is inconsistent, albeit temporarily. You can avoid these unnecessary breaks by clearing the action on the offending trap—`ATC StripAddress`, for example. △

If *n* is specified, MacsBug checks only after a given trap has been encountered *n* times. If *expr* is specified, MacsBug checks only when a given trap has been encountered and *expr* is TRUE. If neither *n* nor *expr* is given, MacsBug checks each time the trap is encountered.

See also ATC, ATD, HC

ATP — A-Trap Playback

Syntax ATP

Description The ATP command plays back the information saved while trap recording is on. (For details on trap recording, see the ATR command.) This information includes the trap name and the contents of the program counter (PC). For operating-system traps, the values of registers A0 and D0 are shown, as well as the 8 bytes pointed to by register A0. For toolbox traps, ATP shows the value of register A7 and the 12 bytes to which it points.

Example

In the example below, SetPort is the most-recently executed trap.

```
ATP
 Trap calls in the order in which they occurred
  A030 OSEventAvail
    PC = 004C7346
    A0 = 004871F8  003A 1F34 2000 004B   D0 = 0000FFFF
  A970 GetNextEvent
    PC = 004C2BCA
    A7 = 0048724C  0048 7290 FFFF 0020 156A 3447
  A030 OSEventAvail
    PC = 004C7346
    A0 = 004871DC  004C 16B4 004D 013C   D0 = 00000000
  A031 GetOSEvent
    PC = 004C7334
    A0 = 00487290  0000 0000 0000 000A   D0 = 00000000
  A9B4 SystemTask
    PC = 004C2800
    A7 = 004871F0  0000 0000 0000 FFFF 0048 7290
  A874 GetPort
    PC = 40815150
    A7 = 004871C4  0048 71C8 0048 7290 4598 3427
  A924 FrontWindow
    PC = 40815154
    A7 = 004871C4  0000 0000 0000 0000 003A 1D40
  A873 SetPort
    PC = 408151AE
    A7 = 004871C8  003A 1D40 004D 013C 1DD3 3F6A
```

See also ATR

ATR — A-Trap Record

Syntax

ATR[A] [ON | OFF]

Description

The ATR command turns trap recording on and off; if no parameter is passed, the command toggles between modes. Trap recording saves information about the *n* most recently executed traps. By default, MacsBug records the last 16 traps. You can, however, specify any number by modifying the 'mxbi' resource in the Debugger Prefs file. Since the traps are saved in a circular buffer, space is the only penalty for recording more traps; time is not a factor.

Appending the letter A to the ATR command tells MacsBug to record information only for traps called from the application heap. (Note that this means the current application heap at the time the ATR command was entered.)

The information saved, which can be displayed with the ATP command, includes the trap name and the contents of the program counter (PC). For operating-system traps, the values of registers A0 and D0 are saved, as well as the 8 bytes pointed to by register A0. For toolbox traps, ATR saves the value of register A7 and the 12 bytes to which it points.

See also

ATP

ATSS — A-Trap Step Spy

Syntax

ATSS[A] [*trap* [*trap*]] [*n* | *expr*], *addr1* [*addr2*]

Description

The ATSS command calculates the checksum for the given memory range before executing the specified A-trap(s). If the checksum value changes, MacsBug is invoked. If *addr2* is omitted, ATSS waits for the long word at *addr1* to change. The ATSS command is optimized for speed with a long word; longer checksum ranges can be slow.

Traps can be specified by either trap number or trap name. Appending the letter A to the ATSS command tells MacsBug to check only when the given trap is called from the application heap. (Note that this means the current application heap at the time the ATSS command was entered.) Specifying two traps indicates a range of traps; MacsBug checks for every trap encountered within this range. If no traps are specified, a default range of A000 through ABFF is used.

If *n* is specified, MacsBug checks only after a given trap has been encountered *n* times. If *expr* is specified, MacsBug checks only when a given trap has been encountered and *expr* is TRUE. If neither *n* nor *expr* is given, MacsBug checks each time the trap is encountered.

See also

ATC, ATD, SS

ATT — A-Trap Trace

Syntax

ATT[A] [*trap* [*trap*]] [*n* | *expr*]

Description

The ATT command displays information about the execution of the specified A-trap(s). Traps can be specified by either trap number or trap name. Appending the letter A to the ATT command tells MacsBug to display information only when the given trap is called from the application heap. (Note that this means the current application heap at the time the ATT command was entered.) Specifying two traps indicates a range of traps; MacsBug displays information for every trap encountered within this range. If no traps are specified, a default range of A000 through ABFF is used.

If *n* is specified, MacsBug displays information only after a given trap has been encountered *n* times. If *expr* is specified, MacsBug displays information only when a given trap has been encountered and *expr* is TRUE. If neither *n* nor *expr* is given, MacsBug displays information each time the trap is encountered.

Example

```
ATT HideCursor
 HideCursor    PC = 0000A6F8      D0 = 0057007D    A7 = 004A6E00
```

See also

ATC, ATD

BR — Breakpoint

Syntax BR *addr* [*n* | *expr*] [';*cmds*']

Description The BR command sets a breakpoint at the specified address. If *n* is specified, MacsBug breaks only after *addr* has been reached *n* times. If *expr* is specified, MacsBug breaks only when *addr* has been reached and *expr* is TRUE. If neither *n* nor *expr* is given, MacsBug breaks each time *addr* is reached. You can also supply one or more commands to be executed once the break conditions are satisfied; each command must be preceded by a semicolon.

Entering BR without any parameters displays the breakpoint table, a list of all breakpoints in the order in which they were set; see the description of the BRD command for details.

▲ **Warning** You should be sure that the given address contains an instruction. MacsBug implements breakpoints by placing a TRAP #F instruction in the word at addr. If addr points to the middle of an instruction, the substituted TRAP #F instruction will be treated as part of the instruction, possibly causing an error. ▲

Be aware that MacsBug stores the information for breakpoints, step commands, and A-trap commands in a single table. New entries are entered at the end of the table. It's possible to receive the error message "Entry will not fit in the table" while entering a BR command if step commands, A-trap commands, and other BR commands have already filled this table.

If you set a breakpoint in a relocatable block, MacsBug stores the breakpoint as a handle to the breakpoint address. This means that if the block moves, the breakpoint is updated automatically.

△ **Important** Setting a breakpoint at a ROM address will cause execution to be slow, because MacsBug must trace through each instruction until the breakpoint address is reached. △

Examples

```
BR TestProc+10                 {break when TestProc+10 is reached}
BR TestProc+20 3               {break when TestProc+20 is reached 3 times}
BR TestProc+30 D0 = 1          {break when TestProc+30 is reached and register D0=1}
BR TestProc+40 A0 <> 0 ';DM A0 40'    {break when TestProc+40 is reached and }
                                      { register A0 is not equal to 0; then display }
                                      { memory at address in A0 for 40 bytes}
```

For a display of the breakpoint table with these breakpoints set, see the BRD command.

See also BRC, BRD, BRM

BRC — Breakpoint Clear

Syntax	BRC [*addr*]
Description	The BRC command clears the specified breakpoint; if no parameters are specified, all breakpoints are cleared.
See also	BR, BRD, BRM

BRD — Breakpoint Display

Syntax BRD

Description The BRD command displays the breakpoint table, a list of all breakpoints in the order in which they were set.

If the BR command that set a breakpoint specified a break only after reaching the address *n* times, *n* is shown in the third column, preceded by the number of times the address has been reached so far. If an expression was entered instead, it's shown in the third column. The fourth column shows any commands that were specified for execution upon breaking into MacsBug.

◆ *Note:* MacsBug implements the GT command by setting a temporary breakpoint. If you enter MacsBug by some other means and execute the BRD command, this breakpoint remains set and you'll see an entry for it in the breakpoint table.

In the example below, the following breakpoints were set with the BR command:

```
BR TestProc+10
BR TestProc+20 3
BR TestProc+30 D0 = 1
BR TestProc+40 A0 <> 0 ';DM A0 40'
```

Example

```
BRD
 Breakpoint table
  Address  Module name               Cur/Max or Expression  Commands
  004635E0 TestProc+10               00000000 / 00000001
  004635F0 TestProc+20               00000000 / 00000003
  00463600 TestProc+30               D0 = 1
  00463610 TestProc+40               A0 <> 0                ;DM A0 40
```

See also BR, BRC, BRM

BRM — Multiple Breakpoints

Syntax BRM *name*

Description The BRM command lets you set breakpoints using partial name matching. You pass BRM a sequence of characters; it sets breakpoints on all names that contain those characters. C++ programmers, for instance, can set breaks on all classes with a given method name or on all methods with a given class name.

Examples

```
BRM 'TParseNode::'
```

This example will break on all methods in the class TParseNode.

```
BRM '::Draw'
```

This example will break on all classes that contain a Draw method.

In both examples, the double colons qualify the names using the C++ syntax. (If the double colons were omitted—for instance, `BRM Draw`—BRM would set breaks on all names containing the characters "Draw.") When the double colon syntax is used, the entire string must be enclosed in quotation marks (since the colon has its own predefined meaning in MacsBug).

See also BR, BRC, BRD

CS — Checksum

Syntax

CS [*addr1* [*addr2*]]

Description

The CS command computes a checksum for the memory range from *addr1* through *addr2* and saves the result. If *addr2* is omitted, CS checksums the long word at *addr1*.

Subsequent CS commands without parameters recompute the checksum and compare it with the previous value. If no address range has been previously specified, entering CS without parameters will return the error message "Address range must be entered before comparisons."

DB — Display Byte

Syntax DB [*addr*]

Description The DB command displays the byte at the specified address. If *addr* is omitted, DB displays the byte at the dot address. Pressing Return displays the next byte. The dot address is always set to the address of the byte displayed.

Example

```
DB 0
{Return}

 Byte at 00000000 = $40     64      64    '@'
 Byte at 00000001 = $81    129    -127    '•'
```

See also DL, DM, DP, DW

DH — Disassemble Hexadecimal

Syntax DH *expr*...

Description The DH command disassembles the given expressions as a sequence of 16-bit opcodes. This command is useful in converting hexadecimal values to assembler mnemonics.

Example

```
DH 4E56 0000

 Disassembling hex value
           00308AB6   LINK       A6,#$0000                         | 4E56 0000
```

DL — Display Long

Syntax DL [*addr*]

Description The DL command displays the long word at the specified address. If *addr* is omitted, DL displays the long word at the dot address. Pressing Return displays the next long word. The dot address is always set to the address of the long word displayed.

Example

```
DL 0
{Return}

 Long at 00000000 = $40810000    1082195968    1082195968    '@•••'
 Long at 00000004 = $40802A14    1082141204    1082141204    '@•*•'
```

See also DB, DM, DP, DW

DM — Display Memory

Syntax DM [*addr* [*n* | *template* | *basic type*]]

Description The DM command displays memory starting from the specified address and continuing for *n* bytes. If *n* is omitted, 16 bytes are displayed. If both *addr* and *n* are omitted, DM displays 16 bytes beginning at the dot address. Pressing Return displays the next 16 bytes. The dot address is always set to the address of the first byte displayed.

Instead of specifying a number of bytes, you can specify the name of a template or one of the basic types used in creating a template. See the TMP command for details.

Examples

```
DM 0

 Displaying memory from 0
  00000000  4081 0000 4080 2A14  004F 6306 4080 20FC  @•••@•*••Oc•@• •
```

Note that the centered dot character (•) represents nonprintable characters. In the next example, `windowList` is a macro defining a low memory global variable, and `WindowRecord` is a template.

```
DM windowList^ WindowRecord

 Displaying WindowRecord at 003A0B14
  003A0B24  portRect         0000 0000 01B3 027A
  003A0B2C  visRgn           003A3E88
  003A0B30  clipRgn          003A4570
  003A0B80  windowKind       0045
  003A0B82  visible          TRUE
  003A0B83  hilited          TRUE
  003A0B84  goAwayFlag       TRUE
  003A0B85  spareFlag        TRUE
  003A0B86  strucRgn         003A4584
  003A0B8A  contRgn          003A4598
  003A0B8E  updateRgn        003A45AC
  003A0B92  windowDefProc    20832A5C
  003A0B96  dataHandle       003A6154
  003A0B9A  titleHandle      HD:Examples
  003A0B9E  titleWidth       0052
  003A0BA0  controlList      003A4610
  003A0BA4  nextWindow       003A05E8
  003A0BA8  windowPic        NIL
  003A0BAC  refCon           003A07E0
```

See also DB, DL, DP, DW

DP — Display Page

Syntax DP [*addr*]

Description The DP command displays a page, or 128 bytes, of memory, starting from the specified address. If *addr* is omitted, DP displays bytes beginning at the dot address. Pressing Return displays the next 128 bytes. The dot address is always set to the address of the first byte displayed.

Example

```
DP 0

 Displaying memory from 0
  00000000  4081 0000 4080 2A14  004F 6306 4080 20FC  @•••@•*••Oc•@• •
  00000010  4080 20FE 4080 2100  4080 2102 4080 2104  @• •@•!•@•!•@•!•
  00000020  4080 2106 4080 2108  4080 64BA 4080 210C  @•!•@•!•@•d•@•!•
  00000030  4080 210E 4080 210E  4080 210E 4080 210E  @•!•@•!•@•!•@•!•
  00000040  4080 210E 4080 210E  4080 210E 4080 210E  @•!•@•!•@•!•@•!•
  00000050  4080 210E 4080 210E  4080 210E 4080 210E  @•!•@•!•@•!•@•!•
  00000060  4080 210E 0000 B010  4080 622E 4080 622E  @•!•••••@•b.@•b.
  00000070  4080 60D0 4080 612C  004D 0456 004D 0456  @•`•@•a,•M•V•M•V
```

See also DB, DL, DM, DW

DSC — Extended Discipline

Syntax DSC [ON | OFF]

Description The DSC command turns the Extended Discipline utility on and off; if no parameter is passed, the command acts as a toggle. This utility examines parameters before traps are called and checks results after the calls complete. If Extended Discipline detects an error, MacsBug is invoked. See the Extended Discipline manual for more details.

DV — Display Version

Syntax DV

Description The DV command displays the version number of MacsBug currently in use.

Example

```
DV
 MacsBug version 6.1
 Copyright Apple Computer, Inc. 1981-1989
```

DW — Display Word

Syntax DW [*addr*]

Description The DW command displays the word at the specified address. If *addr* is omitted, DW displays the word at the dot address. Pressing Return displays the next word. The dot address is always set to the address of the word displayed.

Example

```
DW 0
{Return}

 Word at 00000000 = $4081    16513    16513   '@•'
 Word at 00000002 = $0000        0        0   '••'
```

See also DB, DL, DM, DP

DX — Debugger Exchange

Syntax DX [ON | OFF]

Description By default, two traps, Debugger ($A9FF) and DebugStr ($ABFF), let you enter MacsBug from within your program. The DX command lets you turn these "user breaks" on and off; without parameters, it acts as a toggle.

◆ *Note:* Even when user breaks are disabled, messages specified by DebugStr will still be displayed; commands associated with DebugStr, however, are ignored. Also, the DX command does not affect breakpoints, exceptions, or other A-traps.

EA — Exit to Application

Syntax	EA
Description	The EA command attempts to launch the current application again. The current application heap is freed and reallocated.
See also	ES

ES — Exit to Shell

Syntax ES

Description The ES command allows you to exit from the current application. It executes the ExitToShell trap, which launches the current shell (typically the Finder).

◆ *Note:* The ES command may not work with applications that override system traps. ExitToShell initializes the application heap, usually destroying any system patches located there.

See also EA

F — Find

Syntax F *addr n expr* | '*string*'

Description The F command searches the range *addr* to *addr* + *n* −1 for the specified pattern. When passing a string, be aware that case is significant. If *expr* is given, the width of the pattern is the smallest unit (byte, word, or long word) that will contain the value. Pressing Return repeats the search for the next *n* bytes. The F command sets the dot address to the first byte of the pattern found.

In the example below, the string isn't found the first time. Pressing Return repeats the command and finds it. The dot address is set to 2E1.

Example

```
F 0 200 'Finder'
{Return}

 Searching for 'Finder' from 00000000 to 000001FF
  Not found
 Searching for 'Finder' from 00000200 to 000003FF
  000002E1  4669 6E64 6572 2020  2020 2020 2020 2000  Finder        •
```

G — Go

Syntax G [*addr*]

Description The G command is used to leave MacsBug and resume program execution. This command is most frequently used without an address to resume execution where you left off; in other words, at the current program counter. If *addr* is given, execution resumes at that address.

Command-G is provided as a shortcut. Note that any commands sitting in the command line are ignored.

See also GT, MR

GT — Go Till

Syntax

GT *addr*

Description

The GT command sets a breakpoint at *addr* and resumes execution until the program counter reaches that address.

△ **Important** Setting a breakpoint at a ROM address will cause execution to be slow, because MacsBug must trace through each instruction until the breakpoint address is reached. △

◆ *Note:* MacsBug implements the GT command by setting a temporary breakpoint. If you enter MacsBug by some other means, this breakpoint remains set. (In fact, you can see an entry for it in the breakpoint table if you enter the BRD command.) Executing the G command will resume execution until the breakpoint is reached or another exception occurs.

MacsBug 6.1 comes with a predefined macro `'GTO'` that expands to `'GT :+'`. This macro is useful for executing code until an offset in the current procedure. For instance, typing `GTO 22` expands to `GT :+22`, with the colon expanding to the current procedure name.

See also

G, MR

HC — Heap Check

Syntax HC

Description The HC command checks the consistency of the current heap and reports any errors. Heap integrity cannot be checked rigorously, but is examined for certain telltale signs of corruption. The possible error conditions are given below.

Note that all the heap commands check the heap as they execute; if a heap error is detected, they cancel the operation and return the same error message that the HC command would return.

"Zone pointer is bad": The zone pointer for the current heap (SysZone, ApplZone, or user address) must be even and in RAM. In addition, the bkLim field of the header must be even and in RAM, and must point after the header.

"Free master pointer list is bad": Free master pointers in the heap are chained together, starting with the hFstFree field in the zone header and terminated by a NIL pointer.

"BkLim does not agree with heap length": Walking through the heap block by block must terminate at the start of the trailer block, as defined by the bkLim field of the zone header.

"Block length is bad": The block header address plus the block length must be less than or equal to the trailer block address. Also, the trailer block must be a fixed length.

"Nonrelocatable block: Pointer to zone is bad": Block headers of nonrelocatable blocks must contain a pointer to the zone header.

"Relative handle is bad": The relative handle in the header of a relocatable block must point to a master pointer.

"Master pointer does not point at a block": The master pointer for a relocatable block must point at a block in the heap.

"Free bytes in heap do not match zone header": The zcbFree field in the zone header must match the total size of all the free blocks in the heap.

See also ATHC, HD

HD — Heap Display

Syntax HD [*qualifier*]

Description The HD command displays information about blocks in the current heap. The following qualifiers can be specified:

F:	Free blocks
N:	Nonrelocatable blocks
R:	Relocatable blocks
L:	Locked blocks
P:	Purgeable blocks
RS:	Resource blocks
TYPE:	Resource blocks of this type only

If no qualifier is specified, information about all blocks is displayed. If you specify F, N, or R, MacsBug checks the tag byte of the block headers for blocks with the appropriate bit set. If you specify L, P, or RS, MacsBug checks the master pointers for blocks with the lock, purge, or resource bits set. (For more details, see the Memory Manager chapter of *Inside Macintosh.*)

You can also request information about resource blocks of a particular resource type only (for instance, 'CODE', 'CRSR', and so on) simply by specifying the type. It's not necessary to quote the resource type, unless you want MacsBug to distinguish between uppercase and lowercase characters.

If no blocks of a specified type are found, the HD command returns the message "No blocks of this type found."

An example of the heap display is provided at the end of this command description.

For each block, the first column (`Start`) of the display gives the start of the data in the block, and the second column (`Length`) gives the length of the block, not including the header. Blocks that cannot be moved (nonrelocatable or locked) are indicated by a centered dot character (•) before the start address.

The third column (`Tag`) indicates the status of the block as free (F), nonrelocatable (N), or relocatable (R). For relocatable blocks, the fourth column contains the master pointer, while the fifth and sixth columns indicate whether the block is locked (L) or purgeable (P).

For resource blocks, the resource type, resource ID, file reference number and resource name (if specified) are shown.

Example

```
HD

 Displaying the Application heap
     Start     Length   Tag  Mstr Ptr  Lock Purge  Type   ID   File  Name
  •0046321C  00000100   N
  •00463324  00000004   R   00463318   L
  •00463330  00000070   R   0046330C   L          CODE  0001  0294  Main
   004633A8  00000008   F
   004633B8  00000058   R   00463310
   00463418  00000078   R   00463314
   00463498  00000018   R   00463308         P    CODE  0000  0294
   004634B8  00000004   R   00463304
   004634C4  00001518   F
```

See also HC

HELP — Help

Syntax HELP [*cmd* | *section*]

Description The HELP command displays information about the given command or section. If no parameter is passed, a list of section headings is displayed. Pressing Return displays each section in turn.

◆ *Note:* The HELP information is contained in an 'mxbh' resource that's approximately 10K in size. If space is especially tight, you can use ResEdit to remove this resource from the MacsBug file, thereby disabling the HELP command. Do not ever modify this resource, however, because the HELP command expects the information in a particular format.

Examples

```
HELP

 Return shows sections sequentially. "HELP name" shows that section.
  Editing
  Selecting procedure names
  Expressions
  Values
  Operators
  Flow control
  Breakpoints
  A-traps
  Disassembly
  Heaps
  Symbols
  Stack
  Memory
  Registers
  Macros
  Miscellaneous
  dcmds

HELP Stack

 Stack
  SC6 [addr]
     Show the calling chain based on A6 links. If no addr then the
     chain starts with with A6. If addr then the chain starts at addr.
  SC7
     Show possible return addresses on the stack. A return address is
     an even address that points after a JSR, BSR or A-trap.
```

HOW — Display Break Message

Syntax HOW

Description The HOW command redisplays the break message for the current entry into MacsBug. The HOW command is useful if you're logging to a file and want to record the reason for the break. You might include HOW in a macro of the form

```
'LOG filename;HOW;TD;TF;DM SP 100'
```

See also LOG

HS — Heap Scramble

Syntax HS [*addr*]

Description The HS command toggles heap scrambling on or off. When heap scrambling is on, all relocatable blocks in the heap will be moved (if possible) whenever one of the following traps is encountered: NewPtr, NewHandle, ReallocHandle, SetPtrSize, or SetHandleSize. With SetPtrSize and SetHandleSize, the heap is scrambled only if the block size is being increased.

The only blocks not moved are single blocks between two stationary blocks. The heap is checked before scrambling; if it has been corrupted, MacsBug breaks and reports the error. (See the HC command for a list of possible errors.) Heap scrambling is automatically turned off when a bad heap is detected.

You can specify the address of the heap to be scrambled; if you don't, the address contained in the global variable ApplZone (the beginning of the application heap) is used.

See also HC

HT — Heap Totals

Syntax HT

Description For the current heap, the HT command displays the total number of each type of block, the heap size, and the free space in the heap.

Example

```
HT

 Totaling the Application heap
                                Total Blocks     Total of Block Sizes
  Free                          0038       56    00007472       29810
  Nonrelocatable                0009        9    00046236      287286
  Relocatable                   0022       34    0000E650       58960
    Locked                      0004        4    0000257E        9598
    Purgeable and not locked    0003        3    000001CA         458
  Heap size                     0063       99    0005BCF8      376056
```

See also HD

HX — Heap Exchange

Syntax

HX [*addr*]

Description

The HX command sets the current heap for the other heap commands. The address of a heap zone can be specified by ***addr***. If no parameter is specified, the HX command cycles between the application heap, the system heap, and any other heap specified by a previous HX command.

◆ *Note:* The name (or address) of the current heap is shown in the status region of the MacsBug display.

See also

HC, HD, HT, HZ

HZ — Heap Zones

Syntax HZ

Description In a system running MultiFinder, there will be an application heap for each application. The HZ command displays the addresses of all known heap zones. It identifies heaps by doing a heap check on each block in the MultiFinder heap; if the block passes, it's assumed to be a heap. The HZ command will not display heap zones stored on the stack or in the system heap, nor will it find heap zones that don't start at the beginning of a heap block.

See also HC, HD, HT, HX

ID — Disassemble One Line

Syntax ID [*addr*]

Description The ID command disassembles one line, starting at the specified address. If *addr* is omitted, the program counter is used. Pressing Return disassembles the next line. The dot address is set to the address specified.

See also IL, IP, IR

IL — Disassemble From Address

Syntax IL [*addr* [*n*]]

Description The IL command disassembles *n* lines, starting at the specified address. If *addr* is omitted, the program counter is used. If *n* is omitted, half a screen of code is displayed. Pressing Return disassembles the next *n* lines (if *n* was specified initially) or the next half-screen (if *n* was omitted). The dot address is always set to the address specified.

The procedure name and offsets are given in the first column, followed by the actual addresses. A centered dot character (•) after the address indicates that a breakpoint is set at that instruction. The next two fields contain the opcode and operand; an asterisk character (*) before the opcode indicates the current PC.

The comment field (;) gives the target of a JMP, JSR, or BSR instruction, the trap number of a trap, or the ASCII value of a DC statement. The last field shows the actual hexadecimal words of the instruction; if there are too many words, an ellipsis (...) is shown. Note that this last field is shown only on larger displays, but can be always be seen by sending the output to a file or printer with the LOG command.

Example

```
IL

 Disassembling from 00308A96
  Main
    +000C  00308A96  *JSR     PROCATLEVEL1+0000       ; 00308A6A   | 4EBA FFD2
    +0010  00308A9A   JSR     *+$0312                 ; 00308DAC   | 4EBA 0310
    +0014  00308A9E • JSR     *+$0314                 ; 00308DB2   | 4EBA 0312
    +0018  00308AA2   RTS                                          | 4E75
    +001A  00308AA4   UNLK    A6                                   | 4E5E
    +001C  00308AA6   RTS                                          | 4E75
           00308AB4   DC.W    $0000                   ; '••'       | 0000
  _RTInit
    +0000  00308AB6   LINK    A6,#$0000                            | 4E56 0000
    +0004  00308ABA   MOVEM.L D3/D6/D7/A3/A4,-(A7)                 | 48E7 1318
    +0008  00308ABE   MOVE.L  $0018(A6),D6                         | 2C2E 0018
    +000C  00308AC2   JSR     $002A(A5)                            | 4EAD 002A
    +0010  00308AC6   MOVEA.L -$00AA(A5),A0                        | 206D FF56
    +0014  00308ACA   MOVE.L  $0008(A6),(A0)                       | 20AE 0008
    +0018  00308ACE   MOVEQ   #$01,D0                              | 7001
    +001A  00308AD0   MOVEA.L #$00000316,A0                        | 207C 0000 0316
    +0020  00308AD6   TST.L   (A0)                                 | 4A90
    +0022  00308AD8   BEQ.S   _RTInit+005A            ; 00308B10   | 6736
    +0024  00308ADA   MOVEA.L #$00000316,A0                        | 207C 0000 0316
    +002A  00308AE0   MOVEQ   #$01,D1                              | 7201
```

See also ID, IP, IR

IP — Disassemble Around Address

Syntax IP [*addr*]

Description The IP command displays half a screen of disassembled code, centered around the instruction specified by *addr*. Pressing Return disassembles the next half-screen. If *addr* is omitted, the program counter is used. The dot address is set to the first address displayed.

The procedure name and offsets are given in the first column, followed by the actual addresses. A centered dot character (•) after the address indicates that a breakpoint is set at that instruction. The next two fields contain the opcode and operand; an asterisk character (*) before the opcode indicates the current PC.

The comment field (;) gives the target of a JMP, JSR, or BSR instruction, the trap number of a trap, or the ASCII value of a DC statement. The last field shows the actual hexadecimal words of the instruction; if there are too many words, an ellipsis (...) is shown. Note that this last field is shown only on larger displays, but can be always be seen by sending the output to a file or printer with the LOG command.

Example

```
IP

 Disassembling from 00308A7C
  No procedure name
           00308A7C   ADDQ.W     #$2,A4                                 | 544C
           00308A7E   DC.W       $4556                   ; ????         | 4556
           00308A80   DC.W       $454C                   ; ????         | 454C
           00308A82   MOVE.W     D0,-(A0)                               | 3100
           00308A84   DC.W       $0000,$4EBA                            | 0000 4EBA
           00308A88   DC.W       $02FE                   ; ????         | 02FE
  Main
     +0000 00308A8A   LINK       A6,#$0000                              | 4E56 0000
     +0004 00308A8E   MOVEA.L    (A7)+,A6                               | 2C5F
     +0006 00308A90   JSR        *+$02F8                 ; 00308D88     | 4EBA 02F6
     +000A 00308A94   _Debugger                          ; A9FF         | A9FF
     +000C 00308A96  *JSR        PROCATLEVEL1+0000       ; 00308A6A     | 4EBA FFD2
     +0010 00308A9A   JSR        *+$0312                 ; 00308DAC     | 4EBA 0310
     +0014 00308A9E • JSR        *+$0314                 ; 00308DB2     | 4EBA 0312
     +0018 00308AA2   RTS                                               | 4E75
     +001A 00308AA4   UNLK       A6                                     | 4E5E
     +001C 00308AA6   RTS                                               | 4E75
           00308AB4   DC.W       $0000                   ; '••'         | 0000
  _RTInit
     +0000 00308AB6   LINK       A6,#$0000                              | 4E56 0000
     +0004 00308ABA   MOVEM.L    D3/D6/D7/A3/A4,-(A7)                   | 48E7 1318
```

See also ID, IL, IR

IR — Disassemble Until End of Procedure

Syntax IR [*addr*]

Description The IR command disassembles code beginning from the instruction specified by *addr*; if no address is given, the program counter is used. This command assumes that the specified instruction is part of a procedure. Code is disassembled until the end of the procedure. The dot address is set to the address specified.

The procedure name and offsets are given in the first column, followed by the actual addresses. A centered dot character (•) after the address indicates that a breakpoint is set at that instruction. The next two fields contain the opcode and operand; an asterisk character (*) before the opcode indicates the current PC.

The comment field (;) gives the target of a JMP, JSR, or BSR instruction, the trap number of a trap, or the ASCII value of a DC statement. The last field shows the actual hexadecimal words of the instruction; if there are too many words, an ellipsis (...) is shown. Note that this last field is shown only on larger displays, but can be always be seen by sending the output to a file or printer with the LOG command.

Example

```
IR :

 Disassembling from :
  Main
     +0000  00308A8A   LINK       A6,#$0000                                | 4E56 0000
     +0004  00308A8E   MOVEA.L    (A7)+,A6                                 | 2C5F
     +0006  00308A90   JSR        *+$02F8                  ; 00308D88      | 4EBA 02F6
     +000A  00308A94   _Debugger                           ; A9FF          | A9FF
     +000C  00308A96  *JSR        PROCATLEVEL1+0000        ; 00308A6A      | 4EBA FFD2
     +0010  00308A9A   JSR        *+$0312                  ; 00308DAC      | 4EBA 0310
     +0014  00308A9E • JSR        *+$0314                  ; 00308DB2      | 4EBA 0312
     +0018  00308AA2   RTS                                                 | 4E75
     +001A  00308AA4   UNLK       A6                                       | 4E5E
     +001C  00308AA6   RTS                                                 | 4E75
```

See also ID, IL, IP

LOG — Log to a printer or file

Syntax LOG [*pathname* | printer]

Description The LOG command sends MacsBug output to a text file specified by *pathname* or to an ImageWriter printer via the serial port. MacsBug follows the hierarchical file system conventions; if you don't specify a pathname, it assumes the current directory. If the specified file doesn't already exist, it's created as an MPW text file, which can be opened from word processing applications as well as from MPW. If the specified file already exists and is of type Text, LOG appends MacsBug output to what's already there. To turn logging off, simply type `LOG` without parameters.

The LOG command does not work with the LaserWriter® driver, so you can't send MacsBug output directly to a LaserWriter. You can, of course, send the output to a file and then print it on a LaserWriter.

▲ **Warning** MacsBug, by design, uses as little of the system as possible; the LOG command violates this design criterion. Logging may not work, depending on the state of the file system during your debugging session. You should not log to file server volumes. Also, logging enables interrupts briefly while executing its low-level calls. If your program depends on interrupts being completely disabled, you should not use the LOG command. ▲

△ **Important** If you log to a file while MPW Pascal is running, or while an application is running under MultiFinder, be aware that the log file will be closed when you leave MPW or quit the application. △

MC — Macro

Syntax

MC *name*'*expr*' | *expr*

Description

The MC command creates a macro with the given *name* that expands to '*expr*' or to the current value of *expr*. If *expr* is not quoted, it is evaluated and converted to a string before being entered.

Macros are expanded before the command line is executed; thus they can contain anything you can type in a command line. You can use macros to create command name aliases, reference global variables, and name common expressions.

△ **Important** MacsBug expands all macros on the command line before interpreting any commands. You cannot define a macro and then reference it on the same line, because the reference will be undefined at the time the macro is expanded. △

Macros created with the MC command are lost when you restart or shut down your machine. If you have macros you want to keep, you can define them in a resource of type 'mxbm'.

The Debugger Prefs file contains two 'mxbm' resources, with IDs of 100 and 101, that define standard MacsBug macros (including macros for several hundred common global variables). There are two ways to create your own 'mxbm' resources. First, you can use the file Macros.r (included on the MacsBug disk) as a model for building your own resource. Be sure to give your resource a unique ID, and then use the Rez tool to add it to the Debugger Prefs file. Or, you can use ResEdit; Debugger Prefs contains templates for creating and editing 'mxbm' resources.

Two macro names have been predefined by MacsBug for customizing the debugging environment. If you want to execute certain commands to configure MacsBug to your preferred settings (for example, SHOW, SWAP, LOG, SX, HX, and DX), define them as a macro called FirstTime in an 'mxbm' resource. (Remember that multiple commands must be separated by semicolons.)

When a FirstTime macro is present, a break is forced immediately after MacsBug is loaded and the specified commands are executed. If you want the boot process to continue automatically, end the FirstTime macro with the G command.

△ **Important** On a Macintosh Plus, the G command is required. Since the keyboard is initialized *after* MacsBug, you won't be able to type G to continue.△

A second macro, called EveryTime, can be defined in a resource file or on the fly with the MC command. The commands specified by this macro will be executed each time, except the first time, MacsBug is invoked.

MacsBug treats commands defined by macros just like commands that you enter explicitly. If you create an EveryTime macro, be aware that the last command executed by that macro is set as the default command; this command will be repeated if you press Return.

Examples

```
MC Frame 'A6+10'
```

This example gets the current value of register A6 each time the Frame macro is expanded, and adds 10 to it.

```
MC Save CurrentA5
```

This example remembers the current value of this global variable. You could change it and then restore it by typing

```
SL CurrentA5 Save
```

See also MCC, MCD

MCC — Macro Clear

Syntax	MCC [*name*]
Description	The MCC command clears the macro with the given *name*. If no name is specified, all macros are cleared.
See also	MC, MCD

MCD — Macro Display

Syntax MCD [*name*]

Description The MCD command lists those macros that match the given *name*. If no name is specified, all macros are listed, including both predefined macros loaded from resource files and macros defined during the current debugging session. MacsBug provides partial name matching, returning all macros that begin with the specified name. If you enter `MCD Cur`, for instance, all names that start with *Cur* are shown.

Example

```
MCD Cur

 Macro Table
  Name                Expansion
  CurActivate         A64
  CurApName           910
  CurApRefNum         900
  CurDeactive         A68
  CurDirStore         398
  CurJTOffset         934
  CurMap              A5A
  CurPageOption       936
  CurPitch            280
  CurrentA5           904
  CurStackBase        908
```

See also MC, MCC

MR — Magic Return

Syntax

MR [*offset* | *addr*]

Description

If you've stepped into a procedure and want to get out, you can use the MR command. It sets a temporary breakpoint at the first instruction after the call to the current procedure, by replacing the return address on the stack with a MacsBug address. When the procedure returns, MacsBug gets control. It then performs an RTS in trace mode, breaking at the instruction after the call.

If no parameter is specified, the return address is assumed to be on the top of the stack. If specified, the parameter is interpreted relative to either register A7 or A6. If the parameter is less than the contents of A6, MacsBug assumes that it's an offset from register A7. If the parameter is equal to register A6, it's assumed to be a frame pointer for the current procedure. If the parameter is greater than register A6, it's interpreted as an offset for a procedure higher on the stack.

If the specified address is not in the range between A7 and CurStackBase, the error message "This address is not a stack address" is returned. Also, MacsBug checks that the specified address is in fact a valid return address, in other words, that it immediately follows a JSR, BSR, or A-trap instruction. If this is not the case, the error message "The address on the stack is not a return address" is returned.

Examples

If you are at the first instruction in a procedure, simply typing `MR` will break when the procedure is done.

If you are past the LINK A6 instruction, `MR A6` will break when the procedure is done. With nested procedures, `MR A6^` will break when the procedure that called the procedure you are in is done.

See also G, GT

RAD — Toggle Register Name Syntax

Syntax RAD

Description MacsBug lets you specify the address and data registers in two different ways. By default, MacsBug expects the actual Motorola names for these registers. Early versions of MacsBug, however, used names of the form RD0 through RD7 and RA0 through RA7. The RAD command toggles between these two formats, letting you use the syntax you prefer.

◆ *Note:* If you use the default format, you'll need to precede hexadecimal numbers A0 through A7 with the "$" character, to distinguish them from the Motorola address register names.

See also Registers

RB — Reboot

Syntax RB

Description The RB command unmounts the boot volume and reboots the system.

See also EA, ES, RS

Registers

Syntax registerName [= | := *expr*]

Description Entering a register name displays the register's value. Values can be assigned to registers by using either the "=" or the ":=" operator.

By default, MacsBug uses the Motorola names for all registers; a list of these names is given below. (If you're a long-time MacsBug user, you may prefer the syntax used in earlier versions for the address and data registers. The RAD command lets you toggle between the two formats.)

68000 Registers

Dn	Data Register n
An	Address Register n
PC	Program Counter
SR	Status Register
SP	Stack Pointer
SSP	Supervisor Stack Pointer

68020 Registers

ISP	Interrupt Stack Pointer
MSP	Master Stack Pointer
VBR	Vector Base Register
SFC	Source Function Code Register
DFC	Destination Function Code Register
CACR	Cache Control Register
CAAR	Cache Address Register

68030/68851 Registers

CRP	CPU Root Pointer
SRP	Supervisor Root Pointer
TC	Translation Control Register
PSR	PMMU Status Register

68881 Registers

FPn	Floating-Point Data Register n
FPCR	Floating-Point Control Register
FPSR	Floating-Point Status Register
FPIAR	Floating-Point Instruction Address Register

See also RAD

RN — Set Reference Number

Syntax

RN [*expr*]

Description

The RN command lets you restrict symbol references to the file whose reference number is specified by *expr*. The reference number can be found with the HD command. If no expression is specified, the reference number of the current resource file, contained in the global variable CurMap, is used.

The RN command is useful when you're dealing with multiple files with the same symbol names. When you're working with MPW tools, for instance, there may be multiple code segments with the same name. Once you've specified a reference number with the RN command, subsequent symbol references are restricted to the file with a matching reference number.

Specifying 0 for *expr* restores the default situation where all symbols match.

See also

SX

RS — Restart

Syntax	RS
Description	The RS command restarts the system as if the Restart menu item had been chosen from the Finder.
See also	EA, ES, RB

S — Step

Syntax

S [*n* | *expr*]

Description

The S command steps through the next ***n*** instructions or until the specified expression is TRUE. If neither parameter is specified, the S command simply steps through the next instruction. In contrast to the SO command, the S command will actually trace into subroutine calls, or into the ROM when a trap is encountered.

◆ *Note:* If you find you've entered a number or expression that will never be reached or satisfied, you'll need to use the ES command to terminate the stepping.

An S command entered with a specified range or number of instructions (for instance, `s 10`) might encounter a breakpoint while executing. If this happens, the break into MacsBug terminates the S command.

Command-S is provided as a shortcut. Note that any commands sitting in the command line are ignored.

△ **Important** Stepping through certain MMU instructions can cause MacsBug to hang. If you're doing MMU programming, be aware that MacsBug executes many instructions while executing an S command and expects a valid memory map. △

See also

SO

SB — Set Byte

Syntax SB *addr*(*expr* | '*str*'...)

Description The SB command assigns values to bytes, starting at *addr*. Expressions are evaluated to 32-bit values, and the low-order byte is used. Strings of any length (limited only by the length of the command line) can also be specified; the characters are placed in successive bytes. The dot address is set to the address of the first byte set.

In addition to setting the dot address, the SB command sets DM as the default command; pressing Return after having executed an SB command will display the memory just set.

Example

```
SB 0 1 222 33333
{Return}

 Memory set starting at 00000000
  00000000  0122 3300 0000 0000  0000 0000 0000 0000  •"3•••••••••••••
```

See also SL, SW

SC6 — Stack Crawl (A6)

Syntax SC6 [*addr*]

Description The SC6 command displays the stack frame and address of the current procedure and all procedures above it in the calling order.

The SC6 and SC7 commands must have a range of memory to constrain the search for frames or return addresses. They assume that register A7 is even and points to the top of the stack, and that the global variable CurStackBase points to the bottom of the stack. If any of these conditions is not met, the following error message is returned: "Damaged stack: A7 must be even and <= CurStackBase."

The SC6 command also assumes that register A6 or the parameter is the address of a frame on the stack and that it points within the range between register A7 and CurStackBase. If these conditions aren't met, the error message "A6 does not point to a stack frame" is returned.

◆ *Note:* For historical reasons, SC is provided as an alias for the SC6 command.

Example

In this example, `4CEDE4` was the value of A6 at the time ProcAtLevel1 called ProcAtLevel2. `4CEDDC` was the value of A6 at the time ProcAtLevel2 called ProcAtLevel3. The current value of A6 defines the stack frame for ProcAtLevel3.

```
SC6

 Calling chain using A6 links
  A6 Frame   Caller
   <main>    00041FAA MAINPROC+000C
  004CEDE4   00041F82 PROCATLEVEL1+0004
  004CEDDC   00041F66 PROCATLEVEL2+0004
```

See also SC7

SC7 — Stack Crawl (A7)

Syntax SC7

Description The SC7 command displays a possible calling chain with the stack addresses that contain each caller's return address. A return address must be even and a valid RAM or ROM address, and it must point immediately after a JSR, BSR, or A-trap instruction.

The SC7 command will almost certainly include old or invalid values (in other words, addresses not in the current calling chain), since local stack variables can change the stack top without changing the contents. You can use the frame and return addresses to examine the stack yourself; you can also use the addresses in other MacsBug commands.

The SC6 and SC7 commands must have a range of memory to constrain the search for frames or return addresses. They assume that register A7 is even and points to the top of the stack, and that the global variable CurStackBase points to the bottom of the stack. If any of these conditions is not met, the following error message is returned: "Damaged stack: A7 must be even and <= CurStackBase."

The first column shows possible return addresses. The second column shows the addresses of possible A6 frame values.

When debugging routines that don't use the standard A6 frame conventions, a frame address can be used as a parameter to SC6 to tell it where the A6 links start. For instance, typing `SC6 4CEDD4` will show the same calling chain as in the SC6 example.

SC7 shows a superset of the calling chain. SC6 can then be used to show the true calling chain at the point where SC7 finds the first valid frame.

Example

```
SC7

Return addresses on the stack
  Stack Addr  Frame Addr   Caller
   004CEDEC                4080D5CC Chain+014E
   004CEDE8                00041FAA MAINPROC+000C
   004CEDE0    004CEDDC    00041F82 PROCATLEVEL1+0004
   004CEDD8    004CEDD4    00041F66 PROCATLEVEL2+0004
```

See also SC6

SHOW — Show

Syntax SHOW [*addr* | '*addr*'] [L | W | A | LA]

Description By default, MacsBug displays the stack pointer at the top of the status region, as well as the bytes starting at that address. The address is evaluated each time the display is updated. The number of bytes displayed varies with the screen size and the format of the display. The SHOW command lets you specify the display in word, long word, and ASCII format, by passing W, L, or A respectively. You can also specify a combined long/ASCII format by passing LA.

Entering SHOW without parameters cycles between the four display formats so that you don't need to enter the address expression to change the format.

The SHOW command also lets you specify another area of memory for display. If *addr* is quoted, the specified address is evaluated each time the display is updated. If *addr* is not quoted, the address is evaluated once and the resulting address is always shown.

To restore the default display, enter `SHOW 'SP' L`.

Examples

```
SHOW 'A6+8'
```

This example shows the stack above the previous A6 value and return address; for routines using LINK A6, this will be the routine parameters.

```
SHOW curApName A
```

This example will always show the data at the address defined by the macro curApName.

SL — Set Long

Syntax SL *addr*(*expr* | '*str*'...)

Description The SL command assigns values to long words, starting at *addr*. Expressions are evaluated to 32-bit values. Strings of any length (limited only by the length of the command line) can also be specified; the characters are placed in successive bytes. The dot address is set to the address of the first long word set.

In addition to setting the dot address, the SL command sets DM as the default command; pressing Return after having executed the SL command will display the memory just set.

Examples

```
SL 0 1 222 33333
{Return}

 Memory set starting at 00000000
  00000000  0000 0001 0000 0222  0003 3333 0000 0000  •••••••"••33••••

SL 0 12 'Test'
{Return}

 Memory set starting at 00000000
  00000000  0000 0012 5465 7374  0000 0000 0000 0000  ••••Test••••••••
```

See also SB, SW

SM — Set Memory

Syntax SM *addr* (*expr* | '*str*' ...)

Description The SM command assigns values to memory starting at *addr*. The size of each assignment is determined by the value. Specific assignment sizes can be set by using the SB, SW, and SL commands.

In addition to setting the dot address, the SM command sets DM as the default command; pressing Return after having executed the SM command will display the memory just set.

Examples

```
SM 0 1 222 33333
{Return}

 Memory set starting at 00000000
  00000000  0102 2200 0333 3300  0000 0000 0000 0000  ••"••33••••••••••

SM 0 4 'Test'
{Return}

 Memory set starting at 00000000
  00000000  0454 6573 7400 0000  0000 0000 0000 0000  •Test••••••••••••
```

See also SB, SL, SW

SO — Step Over

Syntax SO [*n* | *expr*]

Description The SO command steps through the next *n* instructions or until the specified expression is TRUE. If neither parameter is specified, the SO command simply steps through the next instruction. In contrast to the S command, SO steps over traps, JSRs, and BSRs, treating them as a single instruction.

◆ *Note:* If you find you've entered a number or expression that will never be reached or satisfied, you'll need to use the ES command to terminate the stepping.

When stepping over a toolbox trap with the auto-pop bit set, MacsBug correctly returns to the address on the top of the stack at the time of the trap call (instead of to the address immediately after the trap). If you step over a LoadSeg trap, MacsBug will stop at the first instruction of the loaded segment.

△ **Important** Stepping through certain MMU instructions can cause MacsBug to hang. If you're doing MMU programming, be aware that MacsBug executes many instructions while executing an SO command, and expects a valid memory mapping. △

◆ *Note:* For historical reasons, T (for *Trace*) is provided as an alias for the SO command. In addition, Command-T is provided as a shortcut; note that any commands sitting in the command line are ignored.

See also S

SS — Step Spy

Syntax SS *addr1* [*addr2*]

Description The SS command is a variation on the S command that lets you keep track of a particular area of memory. For the range between *addr1* and *addr2*, the SS command calculates a checksum before executing the next instruction. If the checksum value changes, MacsBug is invoked. If *addr2* is omitted, SS waits for the long word at *addr1* to change.

The SS command is terminated on the next entry into MacsBug.

The SS command is optimized for speed with a long word; with longer checksum ranges, it can be slow. Programmers needing to watch large ranges may want to use a hardware emulator.

You can also use the SS command as a way of slowing down certain routines—those that draw to the screen, for instance—so you can actually watch how they work.

Example

This example specifies a range that will not change and can be used to watch drawing to the screen.

```
SS ROMBase^(RomBase^+40)
```

See also CS

SW — Set Word

Syntax SW *addr*(*expr* | '*str*'...)

Description The SW command assigns values to words starting at *addr*. Expressions are evaluated to 32-bit values, and the low-order word is used. Strings of any length (limited only by the length of the command line) can also be specified; the characters are placed in successive bytes. The dot address is set to the address of the first word set.

In addition to setting the dot address, the SW command sets DM as the default command; pressing Return after having executed the SW command will display the memory just set.

Examples

```
SW 0 1 222 33333
{Return}

 Memory set starting at 00000000
  00000000  0001 0222 3333 0000  0000 0000 0000 0000  •••"33••••••••••

SW 0 12 'Test'
{Return}

 Memory set starting at 00000000
  00000000  0012 5465 7374 0000  0000 0000 0000 0000  ••Test••••••••••
```

See also SB, SL

SWAP — Swap Frequency

Syntax SWAP

Description The SWAP command controls the frequency of display swapping between MacsBug and the application, depending on whether the system is configured for a single screen or for multiple screens.

For single screens, the SWAP command toggles between drawing step and A-trap trace information to the MacsBug display without swapping the screen, and drawing the information and swapping each time.

For multiple screens, the SWAP command toggles between having the MacsBug screen always visible, and having the MacsBug screen visible only at break.

With multiple screens, MacsBug uses the "Welcome to Macintosh" screen by default. You'll probably want your application on the larger screen and MacsBug on the smaller screen. To select a different screen for the MacsBug display, press the Option key while clicking on the Monitor icon from the Control Panel, drag the Macintosh icon to the desired screen, and reboot.

SX — Symbol Exchange

Syntax SX [ON | OFF]

Description The SX command toggles between allowing and not allowing symbol names in place of addresses. By default, symbol names can be used anywhere an address is used as a command line parameter. MacsBug translates this name into an address by searching the current heap for a matching procedure name. MacsBug also displays disassembled code as offsets relative to a procedure. Since this search process can be slow, MacsBug provides a way to disable it.

See also IL, RN

TD — Total Display

Syntax TD

Description The TD command displays all CPU registers in the command region. Since most 68000 registers are constantly displayed in the status region, this command is useful for remembering the register values between commands.

To display the 68030 MMU registers, use the TM command.

Examples

```
TD (on a Macintosh Plus)

 68000 Registers
  D0 = 00000000      A0 = E0025470      USP  = FFFFFFFF
  D1 = 00000006      A1 = 000CC7B2      SSP  = 000CC6CA
  D2 = FFFF0040      A2 = 000CC7B2
  D3 = 00000000      A3 = 000CC7B2
  D4 = 00000000      A4 = 000213B2
  D5 = 00000000      A5 = 000CD594
  D6 = 00000000      A6 = 000CC6E4      PC   = E002547E
  D7 = 00000000      A7 = 000CC6CA      SR   = Smxnzvc      Int = 0

TD (on a Macintosh II)

 68020 Registers
  D0 = 00000000      A0 = E0017EA8      USP  = D72B5FFA
  D1 = 00000006      A1 = 00487290      MSP  = 234B30CD
  D2 = FFFF280C      A2 = 00487290      ISP  = 004871C6
  D3 = 00000000      A3 = 00487290      VBR  = 00000000
  D4 = 0048FFFF      A4 = 004872D2      CACR = 00000001      SFC = 7
  D5 = 00000000      A5 = 004D013C      CAAR = 08281E55      DFC = 7
  D6 = 004D013C      A6 = 004871D6      PC   = E0017EB6
  D7 = 00000000      A7 = 004871C6      SR   = SmXnzvc      Int = 0
```

See also TF, TM

TF — Total Floating-Point

Syntax TF

Description The TF command displays all 68881 registers. (These registers are not shown in the status region.)

Example

```
TF (on a machine with a 68881)

 68881 Registers
  FP0  = 7FFF FFFFFFFF FFFFFFFF        NAN(255)
  FP1  = 7FFF FFFFFFFF FFFFFFFF        NAN(255)
  FP2  = 7FFF FFFFFFFF FFFFFFFF        NAN(255)
  FP3  = 7FFF FFFFFFFF FFFFFFFF        NAN(255)
  FP4  = 7FFF FFFFFFFF FFFFFFFF        NAN(255)
  FP5  = 7FFF FFFFFFFF FFFFFFFF        NAN(255)
  FP6  = 7FFF FFFFFFFF FFFFFFFF        NAN(255)
  FP7  = 7FFF FFFFFFFF FFFFFFFF        NAN(255)
         EE MC           CC QT ES AE
  FPCR = 00 00    FPSR = 00 00 00 00    FPIAR = 00000000
```

See also TD, TM

TM — Total MMU

Syntax TM

Description The TM command displays the MMU registers common to the 68851 and 68030. (These registers are not shown in the status region.)

Example

```
TM (on a machine with a 68851)

 MMU Registers
  CRP = 7FFF020240800050        TC   = 80F84500
  SRP = 7F55D27300000100        PSR  = 2216
```

See also TD, TF

TMP — Templates

Syntax TMP [*name*]

Description The TMP command lists every template whose name matches the specified name. If no name is specified, all loaded templates are displayed by name. MacsBug provides partial name matching, returning all templates that begin with the specified name. If you enter `TMP My`, for instance, all names that start with *My* are shown.

The Debugger Prefs file contains an 'mxwt' resource with an ID of 100; this resource defines standard MacsBug templates. There are two ways to create your own 'mxwt' resources. First, you can use the file Templates.r (included on the MacsBug disk) as a model for building your own resource. Be sure to give your resource a unique ID, and then use the Rez tool to add it to the Debugger Prefs file. Or, you can use ResEdit; Debugger Prefs contains templates for creating and editing 'mxwt' resources.

Templates are composed of fields. Each field consists of a name, a type, and a count. The basic types are as follows:

Byte	Display in hexadecimal.
Word	Display in hexadecimal.
Long	Display in hexadecimal.
SignedByte	Display in decimal.
SignedWord	Display in decimal.
SignedLong	Display in decimal.
UnsignedByte	Display in decimal.
UnsignedWord	Display in decimal.
UnsignedLong	Display in decimal.
Boolean	Display byte as TRUE (nonzero) or FALSE (0).
pString	Display a Pascal string.
cString	Display a C string (zero-terminated).

For all of the basic types except pString, the count indicates the number of items of that type to display. For instance, a type of Word with a count of 4 can be used to display a Rectangle on one line. With pStrings, the count indicates the maximum string size and is used to compute the next field address. If the string is only as long as the actual number of characters, specify 0 for count, and MacsBug will use the length byte to determine the end of the string.

The basic types listed above can also be used individually with the DM command. Several additional field types are used only in templates:

Text	Displays a text string for count bytes. (Resource types, for instance, can be shown with the Text type and a count of 4.)
Skip	Skips over the next count bytes without displaying.
Align	Aligns to a word boundary (used after C or Pascal strings).
Handle	Dereferences and displays in hex. This type is used to show the address of a data structure, rather than its contents.
^Type	Dereferences a pointer and displays using the specified basic type or template. The display is indented 2 spaces.
^^Type	Dereferences a handle and displays using the specified basic type or template. The display is indented 2 spaces.

If a template named Temp contains a field type of `^Temp` or `^^Temp`, MacsBug assumes the field is a link to another data structure of the same type. For instance, the WindowRecord template (provided in Templates.r) uses a field type of `^WindowRecord` to dereference the pointer contained in the nextWindow field of the windowRecord. Pressing Return displays the next window in the window list.

Linked lists are zero-terminated. If a template contains more than one field specifying a link, MacsBug uses the last field found.

WH — Where

Syntax WH [*addr* | *trap*]

Description The WH command returns information about the location of a given trap, symbol, or address. If no parameter is specified, the program counter is used. Given an address that's in ROM, WH looks for the trap nearest to and before that address, and returns the trap name as well as an offset from the start of the trap. If the address is in the system heap or application heap, WH returns the symbol (name and offset).

MacsBug will also attempt to map a given address to low memory global names. It does this by trying to convert macro values into numbers. If the value is a legal number and matches the given address, the macro name is returned.

If a trap name or number is specified, the WH command returns the trap name, the trap number, and the address of the trap. If a symbol name is specified, WH returns the address.

The WH command sets the dot address; thus `WH name` followed by `IL .` will disassemble the code at *name*.

In the example below, typing `WH` gets information about the PC. It is in the procedure MainProc at offset 000C. The heap block where this procedure was found is also shown. (See the HD command for details.)

Example

```
WH

 Address 000E7D36 is in the Application heap at MainProc+000C
 It is in this heap block:
     Start     Length    Tag  Mstr Ptr  Lock Purge  Type   ID   File
  •000E7CC8  000003D0    R   000E7CAC   L          CODE  0001  0236
```

Appendix A **Command Summary**

Flow control

G — Go

GT — Go Till

S — Step

SO — Step Over

SS — Step Spy

MR — Magic Return

Breakpoints

BR — Breakpoint

BRC — Breakpoint Clear

BRD — Breakpoint Display

BRM — Multiple Breakpoints

A-Traps

ATB — A-Trap Break

ATT — A-Trap Trace

ATHC — A-Trap Heap Check

ATSS — A-Trap Step Spy

ATC — A-Trap Clear

ATD — A-Trap Display

ATR — A-Trap Record

ATP — A-Trap Playback

DSC — Extended Discipline ■

Disassembly commands

IL — Disassemble From Address

IP — Disassemble Around Address

ID — Disassemble One Line

IR — Disassemble Until End of Procedure

DH — Disassemble Hexadecimal

Heap commands

HX — Heap Exchange

HZ — Heap Zone

HD — Heap Display

HT — Heap Totals

HC — Heap Check

HS — Heap Scramble

Symbol commands

RN — Resource Number

SX — Symbol Exchange

Stack commands

SC6 — Stack Crawl (A6)

SC7 — Stack Crawl (A7)

Memory commands

DM — Display Memory

TMP — Display all loaded templates

DP — Display Page

DB — Display Byte

DW — Display Word

DL — Display Long

SM — Set Memory

SB — Set Byte

SW — Set Word

SL — Set Long

Register commands

TD — Total Display

TF — Total Floating-Point

TM — Total MMU

Macro commands

MC — Macro Create

MCC — Macro Clear

MCD — Macro Display

Miscellaneous commands

RB — Reboot

RS — Restart

ES — Exit to Shell

EA — Exit to Application

WH — Where

F — Find

CS — Checksum

LOG — LOG (output to file or printer)

HOW — Display Break Message

SHOW — Show (memory in the sidebar)

DV — Display Version

DX — Debugger Exchange

HELP — Display list of MacsBug commands

SWAP — Swap (screen display)

RAD — Toggle Register Name Syntax

Appendix B **Error Messages**

This appendix lists most of the error messages MacsBug can return.

"Unable to access that address"

"Addresses must be even"

Any command that takes an address parameter can get one of these errors. The first is a 68000 bus error exception, and the second is an address error exception.

"Value expected"

Some commands will supply default parameters when no parameter is specified. This error can be returned by commands that require certain parameters.

"Unrecognized symbol"

Any command that takes a symbol as parameter can receive this error if a valid symbol name could not be found in the heap and the name is not a valid trap name.

"Divide by zero error"

This error is returned when an expression attempts to divide a number by zero.

"Count must be greater than zero"

Any command that takes a count (BR, ATB) requires it to be greater than 0.

"Entry will not fit in the table"

MacsBug stores information about breakpoints, step commands, and A-trap commands in a single table. Note that it's possible to receive this message while entering one type of action for the first time (a breakpoint for instance), since other types of actions may have already filled this table. ■

"Damaged stack: A7 must be even and <= CurStackBase"

The stack commands (SC6, SC7) must have a memory range to constrain the search for lframes or return addresses. They assume that register A7 is even and points to the top of the stack, and that the global variable CurStackBase points to the bottom of the stack.

"A6 does not point to a stack frame"

The SC6 command assumes that register A6, or the parameter if specified, is the address of the first frame on the stack. It must point within the range specified by register A7 and CurStackBase.

"This address is not a stack address"

The MR command can optionally take a parameter specifying where on the stack the return address for the current procedure is located. This address must be even and within the range specified by register A7 and CurStackBase.

"The address on the stack is not a return address"

The MR command must know where the return address for the current procedure is located on the stack, since it replaces this address with an internal MacsBug address. MacsBug checks that the address it replaces is in fact a return address. A return address is defined as an address immediately following a JSR, BSR, or A-trap instruction. (All forms of JSR and BSR are recognized.)

"Floating-point not allowed in expressions"

"64-bit registers not allowed in expressions"

All expressions are evaluated as unsigned 32-bit values; floating-point registers and some MMU registers cannot be evaluated in this context.

"No blocks of this type found"

The HD command was instructed to display only blocks of a specific kind and none were found.

"Address range must be entered before comparisons"

The CS command remembers a range of memory to checksum; subsequent CS commands compute the checksum and compare it against the previous value. If no address range has been previously specified, entering CS without parameters will return this message.

"Low address must be less than or equal to high address"

The CS command requires an ordered address range.

"MMU not installed"

The TM command functions only if the system has a 68851 or 68030 installed. This error also occurs if you try to display or set an individual MMU register.

"68881 not installed"

The TF command functions only if the system has a 68881 installed. This error also occurs if you try to display or set an individual floating-point register.

"Macro expansion exceeds maximum command line length"

Macros are expanded in the command line buffer. This is a fixed-length buffer determined by the width of the command line on the current display.

"The template contains an unrecognized basic type"

The field of the template currently being displayed is not a valid basic type; see the description of the TMP command for a list of all possible types.

"Templates cannot expand more than 8 levels"

Template definitions can themselves contain template definitions, and so on. Expansion is limited to eight levels. Since it's unlikely that a structure would contain this many levels, this message may indicate a template definition that contains a recursive path.

"PC is not inside a procedure"

The ":" character can be used to represent the address of the start of the procedure displayed in the program counter window. If you enter ":" and no symbol information can be found for the program counter, this error message will be displayed.

"Zone pointer is bad"

The zone pointer for the current heap (SysZone, ApplZone, or user address) must be even and in RAM. In addition, the bkLim field of the header must be even and in RAM, and must point after the header.

"Free master pointer list is bad"

Free master pointers in the heap are chained together, starting with the hFstFree field in the zone header and terminated by a NIL pointer.

"BlkLim does not agree with heap length"

Walking through the heap block by block must terminate at the start of the trailer block, as defined by the blkLim field of the zone header.

"Block length is bad"

The block header address plus the block length must be less than or equal to the trailer block address. Also, the trailer block must be a fixed length.

"Nonrelocatable block: Pointer to zone is bad"

Block headers of nonrelocatable blocks must contain a pointer to the zone header.

"Relative handle is bad"

The relative handle in the header of a relocatable block must point to a master pointer.

"Master pointer does not point at a block"

The master pointer for a relocatable block must point at a block in the heap.

"Free bytes in heap do not match zone header"

The zcbFree field in the zone header must match the total size of all the free blocks in the heap.

"Syntax error"

This is a "catch-all" error message; it's used in cases where the error is obvious given the context of the command. Possibilities include:

- An expression contains a value, an operator, but no second value.
- A nested expression does not have matching parentheses.
- An address qualifier other than .B, .W, or .L has been given.
- An illegal character is in the command line.
- The ATSS command does not include an address range.
- The format parameter for the SHOW command is other than L, W, A, or LA.
- The F command does not have the correct number of parameters.
- The value being assigned to a floating-point register is illegal.
- A toggle command has been passed a parameter other than ON and OFF.
- The HD command qualifier is not valid.

Appendix C **MacsBug Internals**

MACSBUG USES AS LITTLE OF THE SYSTEM AS POSSIBLE. In addition, when MacsBug gets control, it effectively halts the processor by disabling interrupts. This appendix gives details on the MacsBug implementation.

Beginning with the 128K ROM, support for debuggers is provided. When a system error or 68000 exception occurs, the ROM code examines the global variable MacJmp to see if a debugger is installed. The high-order byte of MacJmp is used to contain the following information.

Bit	**Meaning**
7	Set if debugger is running.
6	Set if debugger can handle system errors.
5	Set if debugger is installed.
4	Set if debugger can support the Extended Discipline utility.

If a debugger is installed, the register set is saved in the global variable SEVars, and a call is made to the address in the low-order 3 bytes of MacJmp. When the debugger returns, the register set is restored and execution returns at the address in the program counter.

While active, MacsBug installs a bus error handler to catch any illegal memory references. MacsBug does not install an address error handler since it can check whether addresses are even before accessing them.

MacsBug itself forces two kinds of exceptions. The first is used in setting breakpoints. MacsBug replaces the first word in an instruction with a TRAP #F instruction; when the program reaches this point, an exception is generated. The second is used in tracing instruction execution while single-stepping. MacsBug forces an exception by setting the Trace bit of the status register before executing an instruction. ■

MacsBug installs its own trace exception handler whenever

- At least one ROM breakpoint is set.
- A breakpoint was set at the PC when execution resumed. The instruction must be executed before the breakpoint can be reinstalled.
- A step command is in progress.
- A step spy command is in progress.

The SO command steps over JSR and BSR instructions by executing the call with the Trace bit set, replacing the return address with an address inside MacsBug, and then proceeding normally. Stepping over a trap call is done by copying the trap instruction into MacsBug and proceeding from that point.

MacsBug installs its own A-trap exception handler whenever

- An A-trap command is active.
- The Extended Discipline utility is enabled.
- Heap scrambling is enabled.
- It steps into a trap call.

The Debug and DebugStr traps do not preserve the status register (SR). These traps are directed to MacsBug by the dispatcher, which tosses the contents of the SR immediately upon entry.

Since interrupts are turned off, MacsBug gets keys by polling for a keyboard interrupt and then calling the interrupt routine at Lvl1DT+8. MacsBug fields the event by temporarily installing its own PostEvent handler.

MacsBug assumes the display on a Macintosh Plus or Macintosh SE is at address $3FA700, accommodating external monitors that change ScrnBase. MacsBug always appears on the internal display.

On a Macintosh II, MacsBug uses the first item in the gDevList as its display. The device must support 1-bit mode, and the display is limited to 640 by 480 to conserve memory.

While swapping the user and MacsBug displays on multi-bit displays, MacsBug calls SetMode and SetEntries (using the Control trap) to set a bit depth of 1, and a black-and-white color table.

Appendix D Debugger and DebugStr

THIS APPENDIX SHOWS HOW TO DECLARE AND USE THE DEBUGGER and DebugStr macros on a per language basis. ■

Assembly language

Declaration

```
_Debugger   OPWORD   $A9FF          ; predefined in the file ToolTraps.a
_DebugStr   OPWORD   $ABFF          ; not predefined - define yourself
```

Example calls

```
_Debugger                           ; enters MacsBug and displays user break message

STRING PASCAL                       ; Asm directive to make sure to push a
                                    ;  Pascal string
PEA #'Entered main loop'            ; push address of string on stack
_DebugStr                           ; enters MacsBug and displays message
```

Pascal

Declaration

```
{Defined in OSIntf.p (MPW version 2.0) or Types.p (MPW 3.0)}
PROCEDURE Debugger; INLINE $A9FF;
PROCEDURE DebugStr(str: str255); INLINE $ABFF;
```

Example calls

```
Debugger;                           {enters MacsBug and displays user break message}

DebugStr('Entered main loop');      {Enters MacsBug and displays message}
```

MPW C

Declaration

```
/*Defined in Strings.h (MPW version 2.0) or Types.h (MPW 3.0)*/
#include <strings.h>                /* Required for c2pstr() */
pascal void Debugger() extern 0xA9FF;
pascal void DebugStr(aString) char *aString; extern 0xABFF;
```

Example calls

```
Debugger();                         /*enters MacsBug and displays user break message*/

DebugStr("\pEntered main loop");
                                    /*enters MacsBug and displays message*/
```

Appendix E **External Commands**

EXTERNAL COMMANDS ARE EXECUTABLE CODE RESOURCES of type 'dcmd' (similar to 'XCMD' resources in HyperCard®) that augment the built-in MacsBug command set. As in HyperCard, 'dcmd' resources are termed by their type: "dee-commands" (written *dcmds*).

The RAMDump™ utility is an example of a program suited for a dcmd. This tool dumps the contents of RAM to several floppy disks for later examination. It requires only a minimal user interface and is typically run from MacsBug to save and examine the state of the machine.

Dcmds are added to the MacsBug command list and can be used just like built-in commands. In fact, dcmds can override built-in commands if you wish. It's recommended, however, that you don't override flow-contol commands like G, S, ATB, and so on.

Sample dcmds, written in both C and Pascal, are provided in the dcmds folder on the MacsBug disk. Source code, interfaces, and a "glue" file to be linked with the dcmd are also supplied.

A 'dcmd' resource begins with a 4-byte segment header, followed immediately by the program code. Since dcmds are limited to a single segment, the segment header is used to specify a dcmd version number and the amount of space MacsBug needs to allocate for the dcmd's global variables (in other words, the size of its "A5 world"). ■

All calls to a dcmd are made through the entry point defined as the fifth byte of the resource. MacsBug calls a dcmd as a Pascal procedure taking a single parameter—a pointer to a parameter block:

```
TYPE  dcmdBlockPtr = ^dcmdBlock;
      dcmdBlock    = RECORD
                       registerFile:  RegFilePtr;
                       request:       INTEGER;
                       aborted:       BOOLEAN;
                     END;
```

The `RegisterFile` field of `dcmdBlock` is a pointer to an array containing the contents of the registers:

```
TYPE RegFilePtr   = ^RegFile;
     RegFile      = ARRAY [0..17] OF LONGINT;
```

`RegFile` contains the contents of registers D0 through D7, A0 through A7, PC, and SR. The SR is only 16 bits and is stored in the high-order word of the last long word in the array.

`Request` contains a request number that MacsBug sends to the dcmd; it can contain one of the following values:

```
CONST dcmdInit = 0;
      dcmdDoIt = 1;
      dcmdHelp = 2;
```

The first call MacsBug makes to a dcmd is an initialize request (`dcmdInit`); this request is made only once. After the dcmd is initialized, MacsBug can call it to request a brief summary about itself (`dcmdHelp`) or to perform its normal action (`dcmdDoIt`).

The dcmd can change registers by changing the register file values in the `RegFile` array. These values get restored when MacsBug resumes program execution.

The `aborted` field is used to inform the dcmd when the user has terminated the command; it's set to TRUE when the user presses a key (other than the Return key or the Space bar) while scrolling.

A dcmd can make calls back to MacsBug to request actions, like displaying a message or getting a parameter. All calls to MacsBug are routed through a single entry point with the request number added to the stack immediately before the return address. The address of the entry point is stored at (A5)+4 for each dcmd. The memory above A5—normally used for the jump table entries—can be used as an easy way to get the call-back address.

The data structures, constants, and call-back routines are all defined and explained in the provided sample commands.

Dcmds use the MacsBug stack; MacsBug supplies 1K of stack space. Since Pascal calling conventions are used, the dcmd is responsible for popping the parameters off the stack. In addition, the dcmd must preserve registers D3 through D7 and A2 through A6.

Dcmds can call whatever traps they wish. Be aware, however, that the dcmd can be invoked when the system has crashed; it's obviously safest not to use any traps at this point. It's up to the programmer to decide how much of the system to use. Documentation for dcmds would do well to note the trap calls that are made.

The dcmd folder also includes an MPW tool, BuildDcmd. This tool translates an application into a dcmd, and copies it into the Debugger Prefs file. Since the dcmds are in a separate file, there's no complicated installation process to be performed when MacsBug is updated. And since they're not tied to MacsBug, dcmds can potentially be shared with other debuggers.

The dcmds folder also contains an application, Test dcmd, that simulates the MacsBug environment for testing your dcmds. This application allows you to use MacsBug while debugging a dcmd.

Finally, this folder contains actual dcmds that perform useful functions. Information about each of these dcmds is available in MacsBug by typing `HELP` *name*. You can get a list of all installed dcmds by typing `HELP dcmd`.

Appendix F **Did You Know...?**

This appendix contains tips, shortcuts, and interesting facts about MacsBug. Did you know that...

- Holding down the Control key forces a break into MacsBug immediately after it's loaded. This feature works only on Macintosh computers equipped with the Apple Desktop Bus™ (ADB) interface; the Control key was chosen because it's found only on ADB machines. On machines without ADB, the keyboard is loaded after MacsBug, so it makes no sense to break into MacsBug.
- The DebugStr routine with an argument of `';HC;G'` is a useful way to determine where in your program the heap may become corrupted. The HC command performs a heap check; if the heap is corrupted, MacsBug stops and reports the error. If the heap is in order, the G command is executed and program execution resumes. Sprinkling such calls to DebugStr throughout your program lets you hone in on memory culprits.
- A related technique is to use the ATHC command, which checks the heap prior to each trap call. Using this technique means that you don't need to modify your program, but it does have the disadvantage that you can't choose the frequency and location of the checks.
- In the same way that passing `';HC;G'` with DebugStr checks the heap, passing `';CS;G'` checksums a block of memory. If the block has changed, MacsBug takes over; otherwise program execution continues. Remember that the range must be set up with an initial CS command before subsequent CS commands can compare the checksum.
- You can create a custom A-trap trace by executing the ATB command with an associated action. For instance, you can specify the commands `';TD;G'` for execution upon break. Whereas the ATT command shows only select registers, this action displays all registers. You could further customize the trace by displaying memory based on the content of particular registers.
- You can display the result of a function every time it's called by entering the commands `BR FuncName ';MR;DW SP;G'`. Whenever the breakpoint is reached, MacsBug executes the MR (Magic Return) command and displays the top word on the stack (the function result). (Functions that return long words would use `'DL SP'` instead.) Functions that return pointers could dereference the pointer and display the structure (possibly using a template).

- Macros are a quick way to save values. For instance, you could enter `MC save PC` to save the contents of the program counter, and `PC = save` to restore the contents. (Note that this technique does not work with floating-point registers.)

Appendix G **Procedure Definition**

WHENEVER POSSIBLE, MACSBUG ACCEPTS AND RETURNS ADDRESS AS PROCEDURE NAMES and offsets. Names are found by scanning relocatable heap blocks for valid procedure definitions. A procedure definition in the simplest case consists of a return instruction followed by the procedure's name.

A procedure is defined as follows:

[LINK A6]

Procedure code

RTS or JMP(A0) or RTD

procedure name

procedure constants

The LINK A6 instruction is optional; if it is missing, the start of the procedure is assumed to be immediately after the preceding procedure, or at the start of the heap block.

The procedure name can be a fixed length of 8 or 16 bytes, or of variable length. Valid characters for procedure names are a–z, A–Z, 0–9, underscore (_), percent (%), period (.), and space. The space character is allowed only to pad fixed-length names to the maximum length.

With fixed-length format, the first byte is in the range $20 through $7F. The high-order bit may or may not be set. The high-order bit of the second byte is set for 16-character names, clear for 8-character names. Fixed-length 16-character names are used in object Pascal to show class.method names instead of procedure names. The method name is contained in the first 8 bytes and the class name is in the second 8 bytes. MacsBug swaps the order and inserts the period before displaying the name. ■

With variable-length format, the first byte is in the range $80 to $9F. Stripping the high-order bit produces a length in the range $00 through $1F. If the length is 0, the next byte contains the actual length, in the range $01 through $FF. Data after the name starts on a word boundary. Compilers can place a procedure's constant data immediately after the procedure in memory. The first word after the name specifies how many bytes of constant data are present. If there are no constants, a length of 0 must be given.

Examples of valid assembly-language procedure definitions are given below.

```
; Variable-length name with no constant data.

Proc1       PROC
            LINK  A6, #0
            UNLK  A6
            RTS
            DC.B  $8C, 'VariableName'
            DC.W  $0000
            ENDP

; Fixed 8-character name.

Proc2       PROC
            LINK  A6, #0
            UNLK  A6
            RTS
            DC.B  $80 + 'F', 'ixed   '
            ENDP

; Fixed 16-character name.

Proc3       PROC
            LINK  A6, #0
            UNLK  A6
            RTS
            DC.B  $80 + 'M',  $80 + 'e', 'thod  Class   '
            ENDP
```

Index