ICOM SIMULATIONS, INC.

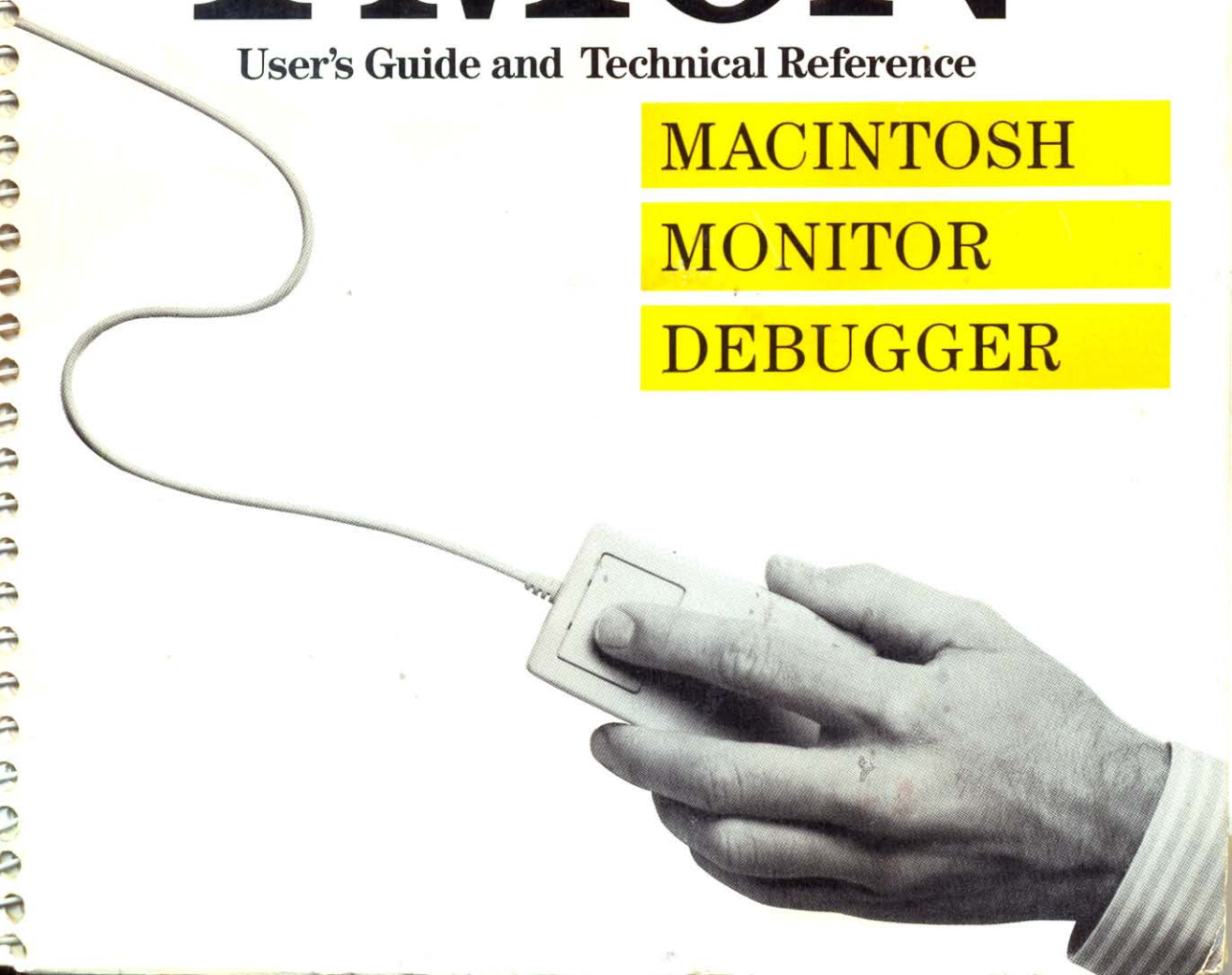
TMON™

User's Guide and Technical Reference

MACINTOSH

MONITOR

DEBUGGER



Return Your Registration Card

ICOM Simulations, Inc. is committed to providing quality products and the highest level of services for our customers. To take full advantage of ICOM's services, you must be a REGISTERED user of TMON. Please take a few moments now to complete the registration card located in this package and forward it to ICOM as soon as possible. As a registered user, you will be entitled to the following benefits:

Customer Service and Product Support. As a part of the service provided to our registered users, we provide a limited warranty on software, and support and assistance by telephone. If you experience any difficulty in using TMON, please refer to your *User's Guide* and *Technical Reference*. If you still need assistance, call our Technical Support Department between 9:00 AM and 5:00 PM (Central Time) Monday through Friday. Our phone number is 312/520-4440. Please have your registration number available, as well as information relevant to your question.

ICOM Upgrade Information. ICOM is constantly looking for ways to enhance its products by incorporating new capabilities and features. These new upgrades will be available to our registered users. As a registered user, you will be regularly updated by mail on new upgrades.

New Product Information. As a registered user, you will be provided with early information on new products and special offers.

Site Licenses

Information about site licenses and volume purchases can be obtained by contacting the Marketing department of ICOM at the number given above.

Limited Warranty on Media and Manuals

If you discover physical defects in the media on which this software is distributed, or in the documentation distributed with the software, ICOM Simulations, Inc. will replace the media or documentation at no charge to you, provided you return the item(s) to be replaced with proof of purchase to ICOM or an authorized ICOM dealer during the 90-day period after you purchased the software.

ALL IMPLIED WARRANTIES ON THE MEDIA AND DOCUMENTATION, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though ICOM has tested the software and reviewed the documentation, ICOM makes no warranty or representation, either express or implied, with respect to this software or documentation, its quality, performance, merchantability, or fitness for a particular purpose. As a result, this software and documentation is licensed "as is," and you, the purchaser are assuming the entire risk as to its quality and performance.

In no event will ICOM be liable for direct, indirect, special, incidental, or consequential damages resulting from any defect in the software or its manuals or any additional documentation, even if advised of the possibility of such damages. In particular, ICOM shall have no liability for any programs or data stored in or used with ICOM products, including the costs of recovering such programs or data.

The warranty and remedies set forth above are exclusive and in lieu of all others, oral or written, express or implied. No ICOM dealer, agent or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

TMON

Version 2.8

User's Guide and Technical Reference

ICOM SIMULATIONS, INC.

648 S. Wheeling Road Wheeling, IL 60090 312/520-4440

Credits

Author
Waldemar Horwat

User Area
Darin Adler
Waldemar Horwat

User's Guide
Paul Snively

Technical Reference
Waldemar Horwat

Contents

Summary of Features

TMON is an object-level symbolic monitor/debugger for the Macintosh personal computer. It will work on all Macintoshes except the Macintosh XL and the Macintosh 128K. Among the features included are:

A *fast* implementation of windows on the screen that has these advantages:

- Information is not lost when it scrolls off the screen.
- Registers, breakpoints, program code, subroutines, data, stack, heaps, and resource files can all be examined at the same time.
- Multiple sections of code can be viewed.
- Windows can be scrolled up or down.
- Disassembly and dump windows can be anchored to registers.
- Instruction and effective addresses in disassembly windows are identified using labels.

Windows update *continuously*. When one window is changed, other windows instantly reflect the change.

An interactive 68000 assembler/disassembler.

- Includes reverse scrolling of disassembly windows.
- A000 traps are displayed by their names, not numbers.
- Labels may be used both by the disassembler and in assembling.

Label and symbol capabilities.

- Labels may be used in any expression.
- Labels may be recognized automatically from routine names in code.
- Labels may be loaded from .MAP files.
- Labels may be entered directly from TMON as either absolute or resource-relative.
- Names of the A000 traps are used as labels during examination of ROM routines.

Predefined labels for low-memory globals.

An interactive hexadecimal and ASCII memory dump and change.

File windows which identify all resources in all open resource files.

- Resources not currently in memory are also displayed.
- Resource flags are shown in an easy to read format.
- Resource types, IDs, names, references, and handles are shown when appropriate.
- Information displayed is checked for consistency.

Heap windows displaying the contents of the application or system heaps.

- The location, size, and type of all heap objects are displayed.
- The addresses of handles and flags are displayed for relocatable objects.
- The resource ID, type, and file are displayed for objects which are resources.
- Windows, controls, window regions, scraps, and various other heap objects are identified. A user routine can be made to identify other heap objects when appropriate.
- Information displayed is checked for consistency.

Register windows which display and allow changing of all 68000 registers.

- The flags are displayed in an easy to read format.

Saving, loading, and exchanging registers with an alternate register set.

Converting numbers and expressions between hexadecimal, decimal, binary, ASCII, A000 trap names, and labels.

Use of expressions involving hexadecimal, decimal, and binary numbers, ASCII values, addition, subtraction, multiplication, division, boolean operations, indirection, parentheses, A000 trap names, and labels.

Up to seven breakpoints.

Single-step execution of programs.

- All stepping and tracing features work in ROM.
- Tracing into A000 traps is possible.
- A convenient function for skipping subroutines during tracing is included.

Searching a block of memory for a 1, 2, 3, or 4 byte value.
Word-aligned, as well as byte-aligned searches.

Block move, compare, fill, and checksum.

Interception of almost all exceptions and system errors.

Interception of any A000 traps upon request.

Interception of program at a specific point when interrupt is pressed.

Quiet recording of all or specific A000 traps which allows the course of execution of a program to be quickly traced. This function may also be used for performance analysis as it records the times of the A000 traps.

A heap check function, which may be automatically run on A000 traps.

A highly optimized heap scramble function, which may be automatically run on A000 traps as well. A purge option is available. The heap scramble clears the unused blocks, providing additional debugging security.

A000 trap discipline, which checks the parameters of A000 traps, catching errors before they cause damage.

Symbolic displays of window records, control records, TextEdit records, and file parameter blocks.

A variable-length user area designed to allow customization of the Monitor.

Printing to a printer, external computer, or terminal.

- Any window on the screen can be printed.
- Disassemblies and dumps of arbitrarily large blocks of memory can be printed.
- Heap and resource file dumps can be printed.
- XOn/XOff and hardware handshaking are supported.
- Printing can be done from either port.

Mouse unfreeze and vector refresh options, which may be very helpful after a program crashes.

The ability to easily disable the more system-dependent Monitor functions like labels and heap identification in case these functions fail because the system is in an inconsistent state.

A Monitor self-test run continuously to provide extra security.

Choice of either system heap or high memory to load the Monitor allowing the Monitor to work with virtually all programs.

The ability to quietly load the Monitor upon starting the Macintosh without any interaction.

The ability to load the Monitor as an INIT. This allows debugging of other INITs and lets another application be the startup application.

The capability of automatically patching the Monitor code via a user area routine.

TMON 2.8 does not support the extra features of the 68020, 68030, 68881, and 68851 processors and coprocessors that are not also found on the 68000 (i.e. it will not disassemble the extra instructions and will not display the extra registers). TMON 2.8 does, however, work with any of these processors.

Congratulations on your purchase of TMON Version 2.8!

In the two and one half years since TMON's initial release much has changed in the Macintosh development world. Apple introduced new ROMs which were 128K in size, and in so doing also introduced several new wrinkles to developers: a new file system, new bugs, more power, and more flexibility. TMON evolved only marginally to accommodate the 128K ROMs; for some time the only way that TMON could be said to be supportive of them was that it was capable of running the Extended User Area (EUA) written by Darin Adler, which patched TMON to work with the 128K ROMs. Fortunately, most TMON users seemed to use EUA, since it was in the public domain.

As even more time went on, other changes were made to the Macintosh architecture as well. Modifications such as Levco's Prodigy 4, with its 68020 microprocessor, 68881 math coprocessor, and four megabytes of RAM became popular as development systems because of their speed and power. Several alternative screens also appeared on the market for those who wished for a larger screen on their Macintosh. Earlier versions of TMON did not work well on large screens, and did not work at all on machines with a 68020 microprocessor, such as the new Macintosh II.

For these reasons we are proud to offer you TMON Version 2.8. It is fully compatible with all Macintosh architectures with at least 512K of RAM with the sole exception of the Macintosh XL, and is compatible with all of the currently available Macintosh microprocessor and ROM configurations. It is also compatible with most available third party large screen modifications. (If you find that TMON does not work properly on your hardware configuration, please contact us and let us know what is non-standard about your system so that we can ensure compatibility with it in a future release.)

We regret that you find it necessary to use TMON (no one likes to chase lurking bugs), but we are glad that you have chosen TMON as your tool. We hope that TMON saves you effort that would be best applied to other areas of the development cycle. Who knows, TMON might even prove fun to use!

Before I begin, I'd like to point out that when I use the third person in the English language, I use the English language as it was taught to me, which is to say that I use "he" to refer to either males or females, since users of TMON can obviously be either. If someone can provide a graceful gender neutral substitute, please do. I'd love to hear it. In the meantime, "he/she," "him/her," "his/her," etc. just don't cut it.

I'd like to take this opportunity to thank Jay Zipnick, Bill Leininger, Darin Adler, and Dave Feldman for reviewing this manual and offering many valuable comments, most, if not all, of which you will find incorporated here. A special thanks is also due to our beta testers, without whom TMON would not be as reliable as it is.

Now, allow us to show you how TMON can help you eliminate bugs in your software...

A Few Words About the Macintosh Keyboard

Before I talk about debugging I should point out that debugging a piece of software on any computer inevitably becomes a keyboard-intensive task. This is even true on computers such as the Macintosh, which is normally a mouse-intensive machine. It's probably a good idea to take this opportunity to discuss some aspects of the Macintosh keyboard that may not be immediately obvious.

Perhaps the most significant key on the Macintosh keyboard whose function is not always obvious is the **Command** key, which looks like a cloverleaf (⌘). On some keyboards this key also has the outline of an apple on it. This key is one of the *modifier keys*. Modifier keys are keys that somehow change the behavior of another key. They normally do this when they are held down and another key is pressed at the same time. The most obvious example of a modifier key is the **Shift** key. You hold down **Shift** and press another key, and **Shift** modifies the behavior of the other key by making it generate an uppercase character, as opposed to a lowercase character.

The **Command** key does what its name implies; it causes the other key to execute some command associated with it. Examples will be given later.

Another modifier key is the **Option** key. Holding down the **Option** key and pressing another key will also generate some other character than what you expect. **Option** is also used with the **Interrupt** button on the programmer's switch; this function will be explained in detail later.

Some Macintosh keyboards have a key that is labeled "Control." In the early days of microcomputers the **Control** key served much the same function as the **Command** key does today. It was used primarily to issue commands to a program. On the Macintosh the **Control** key is used to create still another range of ASCII characters on the keyboard.

As a TMON user, you will find **Option** very useful for getting into TMON, and you will find the **Command** key (hereafter referred to as the "⌘" key) useful for doing the same things that you can do with TMON's buttons, which will be explained later.

What is a Debugger?

There comes a time in the development cycle of any but the most trivial of computer programs that problems will arise (although students of Hoare and/or Dijkstra will argue this point). These problems, referred to as "bugs," come in many shapes, sizes, flavors, and species (but they're all ugly). Bugs can be divided into two major categories: design flaws and implementation flaws.

Design Flaws

These bugs can be quite insidious, or they can merely be the cause of the programmer hitting himself on the forehead and mumbling vaguely about stupidity. They are caused by implementing a solution to a problem that is just plain wrong. Unfortunately, within the context of the current state of the art, the only debugger capable of detecting this type of bug is the human brain. ICOM Simulations, Inc. is not currently marketing this type of debugger.

Implementation Flaws

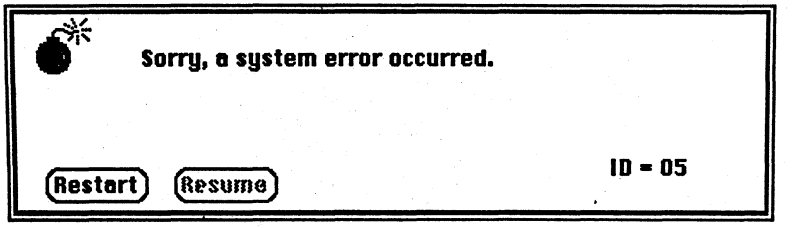
These bugs can also be quite insidious, or can be the cause of minor self-inflicted headaches as noted above. Unlike design flaws, the problem with implementation flaws is that while the solution designed for the problem at hand is perfectly fine, the manner in which the solution has been described to the computer is faulty. Unfortunately for the programmer, these problems can manifest themselves in many ways, some of which may bring the computer totally to its knees. (There is at least one error which Macintosh assembly language programmers can commit which will cause an instantaneous system reset—talk about being tough to track down!)

What Are Some Common Macintosh Bugs?

Well, before we can answer this question, we need to provide a bit more clarity in describing just what some bugs' symptoms are. There are two major categories of Macintosh bugs: Dire Straits bugs and everything else.

Dire Straits Bugs

Occasionally in the course of using a program on a Macintosh the user is treated to a unique display which consists of a small rectangular window containing an icon of the old stereotypical bomb (a black ball with a burning fuse), a message saying "Sorry, a system error occurred," and a mysterious message, "ID=*xx*," where "*xx*" is essentially any number from one to thirty-three (there are more of these little beasties, but I have yet to actually *see* any number higher than thirty-three). One of these boxes looks something like this:



Note that there are two buttons on the box, one that says "Restart" and one that says "Resume." Many a Macintosh user has complained that the "Resume" button is usually grayed out and unusable, forcing them to re-boot the machine as if they were turning it on for the first time. In reality, when you see one of these things, it *can* mean that the system is in such a mess that any attempt to do anything other than restart *could* result in "unpredictable behavior," including the irrevocable loss of everything on your disk. Restarting may just be your best bet.

The important thing to realize about "Dire Straits" errors is that they are bad enough that the system tries to catch them itself and, if successful, brings the normal operation of the computer to a standstill. Oftentimes the user's only recourse is to start over from scratch and hope that the problem doesn't persist. Every so often a program will actually enable the "Resume" button (it's the application's responsibility to do that) and try to make it do something at least moderately helpful, such as closing all currently open files and exiting to the Finder (whether this is advisable or even possible, depending upon the current status of the system, is open to debate).

Everything Else

By definition, "everything else" bugs are a lot more common than bugs that manifest themselves by bringing the system to a standstill (although there are some problems that can bring the system to a standstill *without* being caught by the system and stopped via the Dire Straits alert box). One problem that was quite prevalent in very early Macintosh software was one that caused the display to behave in a *very* bizarre fashion and usually caused the Macintosh sound hardware to create sounds somewhat akin to machine gun fire. In general, these are symptoms of a problem that proves fatal to the machine, although no Dire Straits alert box is ever produced. The system merely dies miserably. Many other bugs are possible, most of which cause the program to function improperly, but do not cause the machine to give up the ghost. For the most part, though, "everything else" bugs will cause the program to behave incorrectly, but it will be entirely up to you to track down the cause of the problem and solve it.

Now we can answer the question on the previous page, namely "What are some common Macintosh bugs?" Perhaps the most common Macintosh bug is the one that causes the Dire Straits alert box to show up with an ID=2. This means that the 68000, which requires data of more than one byte in length to be at an even address, tried to access more than one byte from an odd address. This alert is quite common in programs written by beginning Macintosh programmers; it even crops up in experienced Macintosh programmers' code from time to time. It's easily recoverable in almost all circumstances; returning to the Finder somehow is usually sufficient and even safe.

Another common Macintosh boo-boo is to ignore disk inserted events, figuring that the user will never have either the need or the opportunity to insert another disk. This is not the case; in the vast majority of applications (i.e. those that use GetNextEvent) the user can use ⌘-Shift-1 or ⌘-Shift-2 to eject a floppy (unless your application somehow disables the function keys, a difficult, extremely obnoxious, but not impossible thing to do). The user can then insert a different disk, and it may never even have been initialized! Your application should handle disk inserted events correctly, or problems are guaranteed.

Unfortunately the most common Macintosh programming error, passing one or more bad parameters to a ROM trap, cannot be defined or explained within the scope of a single paragraph, or even within the scope of a single book! All that this error means is that a) the programmer got cocky and thought that he could remember the parameter list for a particular ROM trap when in reality such perfect knowledge escaped him, or b) the bad parameter was a handle that hadn't been dereferenced or something along those lines. The solution to a) is to look up the argument list for a ROM trap if there's any question at all as to what it is. That's what *Inside Macintosh* is for! The solution to b) can be quite difficult to find, and that's what debuggers are for.

Now that we have defined our terms more clearly, we can begin to answer the question posed a few pages back: What is a debugger?

A debugger is a tool that assists the programmer in tracking down implementation flaws. That is, a program, once designed and written, is exhibiting undesirable behavior, whether system-killing or otherwise. A thorough examination of the program has convinced the programmer that there is nothing wrong with the program's concept, therefore the problem must lie in the program's implementation. (Note that many programmers will insist that, at first glance, there's nothing wrong with the implementation, either. This attitude may even last through the second or third glances at the code! How many times have you heard a programmer utter that famous phrase, "It must be a hardware problem?")

The debugger exists in the system as a piece of software which is somehow out of reach of the normal operations of the computer. It generally reserves memory for itself in some special way so as not to interfere with the computer's normal way of doing things (and also so as to avoid being damaged if the computer's memory management scheme should be broken by whatever bug the debugger is looking for). It also intercepts almost all of the computer's mechanisms for dealing with error conditions; the debugger provides tools for error tracking and recovery that are considerably more robust than the computer's own (surprising, but true)!

So, in summary, the debugger is a special program that lies "beneath" the built-in software of the computer (in the sense that, ideally, the debugger is invisible until/unless it is asked for by the programmer or a system error occurs, whichever comes first). When the debugger is called upon, it places the programmer in an environment which is conducive to finding errors in the problem program.

Why Should I Use TMON?

Good question! Hopefully we'll be able to provide some thought provoking answers. As part of providing those answers, perhaps we should first answer the question...

What is TMON?

We could cop out at this point and say that TMON is a Macintosh debugger, but we're not generally in the habit of insulting people's intelligence. TMON is a highly sophisticated interactive (we'll explain what that means in a moment) multi-window (we'll explain why this is neat later on, too) symbolic (this is particularly nifty) debugger for the Macintosh *family* of microcomputers.

TMON is interactive. This goes partially hand-in-hand with being multi-window; TMON doesn't lock you into any particular mode at any particular time. If you're looking at a disassembly of part of a program and suddenly decide that a breakpoint at a particular spot would be helpful, you don't have to leave the disassembly or worry that it will scroll off the screen. Just open the breakpoint window, pick one of TMON's seven breakpoints, and enter the appropriate address into the breakpoint line. Not only will the breakpoint window show the address of the breakpoint, but an asterisk ("*") will appear next to the breakpointed instruction in the disassembly window to indicate that execution of the program will stop at that point due to a breakpoint being set there.

TMON is a multi-window debugger. The Macintosh is a multi-window computer, so why not have a multi-window debugger? The same thoughts that went into making the Macintosh itself easier to use went into making TMON easier to use. TMON can have several windows open at one time, up to a maximum of nineteen. Most TMON windows can be duplicated, i.e. there can be more than one of their type open at one time. For example, you may wish to have two or three disassembly windows open showing disassemblies from different addresses. This is fine with TMON. Once a window of a given type is open, to open any more you must hold down Shift while opening the window. A window can be opened by clicking on the appropriate button at the top of the screen or by holding down the "⌘" key and typing the first letter of the button (e.g. ⌘A for an "Asmbly" window). There are four types of windows, however, for which multiple copies would not make any sense. These are the breakpoints window (TMON has seven breakpoints, period), the registers window (The 680x0 family of microprocessors has only so many registers, period), the options window (options are global), and the user window (TMON can only support one user area at a time).

TMON's windows are a little different in appearance and operation from normal Macintosh windows. There is a good reason for this. TMON, a debugger, must be as independent of the Macintosh operating system as it possibly can. This means that it cannot expect the Macintosh operating system or toolbox to be in a reliable state, because if some important aspect of the Macintosh operating system were to break under the weight of a large bug, so would TMON! That would be intolerable. Therefore TMON implements its own windows, handles the keyboard invisibly to the event manager, and so on. So bear with us if the somewhat strange windows take some getting used to.

How are these windows different? Well, for starters, they cannot be resized in the horizontal direction—they are always 512 pixels wide, like the classic Macintosh screen. For another thing, they can be dragged by clicking just about anywhere, not just in the title bar (besides, TMON windows don't even *have* an obvious title bar). The TMON windows' scroll bars have no "thumb" (the little white box between the arrows) and no "gray area." Actually, there are good technical reasons for the scroll bars being the way they are. The upshot of not having thumbs or gray areas is that the window can be dragged by this area as well, with an interesting difference from being dragged by other areas; dragging by the scroll bar will *not* bring that window to the front.

TMON is a symbolic debugger. This means that you can refer to the code that you are debugging by whatever you chose to call it when you created it (within certain limits). There are two ways that TMON can determine what label to assign to a particular range of code: embedded labels and .MAP files.

Embedded labels are labels that are actually built-into the program in such a way as not to interfere with program execution but to allow identification of particular subroutines in the program. (Specifically, they are embedded immediately after the UNLK and RTS or JMP instructions that end a function or procedure.) These labels will be a maximum of eight characters long. Some compilers have an option to include these labels. This capability was first found in the Lisa Pascal compiler that most early Macintosh programs were written

with. Several other development systems created since then, such as Apple's popular Macintosh Programmer's Workshop, support this option as well.

TMON will also read .MAP files which have been generated by the MDS and Consulair Mac C systems, as well as .MAP files created by some other development systems, such as TML Pascal. This also provides names for procedures and functions, with an added bonus that embedded labels don't share; since the .MAP file is generated by the linker, it contains labels for the libraries that the compiler uses, whereas these labels are not embedded in the code.

Regardless of how the labels are stored, they can be used in any expression evaluation function anywhere in the system (expressions in TMON will be explained in detail when we discuss the Number window). For example, it is common, when programming the Macintosh in a high level language like Pascal, to have a procedure which is capable of dealing with a long integer, half of which is a menu ID and half of which is a menu *item* ID. If expressed in this fashion, this routine can be used either as a menu handler or a ⌘ key equivalent handler. If the procedure is called DoCommand, and the label has been entered into TMON either by being embedded after DoCommand's code, by being read in from a .MAP file, or by being manually entered, then it is perfectly valid to think the following:

Since menus tend to be a central part of any Macintosh application, it might prove useful to set a breakpoint at the point at which menu handling begins. Ideally, this breakpoint will also catch the use of ⌘ key equivalents, although this may or may not be the case; it's application dependent (if you used something like DoCommand, it *will* be the case). Normally what you would have to do is:

1. Enter the application.
2. Enter TMON by holding down **Option** and pressing the **interrupt** button.
3. Find the code that looked like the DoCommand code.
4. Open the breakpoint window.
5. Type in the hexadecimal address of the beginning of DoCommand.

With the symbolic nature of TMON, though, it's much simpler:

1. Enter the application.
2. Enter TMON by holding down **Option** and pressing the **interrupt** button.
3. Open the breakpoint window.
4. Type in "DoCommand" with the quotes.

Both of these will set the same breakpoint, but one requires that you go searching your code for the appropriate address. Not only that, it assumes that you can tell what your source compiled to.

☞ You can use a label anywhere an expression is being evaluated. Breakpoint setting, disassemblies, dumps, the number window—all are candidates for having a label typed in to them. Anywhere you can use an address, you can use a label. Be sure to type in the quotes so that TMON knows that you're referring to a label.

Just to make sure that it worked, you may wish to do something that will invoke a menu function (select something from a menu or press a ⌘ key equivalent). Once the choice has been made, control will pass to DoCommand to deal with it, at which point the breakpoint that you set will drop you into TMON, from whence you can do great things, like single step, trace, etc. to see what's going on. Neat, huh?

TMON was designed with features that had never been available with any Macintosh debugger. One example of this is the "trap discipline" function. The concept of trap discipline first appeared as a standalone application written by Steve Capps, a member of the original Macintosh team at Apple and one of the authors of the Finder. This application checked some of the more arcane parameters of some of the more arcane traps and reported things that it found out of line. Darin Adler took this concept, expanded it to cover virtually *all* parameters of virtually *all* traps, and gave it two strengths—lenient and strict—to make it easier to determine whether a parameter was way out of the realm of the real or whether it was just questionable. Using discipline, it is possible to catch many errors before they happen—passing a NIL handle to TEIdle, for example. It is features like trap discipline that set TMON apart from any other Macintosh debugger currently available.

Installing, Entering, and Leaving TMON

Well, here it is: the portion of the manual that you've been waiting for. Hopefully now you have some idea as to what using a debugger is all about, and you also have an acquaintance with what TMON is and how it differs from other Macintosh debuggers.

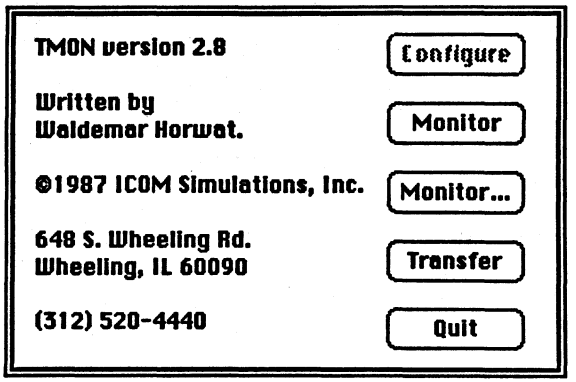
Now it's time to take a look at TMON in more detail. We'll explain how you install it, how you can get into it, and how you can get out of it. In the sections following this one we'll explain each function.

Loading TMON

The first thing that you have to know is how to get TMON into your system. TMON is provided on a 400K floppy disk that is *not* copy protected. One upshot of this is that TMON can easily be copied to wherever you need it; another upshot is that TMON can easily be pirated. Please respect our not giving you a copy protection hassle by not giving us a lack-of-income hassle.

TMON can be loaded two ways. For occasional use, it can be started as an application. This is described below. For day-to-day use, it is best to put the TMON application, along with a file called "TMON Startup" (included on the TMON disk), in your system folder. This is described in more detail in The Startup Loader section of the *Technical Reference*.

The TMON application is, by all outward appearances, a normal Macintosh application. It is executed by clicking on TMON and choosing "Open" from the "File" menu or by double-clicking on the file. The TMON application will be loaded. If Option, ⌘, Shift, or the mouse button are being held down, or if any combination of these are held down, TMON will display this dialog box:



This dialog is "command central," if you will, for TMON. It allows you to configure TMON to your liking (Configure), enter the Monitor (Monitor), enter the Monitor after choosing a particular user area (Monitor...), Transfer to another program (Transfer), or quit to the Finder (Quit). TMON comes from us configured to what we hope is a useful set of default conditions. You can, however, change them and save them as your customized user area file.

If you do *not* hold the keys or mouse button down, TMON will look in the current directory (the disk that contains TMON for MFS users; the folder that contains TMON for HFS users) for a file called "User Area." If it finds one, it will try to load it as the default user area. If there is no file called "User Area" available to TMON, it will use the user area that is built-in.

The user area is the customizable portion of TMON; it can be written or rewritten by you or some other third party as needed. One of the things that the user area can be configured to do is force TMON to quit to the Finder immediately upon loading. This is so that TMON can be made the startup application and be made to install itself without any intervention on your part; this is also why it is sometimes necessary to hold down the Option or ⌘ key or the mouse button in order to see the main dialog.

Probably the most commonly used function on this dialog is the "Monitor" function. Clicking on this button will cause the debugger to be installed along with the appropriate user area. Once the debugger has been installed, control will pass to the debugger. We'll explain how the debugger looks in a moment.

The "Monitor..." button is for power debugger-users who use more than one user area. If you click on it, the Standard Get File dialog will come up and allow you to select which user area file you wish to load when the debugger is installed. That user area will then be installed with the debugger, and control will pass to the debugger just as it does if you choose "Monitor."

The "Transfer" button was originally intended to allow 128K Macintosh users to load TMON and then transfer to another application, since on 128K Macintoshes there were many circumstances under which TMON would take up too much memory to allow the Finder to run! We've left the "Transfer" button in as a convenience; if you click on it, a Standard Get File dialog appears which allows you to choose the application. Note that the TMON "Transfer" function does not switch to the System file on the new volume (assuming that the volume is different and contains a System file).

Clicking on the "Quit" button will return you to the Finder. A great deal of debate was involved in the decision as to whether to include this feature or not (just kidding, folks, just kidding)!

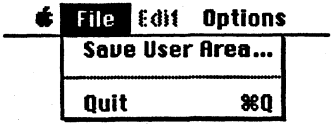
The "Configure" function takes you into another section of the program that deals with all of the things about the debugger that can be easily changed. In the dialog box shown above, the "Configure" button has been disabled. This is because the debugger hasn't even been loaded yet, and the program doesn't know which user area file to change this information in (remember, the things about TMON that can be changed are stored in the user area file). For detailed information on how to load the debugger, read the "Getting Into TMON" section. When the debugger has been loaded, this button will become enabled. When you click on this button, an empty screen with a menu bar appears.

Configuration

The configuration menu bar consists of four menus. The first three are the ones that Apple says should be in every Macintosh application (and we try to follow the rules as much as possible—that's why these menus are there). They are the Apple menu, the File menu, the Edit menu, and the Options menu.

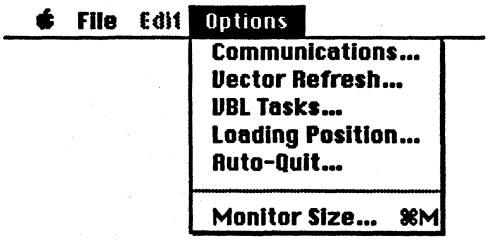
The Apple menu contains the desk accessories that you have installed in your System. They are there so that you can use them if you need them (you may wish to note your configuration for a particular user area in the Note Pad desk accessory, for example).

The File menu contains two items: Save User Area and Quit. Save User Area allows you to save your configured user area on disk. Quit returns you to the main dialog. Here's what the menu looks like:

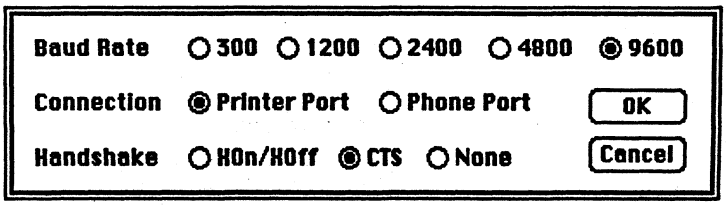


The Edit menu is there for the sake of the desk accessories. It won't even become enabled unless you open a desk accessory.

The Options menu is the one that allows you to define how you wish TMON to work. Let's look at it now:

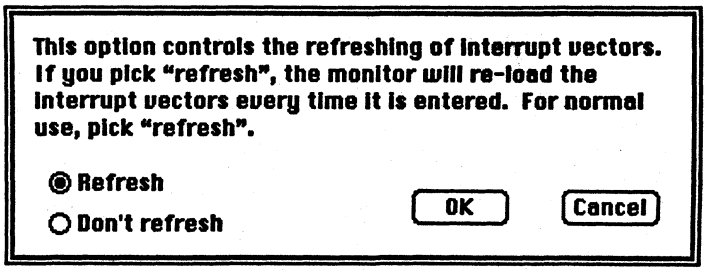


The Options menu has seven user-configurable items. The first is the communications settings.



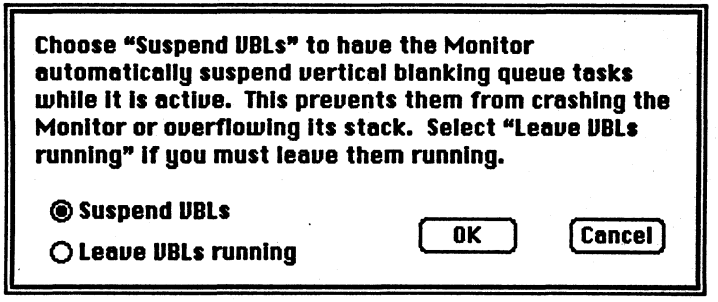
The purpose of the communications dialog is to allow you to define how TMON will communicate with a device that is connected to one of the Macintosh's serial ports. Most of the time this device will be a printer. The dialog above shows the default values, which are fine in many cases, and are correct for the ImageWriter and ImageWriter II. When you use any of the printing features of TMON, it will attempt to send its output to the port specified by this dialog, using the baud rate and handshaking protocol specified here as well.

Vector Refresh is the next option. Its dialog looks like this:

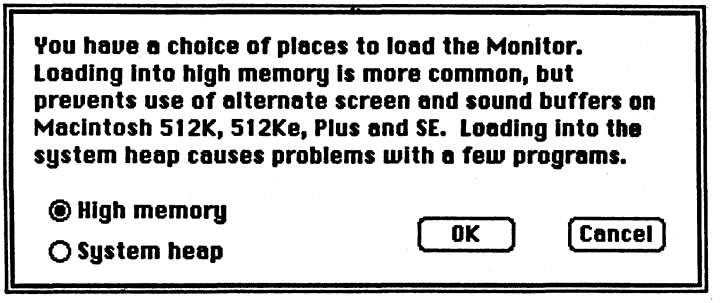


The 680x0 family of microprocessors has several locations in low RAM (called *vectors*) which are used to define the behavior of the system under certain conditions. Debuggers store addresses of their internal routines in these vectors so that they have control of the system if some special circumstance arises. TMON in particular not only puts addresses there when it is loaded, it also puts its values there *every time the Monitor is entered.* This is so that, in the event of a crash of some kind, the debugger's values will be in the correct locations if the debugger can be entered at all. Some programs store values in these vectors and will not work properly unless these values are allowed to stay. For this reason, you can tell TMON *not* to refresh those vectors every time it is entered. The default is *not* to refresh the vectors, since a few popular applications (e.g. MacWrite) use them.

The next item is one that allows you to define whether vertical blanking (*VBL*) tasks are left running while TMON is active or not. Previous versions of TMON left VBL tasks running, which proved useful in getting screen snapshots from within TMON but left a great deal to be desired for those who were trying to debug things that depended upon VBL tasks (it's historically been very tough to debug things while they're running). In particular, AppleTalk development with TMON while VBL tasks are running can be a challenge. Also, some VBL tasks expect not to be in a monitor, or use too much stack space (TMON doesn't offer very much). Such VBL tasks can cause system-crashing damage if left running while TMON is active. In order to provide the user with a choice of whether to leave VBL tasks running or not, we have provided the following dialog:

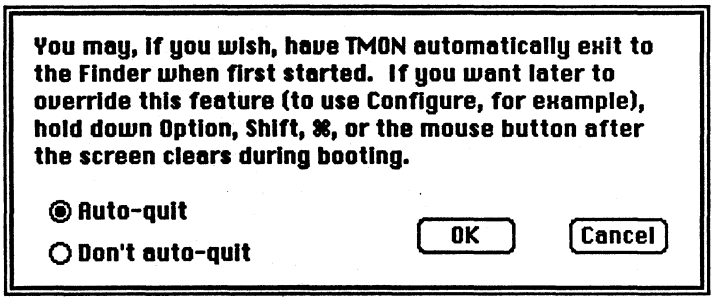


Next is Loading Position. This allows you to choose whether you want the debugger to live in the system heap or in high RAM. Its dialog looks like this:

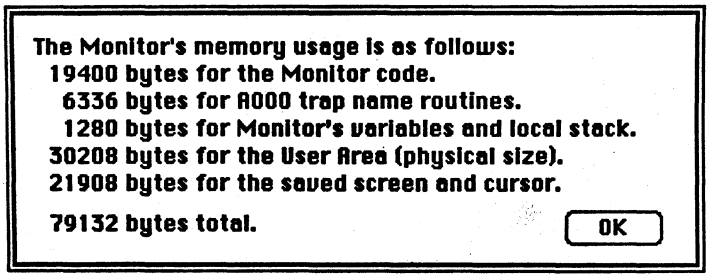


As the dialog indicates, it is more common to load the debugger into high RAM than it is to load it into the system heap, although each has its advantages and disadvantages. Loading into high RAM causes the alternate screen and sound buffers to become unavailable, and loading into the system heap causes some applications to malfunction. Specifically, applications which use _SetTrapAddress to change a ROM trap to point to the application heap zone will fail if running on a 64K ROM machine with TMON installed in the system heap zone. The reason is that on 64K ROM machines trap addresses had to be within 32K of either the beginning of ROM or the beginning of the system heap zone, depending upon whether they were in ROM or RAM. Installing a routine in the application heap zone may or may not have worked—it was dependent upon whether the address was within 32K of the beginning of the system heap zone or not. With TMON installed in the system heap zone, the application heap zone is *never* within 32K of the beginning of the system heap zone, and the faulty application would fail miserably and mysteriously. Note that since 64K ROM machines are becoming so scarce, this is generally not a problem anymore, and it seems preferable to have TMON installed in the system heap zone so that the alternate screen and sound buffers are available to applications which wish to use them.

The last option that changes the way in which TMON works is Auto-Quit. Auto-Quit allows you to define whether TMON goes to the main dialog when it is loaded, or whether TMON simply installs itself and immediately goes back to the Finder. The default is to auto-quit so that TMON can be the startup application on your disk and be loaded every time you boot your system without any intervention on your part. The dialog for this option looks like this:



I said that the Auto-Quit option was the last one that allowed you to change TMON's configuration, and it is. The last item on the menu, Monitor Size, is an informative one rather than one that causes a change. Monitor Size provides information regarding the amount of space that the debugger *currently* takes up; that is, it shows the information *before* any changes are made (to see the effects of your changes on the size of the debugger, save your changes to a user area file, restart your system, load TMON with your customized user area, and *then* use Monitor Size). Here is an example of what Monitor Size shows:



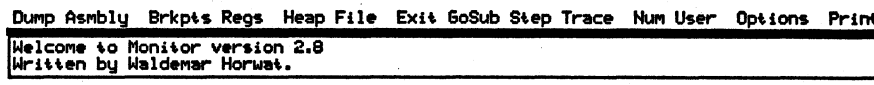
Now you know all about what lies beneath that "Configure" button on the main dialog. Note once again that this button will *not* be enabled unless the debugger has already been loaded into memory (which means that TMON knows what user area to modify with your configuration).

Remember that once you have made your changes you must use the Save User Area option in the File menu to save your configured user area to disk!

Getting Into TMON

Now that you know how to configure the debugger, it's time to talk about the "Monitor" and "Monitor..." buttons, as well as system errors and the Interrupt button on the programmer's switch. First let's talk about the "Monitor" button. Actually, we should talk about the "Monitor..." button first, or at least at the same time as "Monitor" since they both do the same thing with only one difference. "Monitor..." brings up a Standard Get File dialog and allows you to choose a user area file to use when the debugger is installed. "Monitor," on the other hand, simply installs the debugger according to the configuration stored in the file called "User Area" (if TMON found one) or according to the built-in default user area (if there is no "User Area" file).

In any case, when you click on either "Monitor" or "Monitor..." the display will look something like this:



Initially, this is all that you see from TMON: a "button bar" (we'll talk more about that later) and a window indicating what version of TMON you are using and mentioning the fact that Waldemar Horwat wrote it (yes, Waldemar Horwat does exist).

Note that once TMON has been initialized in this fashion or installed automatically, there are a few ways that it can be entered. One is by causing any error that would normally result in a "Dire Straits" box. Another is by pressing the **interrupt** button on the programmer's switch that Apple told you to install only if you were a developer. Still another is to put a _Debugger trap at a point in your program at which you want to enter TMON. Entering TMON due to a "Dire Straits" error will present you with diagnostics relating to the error; entering by pressing **interrupt** will present you with a message to the effect that an interrupt has occurred. There are variations on the **interrupt** method that I will discuss later.

Getting Out of TMON

Terrific. You can load the debugger, configure it, and get into it in one of basically five ways ("Monitor," "Monitor...," a system error, **interrupt**, or a _Debugger trap). Once you're in it, how do you get out of it? And how do you get rid of it completely?

To leave the debugger and go on about your business, click on the "Exit" button from the button bar or press ⌘, hold it down, and press the "E" key (for Exit). TMON will put you back wherever you were when you entered TMON (assuming that you haven't done anything to change TMON's perception of where you were. We'll talk more about that later).

The only way to get rid of the debugger completely is to re-boot your machine and not let it load TMON. Once TMON has been installed, nothing short of re-booting will get rid of it.

The Monitor Environment

This section describes the Monitor. The Monitor is the part of TMON that is installed into the machine. The Monitor has its own user interface, with a button bar, windows, and special conventions for user input.

The heart and soul of the Macintosh is its user interface. The Macintosh embodies user interface concepts that were *avant garde* at the time that they were created—in the early 1970s. At the time, though, the hardware to do the things imagined at Xerox PARC (Palo Alto Research Center) took up entire tables, not one and one half square feet of desktop. Since hardware has improved in price vs. performance since then, examples of the PARC-style user interface are becoming more common.

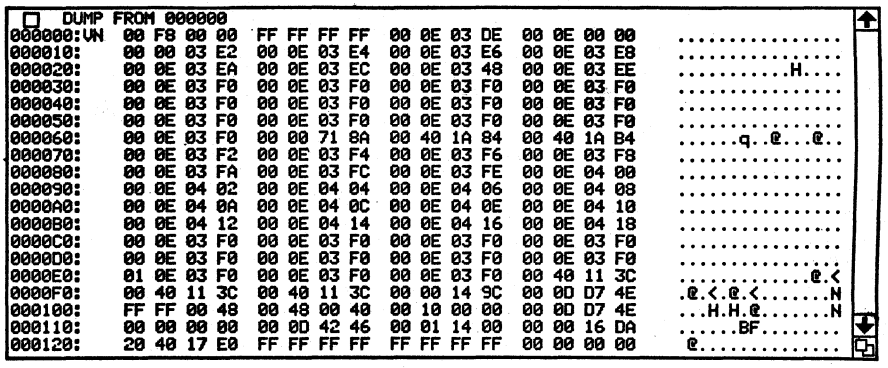
TMON, as was mentioned above, follows the Macintosh user interface guidelines fairly closely, although it can't follow them completely. If a debugger is to be able to go anywhere, including wandering around in the operating system, it needs to be as independent of that operating system as it possibly can—particularly if the operating system is not re-entrant, i.e. was not written with the idea of having portions of it being executed by more than one process at a time. Since TMON had to implement its own versions of things like windows, it seems like a good idea to explain these things briefly.

The Button Bar

The button bar lies at the top of the screen, where the menu bar normally lies for Macintosh applications. When you click the mouse on one of these, instead of pulling down a menu, it performs a particular function. Many of these buttons open a window. Some of them perform some specific action that doesn't require a window. We'll look at the features of each button a bit later, but not necessarily in left-to-right order across the button bar.

The Windowing System

As was mentioned earlier, TMON's windows are a bit different from what you're used to on the Macintosh. Let's look at a typical TMON window and see how it differs from what you'd expect:



This, obviously, is a dump window. It was created either by clicking on the "Dump" button on the button bar or by holding down the ⌘ key and pressing the "D" key.

At the top of the window it says "DUMP FROM" followed by, in this case, an address which happens to be 000000. This is as close as TMON windows get to having a title bar. Don't be fooled by my calling it a title bar, though; its purpose isn't the same as a normal Macintosh window's title bar's. There's nothing that says that you have to drag a dump window by its title bar; you can drag it by practically anything. Specifically, TMON windows can be dragged by their title bars, their contents, or the area in the scroll bar between the arrows. The places that dragging will not work are in the close box, the grow box, and the arrowheads. As an added bonus to all of this, dragging a window by the scroll bar area will *not* bring the window to the front. This makes it easy to reorganize without disrupting your active window.

Speaking of active windows, TMON windows do not have the normal hilighted/unhilighted appearance that the Macintosh normally uses to indicate which window is active and which are not. Instead, any given TMON window may have a vertical blinking bar. Whichever window has the bar is the active window; all others are inactive. By the way, all that "active" means in the context of TMON is that when you type, the information will be put wherever the blinking bar is in that window (assuming that that is possible—typing ASCII that contains invalid hexadecimal characters when the blinking bar is in the hexadecimal portion of the dump window will accomplish nothing). However, just because you can't type in the inactive windows, don't assume that they're sitting there doing nothing! *All* TMON windows have their contents updated continually so that as the state of the system changes the windows are updated to reflect that change.

Almost all TMON windows have a close box. The exception to this rule is the special message window which occasionally appears near the top of the screen. It has no close box; instead it disappears upon the first mouse click or key press after it appears. Special messages are things like the welcoming message when you first enter TMON or notifications of what system error caused TMON to be entered. Clicking on the close box makes the window disappear, just as you would expect.

Another common element of most TMON windows is the grow box. It too behaves as you would expect, with one crucial difference: TMON windows are *always* the width of the original Macintosh's 9" built-in screen. In other words, TMON windows can only be resized vertically, not horizontally. Since information in TMON is laid out in a more-or-less linear fashion (i.e. line by line), it makes sense to control the number of lines displayed, but not as much sense to control how much of each line is displayed.

Typing

A couple of paragraphs ago, I mentioned typing information into TMON. In general, anytime there's a blinking vertical bar in a TMON window, you can type some kind of information there. It's up to the window to decide whether or not what you're typing makes any sense (invalid hex digits in a hex dump do not). Two things that hold true regardless of what window you're typing in. The line that the blinking bar is on is *not* continually updated along with the rest of the window (in order to prevent what you're editing from changing on you on the fly), and if you end your input by pressing **Return**, TMON will only change the data from the beginning of the line to wherever the cursor was when you pressed **Return**, whereas if you press the **Enter** key TMON will accept the entire field from beginning to end.

☞ This one is important! Reread what I just said. Anytime TMON is taking keyboard input, you can press **Return** if you want TMON to take everything *before* the cursor, or you can press **Enter** if you want TMON to take *everything* in the field, including what comes *after* the cursor. This capability is quite useful, and specific circumstances where it is useful will be explained as we get to them.

To position the cursor at the uppermost and leftmost position of the window, just press the **Tab** key. To delete a typed character use the **Backspace** (or **Delete**) key. Also, if you are typing something and wish to cancel the entire line's changes, just click the mouse on another line without pressing **Return** or **Enter**. The changes that you had typed will be ignored.

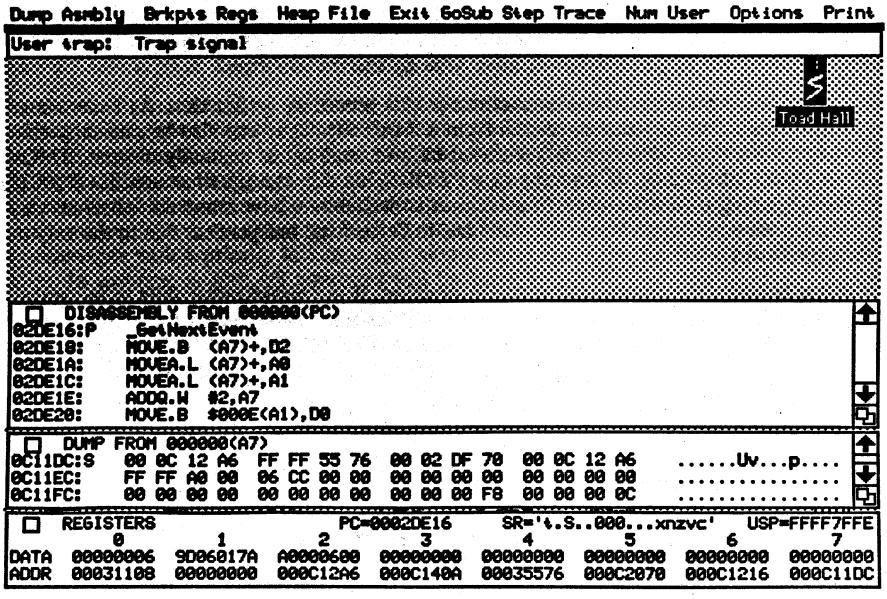
If you're using a numeric keypad, a Macintosh Plus keyboard, or a keyboard on any machine more recent than the Macintosh Plus, you have a few other options as well. The left and right arrow keys will move the blinking vertical bar left and right, and the clear key will clear the entire line.

Basic Features

Let's take a look at some of the most common and most useful TMON functions.

Assembly

This feature is simultaneously one of the most basic (in the sense that it's something that's frequently used and should be a part of any debugger) and the most complex (in the sense that some of its capabilities are not obvious and are rather sophisticated). Let's take a look at one possible "Asmbly" window.



Actually, I told a little white lie. This is obviously a dump of a whole screen, not just the "Asmbly" window. The "Asmbly" window is the one that says "DISASSEMBLY FROM 000000(PC)" at the top. It's been sized to show six instructions and moved next to a dump window.

There are several notable things about the "Asmbly" window. First of all, instead of being a disassembly from some arbitrary address, it is a disassembly from 000000(PC), or whatever the current PC value is plus zero. This capability is called "anchoring," because it "anchors" the window to a particular register so that whenever the PC changes, the disassembly window will automatically change to reflect the new disassembly range. There's nothing magical about this; to do this with a newly opened "Asmbly" window (which initially disassembles from address zero), you just click on the address given and type "(PC)" without the quotes. Since you didn't specify an offset, TMON will assume that you meant "000000(PC)" and update the "Asmbly" window accordingly. Note that you can anchor any dump or disassembly window to any register except SR, not just to the PC.

Speaking of registers, now is probably a good time to mention a crucial fact about registers in TMON. There are times when TMON expects you to refer to a register, and there are time when TMON expects a value. TMON is typeless; it can't tell the difference between a register and a value on its own, especially since the names of the 680x0 data and address registers are all valid hexadecimal values (D0-D7 and A0-A7). There are times when it becomes necessary to be able to avoid ambiguity. This is one reason that you can refer specifically to a register by prefixing its name with an "R," e.g. RD0, RA5, and so on.

Prefixing a register name with an "R" causes the register to be treated as a value, and the value is whatever the register contains at the time. The best way to see this is to open the "Num" window by clicking on the "Num" button in the button bar or by pressing ⌘N. Try typing in things like D0, A3, and so on. Now try typing RD0, RA3, and so on. You may want to open the "Regs" window to see why the "Num" window shows the numbers that it does when you use the "R."

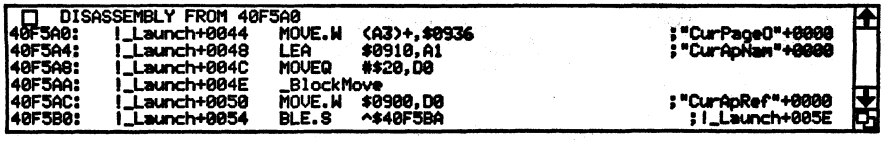
The next thing that I should point out is the column of addresses along the left side of the window. Those are the absolute addresses of the first byte of each disassembled instruction. In other words, the address for the first line is the current value of the PC (and if you look at the bottom window, which is the "Regs" window, you'll see that the value of the PC there is the same as the address in the "Asmbly" window, but we're getting ahead of ourselves by looking at the "Regs" window).

Note the "P" next to the first address. That "P" stands for "PC," and it's just a way that TMON has of letting you know that that's what the current value of the PC is. Of course, since this particular disassembly window is anchored to the PC, that "P" will always be by the first address.

In the next column are the labels for the instructions in this procedure or function. As I mentioned before, these labels can be built into the code or read in from a .MAP file. They can also be manually entered, although this is not often done. If no labels have been built-in, read in, or manually entered, TMON will allow you to refer to a piece of code by its resource type ("CODE," in this case), its resource ID ("0001"), and some offset from the beginning of the resource. In general, it is desirable to have some non-absolute way of referring to Macintosh code, since Macintosh code has to be position independent due to the way the Macintosh memory manager works.

The next column contains the disassembly of the instruction. If the disassembler fails to recognize a particular byte pattern as an 68000 instruction, it will display "????" as the mnemonic for the instruction. If it encounters a ROM trap that it doesn't recognize, it will say "ROM" followed by the hex value of the opcode. ROM traps which TMON does recognize are disassembled by name if you have configured TMON to do so.

The last column may have the label (with an offset, if necessary) which is used to refer to one or the other of the operands for the instruction. This way, when you see a reference to an address you'll know the name of the location. Here's what that looks like:

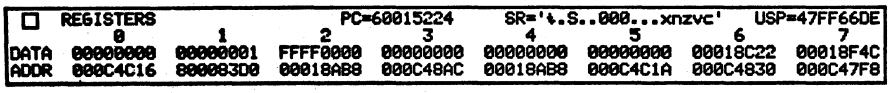


That takes care of the visible aspects of the "Asmbly" window, but there are a couple more things that need to be mentioned. First of all is that the scroll bar does work, and works fairly well in both directions. When you try to scroll an "Asmbly" window backwards, TMON guesses how long the previous instruction is. Occasionally TMON will misinterpret and get out of sync. Just keep going and TMON will get back in sync sooner or later. As an alternative, you can press the **Tab** key to get the cursor to the address line and press **Return**. That will advance the address by two bytes, forcing TMON to disassemble from that address. Continue this until you're back in sync.

Another invisible aspect of the "Asmbly" window is that it is, indeed, an assembly window! Not only does it disassemble what's at a particular address, it allows you to type in a new instruction which replaces the old one in RAM. Just position the cursor over the instruction to be replaced and type in the new instruction (using spaces between fields rather than tabs). This can be quite useful for applying quickie patches to a program during debugging (just make sure that any patches that solve a problem get implemented in the source code sooner or later)!

Registers

The "Registers" window allows you to view and change the contents of any of the 68000 microprocessor's registers. It can be opened either by clicking on the "Regs" button in the button bar or by holding down the "⌘" key and pressing the "R" key. The window looks like this:



This window shows the 68000 registers at a glance. The PC is the first register shown, and its field is where the cursor goes when it goes to the "top" of the window, i.e. when you press the **Tab** key the cursor will be positioned before the first digit of the PC value, making it easy to change the PC.

The Status Register is shown next. Since the Status Register is normally interpreted on a bit-by-bit basis, most of the bits have been given mnemonic character names in order to make their meaning more clear. The Status Register is displayed as a string of characters for that reason. If a character is lowercase, it means that the flag is reset. If the character is uppercase, it means that the flag is set. Periods mean "don't care" or "unused." There are three bits shown which are the exception to the character rule; in the example shown above they are all zeroes. This is the interrupt mask for the 68000. It holds the current interrupt level for the processor. Any interrupts with a priority less than this binary value (zero, in the example) will be ignored. Since the level in the example is zero, all interrupts will be handled.

In the example Status Register above, the trace bit is off, meaning that the processor will not generate a trace exception after each instruction; the supervisor bit is on, since historically the Macintosh has always run in supervisor mode; and all of the arithmetic flags (extend, negative, zero, overflow, carry) are off.

Next is the USP, or User Stack Pointer (as opposed to the Supervisor Stack Pointer, or SSP, which is shown in the window as "A7"). Although the Macintosh has historically functioned in supervisor mode, and the operating system software expected to be in supervisor mode, some applications may wish to enter user mode briefly, or future operating systems may work in user mode. If they do, the stack pointers will be swapped, so it is useful to be able to see the USP.

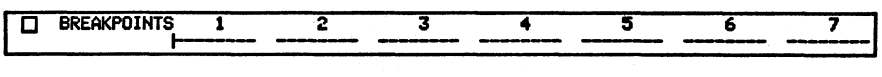
The remaining two rows in the window are the 68000's general purpose registers. They are divided into the eight data registers and the eight address registers. Note again that A7 is synonymous with the current stack pointer.

In the event that you need to change the contents of any of the registers, you may click somewhere within the value shown and enter any valid hex expression and press **Return**. The value will be assigned to the register. Note that this should only be done if you are *certain* that the register should be changed.

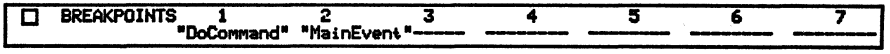
Breakpoints

Breakpoints are another feature that you will find in just about every debugger that's of any value. A breakpoint is something that you can put in your program so that when the computer tries to execute the instruction at that location, it passes control to the debugger. When it has done that, you can use the debugger to make sure that everything is in order. Breakpoints are a very useful tool.

TMON has one breakpoints window that can be opened by clicking on the "Brkpts" button on the button bar or by holding down the ⌘ key and pressing the "B" key. It looks like this:



The breakpoints window allows you to set up to seven breakpoints in your code. To set a breakpoint, click on the dotted line and enter something that evaluates to an address (it can be an absolute address, a label, or some other expression), and press **Return**. Note that multiple breakpoints can be set if the addresses are separated by a space. Here is what this looks like:



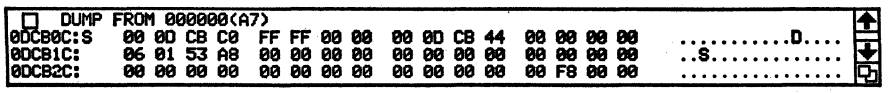
In the above example, the user is setting two breakpoints, one at the address with the label "DoCommand" and the other at the address with the label "MainEvent." Note that the labels are in quotes and are separated by a space.

Once your breakpoints have been set, you may exit TMON by clicking on the Exit button in the button bar or by holding down the ⌘ key and pressing the "E" key. When the program encounters a breakpoint, TMON will be entered with the PC value equal to the address that you set the breakpoint at. At that point you can examine or change values in registers, the heap, low memory, etc. Note that the instruction at the breakpoint has not yet been executed. Also note that in any open "Asmbly" windows that show the breakpointed instruction, an asterisk ("*") will appear immediately to the right of the address of the instruction. This is so that you can see breakpoints in disassemblies at a glance.

Breakpoints are cleared by entering a hyphen ("-") in the line with the address for that breakpoint. If you have more than one breakpoint set, you can clear any individual breakpoint by clicking at the beginning of its address and typing the hyphen, then pressing **Return**. As with setting multiple breakpoints, you can clear multiple breakpoints by separating hyphens with a space.

Dump

The ability to view displays of arbitrary areas of memory so that you can see or change what they contain is also a very basic debugger feature. To open a dump window, click on the "Dump" button in the button bar or hold down the ⌘ key and press the "D" key. A dump window looks something like this:



This is a small dump window; in fact this window is as small as TMON windows get (three lines of data). Dump windows show a combination of hexadecimal and ASCII data, with the hex on the left and the ASCII on the right. Note again that you can click on the line with the "000000(A7)" and type in any expression, including labels or addresses, or you can anchor the dump to a register so that every time the register changes the dump will follow suit. (In this example the dump is anchored to the stack pointer; this window will always show a dump of the top $30 bytes of the stack.)

A word about dumping application globals is in order. Application globals have negative offsets from the A5 register, and are usually accessed with the 680x0's register indirect with displacement addressing mode, in which case the displacement is a four digit hexadecimal value. However, due to the way TMON handle the sign bit, when using the dump window to look at globals the displacement must be expressed with the sign bit carried out to the 24th bit, e.g. FFFC42(A5) instead of the incorrectly interpreted FC42(A5).

Making changes in a dump window works much the same way as making changes in an assembly window. You can click anywhere within either the hex data or ASCII data and type in your changes. All of the normal rules of TMON user input apply. In addition, the dump window is smart enough not to allow you to type invalid hex data in the hex portion of the window (although you can enter any expression that evaluates to a hexadecimal value). Also be aware that the cursor does not automatically wrap around at the end of the line—you must explicitly press **Return** or **Enter**. Normally when you type ASCII in TMON it is converted to uppercase, but this is not true in the ASCII portion of the dump window—it is case sensitive.

Print

Actually, there are two sides to the printing coin. One is the "Print" button on the button bar. Clicking on it, or holding down the ⌘ key and pressing the "P" key, will cause the contents of the active window to be sent to whatever port using whatever protocol you chose in the configuration of TMON. Normally this means that the window will be printed on your ImageWriter printer (note that TMON, due to its nature, doesn't print to a LaserWriter via the AppleTalk network).

The flip side of this coin arises when you want a printout of a particularly long dump, disassembly, heap analysis, or resource file listing. Then you need to use the "Print" function from the user area. To do this, first click on the "User" button in the button bar, or hold down the ⌘ key and press the "U" key. You should see something like this:

```
□ USER area starts at $01DBBE.  Physical size is $7600; logical size is $7600
Toggle pages {memory functions}:
Block move (src dst len):
Block compare ○ (adr1 adr2 len):
Fill (bgn end val [vLen]):
Find {byte aligned} ○ (val [vLen [bgn [end]]]):
Template {WindowRecord} (addr):
Stack addresses ○ (addr):SP
Stack crawl ○ (addr):RA6
Load resource ○ (type ID):
Print {dump} {error=0000} (bgn end):
```

The "User" button brings up the closest thing to a menu that TMON has; it's a window that contains nothing but choices. This window is actually three pages, and the first line is how you get from one to the other. The function is called "Toggle pages". We don't need to toggle pages at the moment, however, since the "Print" option is on this window. It's the last choice listed above.

You'll notice the word "dump" in curly braces after the word "Print." Curly braces indicate a message that will change under certain circumstances. In the case of "dump," if you position the cursor after the colon on that line and press **Return** or **Enter** without providing any parameters, the message will change. These are your print options. In this case, the options are dump, disassembly, file, and heap. The error message also changes, but only as a result of trying to print something. You cannot change it manually.

Items in parentheses are parameters to the function. You must enter these in order for the function to work. These may also change as the nature of the function changes. For example, dumps and disassemblies require a starting and ending address, whereas resource file content listings require a file reference number, and heap analyses require a heap zone number.

To use the "Print" function, first position the blinking vertical bar after the colon on that line. Press **Return** or **Enter** until the proper type of operation is listed in the braces. Supply the parameters asked for by typing them after the colon, and press **Return** or **Enter**. That's all there is to it!

Step

The ability to execute a program literally one instruction at a time is another important basic feature of any debugger, and TMON is, of course, no exception. Stepping through the program can be accomplished by either clicking on the "Step" button in the button bar or by holding down the ⌘ key and pressing the "S" key. Stepping through the program causes TMON to pass control back to the application being debugged for the duration of one instruction. This way you can see the effect that any given instruction will have on the system.

Note that any windows open when you use Step will be updated to reflect whatever the instruction changes, thanks to TMON's constantly updated windows. So if you step through an instruction that changes some memory location, and that location currently appears in a dump window, then the dump window will change to reflect the new value in that memory location.

Actually, it is not quite true that stepping will execute one instruction at a time. If the instruction to be executed is a ROM trap, TMON will treat the trap as if it were a single instruction. In other words, using "Step" will not allow you to go through the ROM routines one instruction at a time. However, another function *will* allow you to do this. It's called...

Trace

Trace behaves in exactly the same fashion as Step, except that when ROM traps are encountered, Trace will let you step through the trap dispatcher and the ROM code itself. This is handy for seeing exactly what those ROM routines are *really* doing to your program (*Beyond Inside Macintosh*, as it were). You can use Trace by clicking on the "Trace" button in the button bar or by holding down the ⌘ key and pressing the "T" key. Other than the handling of ROM traps, Trace is identical in all respects to Step.

GoSub

The GoSub function is identical to Step except that any JSR or BSR instructions are allowed to execute indivisibly. This is handy when you are debugging something and you encounter a JSR or BSR to a subroutine that you are not interested in. You can simply GoSub and TMON will not regain control until the subroutine has returned. You can use this function either by clicking on the "GoSub" button in the button bar or by holding down the ⌘ key and pressing the "G" key.

Exit

The Exit function is one that has been touched on already, but the explanation of its purpose bears repeating. If you click on the "Exit" button in the button bar or hold down the ⌘ key and press the "E" key, TMON will relinquish control and let the 680x0 start executing code at wherever the PC currently points (hopefully somewhere in a relatively bug-free environment). This is useful for letting an application execute in an all-out fashion. (Well, almost all out; TMON still has its addresses stored in those low RAM vectors unless the application has overwritten them and you told TMON not to refresh them.) The exit function is used when a) the debugging session is over, or b) you have told TMON to regain control at a point that would take you hours to Step or Trace to. A good example of this latter use is to execute the program in full gear until a breakpoint that you've set is reached.

Block Move

Moving an arbitrary block of memory from one place to another is sometimes a useful thing to do from within a debugger. Among other things, it's handy for copying values from one variable to another (assuming that you can determine the addresses of the variables). Block Move is a user area function. It's found in the "memory functions" window. Click on the "User" button in the button bar or hold down the ⌘ key and press the "U" key. The user area window will appear. If the first line ("Toggle pages") under the title does not say "{memory functions}", click to the right of the colon on that line and press Return until it does. Just to refresh your memory, here's what the window should look like:

```
☐ USER area starts at $01DBBE.  Physical size is $7600; logical size is $7600
Toggle pages {memory functions}:
Block move (src dst len):
Block compare ↔ (adr1 adr2 len):
Fill (bgn end val [vLen]):
Find (byte aligned) ↔ (val [vLen [bgn [end]]]):
Template (WindowRecord) (addr):
Stack addresses ↔ (addr):SP
Stack crawl ↔ (addr):RA6
Load resource ↔ (type ID):
Print (dump) (error=0000) (bgn end):
```

You'll see that Block Move is the first function after the one that toggles the windows. You'll also see that it is a simple function that takes three parameters: the source address, the destination address, and the number of bytes to move. Type these three items after the colon on that line (click to the right of the colon to put the cursor there), and press Return or Enter. That's all it takes to move bytes from one place to another! Be sure that you know what you are doing with this function; it's basic, but powerful, and moving memory around at random not only *can* crash your system, it probably *will* crash your system! Note that all of the rules about entering addresses in TMON apply to user area functions as well (they can be any expression, including labels, etc.).

Block Compare

Block Compare is another user area function that compares two blocks of memory to ensure that they contain the same values. It, too, is a user area function, and is in the same window as Block Move. It takes the same three parameters as Block Move, namely a beginning address of the first block, a beginning address of the second block, and a byte count. Initially the curly braces for the Block Compare function are empty. Any mismatches will be reported there as "Mismatch at *xxxxxx/yyyyyy*," where "*xxxxxx*" is the address of the mismatch in the first block, and "yyyyyy" is the address of the mismatch in the second block.

Fill

The Fill function allows you to fill a block of memory with a particular value. This function takes four parameters: the beginning address of the block to fill, the ending address of the block, the value to fill with, and the size of the value. The square brackets around the fourth parameter mean that it is optional. If it is not specified, TMON will assume that it is only to use the least significant byte of the value to fill. Note that the size can only be one through four bytes. This function is also very dangerous, so use it with extreme caution! Like moving blocks to undefined areas, filling random memory blocks is a good way to crash a system.

Find

Find does exactly what it sounds like. It searches for a particular value throughout a particular block of memory, reporting any successful searches as it goes along. It is a user area function, and is in the "memory functions" window just like Compare and Fill. It takes four parameters: the value to search for, the length of the value, the beginning address to search from, and the ending address to search to. Note that all parameters other than the value itself are optional, but also be aware that user area function parameters are positionally dependent. In other words, if you want to specify a beginning address you must specify the length of the search value, since that parameter comes before the beginning address. Note that the only valid lengths are one through four. If you press Return without entering any parameters, Find will toggle between doing its searches on byte boundaries or word boundaries (word boundary searches are useful for finding 680x0 instructions, since the 680x0 requires its instructions to lie on word boundaries. They're also useful for finding things in data structures that you know are word aligned, such as heap blocks.)

If any matches are found, a message giving the address of the match will be displayed between the curly braces. The address will also be assigned to TMON's pseudoregister, called "V."

It is definitely worth talking about "V" now. "V" is a register that is set by many of the user area functions that return values. The find function is one of them. This is so that when you have used a user area function to determine a value you don't have to type that value in whatever other portion of TMON you are using—you can simply use V. For example, when you have found an address, you can open a dump window and type "V" as the address to dump from. Or you can anchor a window to V and watch your dump(s) and/or disassembly/disassemblies change as V changes. The usefulness of V has been underestimated by many people. It is a powerful tool for making use of information that user area functions return to you. Remember it.

Intermediate Features

Now it's time to explain some of the features that are not so common and which are, perhaps, slightly more powerful than the preceding ones.

Trap Intercept

Trap intercept is a user area function that allows you to specify a trap or range of traps that, if encountered within the specified PC range (or anywhere, if no range is specified), will cause the debugger to be entered. This function can be found on this "A000 trap functions" portion of the "User" window. First open the "User" window by clicking on the "User" button in the button bar or by holding down the ⌘ key and pressing the "U" key. If the first line ("Toggle pages") under the title does not say "{A000 trap functions}", click to the right of the colon on that line and press **Return** until it does. The window should look something like this:

```
☐ USER area starts at $01DBBE.  Physical size is $7600; logical size is $7600
Toggle pages {A000 trap functions}:
Trap record (t0 [t1 [PC0 PC1]]):
Record {} (fullStop nMsg [loc]):
Trap {heap check, scramble} (zone#):
Heap {} (zone#):
Trap discipline {lenient} (t0 [t1 [PC0 PC1]]):
Trap checksum (t0 [t1 [PC0 PC1]]):
Checksum (bgn end) {A426}:400000 41FFFF
Trap intercept (t0 [t1 [PC0 PC1]]):
Trap signal (t0 [t1 [PC0 PC1]]):_GetNextEvent _EventAvail
```

Trap intercept is the second function from the bottom; it takes at least one parameter, with the remaining parameters being optional. Note that when a user area function requires a trap or trap range, you may simply type in the trap name preceded by an underscore ("_"), just as the trap name appears in an "Asmbly" window. See the "Trap signal" function in the window above for an example of a trap range (_GetNextEvent through _EventAvail). Often you will just wish to enter a single trap; in that case you can simply type in the trap name and TMON will supply the other parameter automatically (it will be the same as the one you entered—in other words, TMON constructs a trap range consisting of one trap). If you wish to include more than one trap, type them in, separating them with a space. You will need to enter a range, even if it consists of one trap, if you need to enter one or both of the optional PC values. If you enter the traps in the range in reverse order, TMON will reverse them for you automatically (so that trap ranges always go from lowest trap number to highest).

Note that trap numbers have changed from the 64K ROMs to newer ROMs. On 64K ROMs, only trap numbers from $000-$1FF were valid. On 128K or 256K ROMs, trap numbers from $000-$FFF are valid. Also on newer ROMs, the toolbox/OS flag bit is *the* definitive way to identify a trap as an OS trap or a toolbox trap. The upshot of all of this is that you need to be conscious of what you are really saying when you type in a trap name or number. For example, the correct way to say "do this on all traps" is to enter a numerical range from $000 to $FFF. This will work the way that you expect it to.

One popular use of Trap intercept is to intercept _InitGraf so that when an application is launched TMON will gain control very early in the program's execution, since _InitGraf is executed very early in the program.

Checksum

The "Trap checksum" function is used to determine whether a particular range of memory has had any changes from one ROM trap to another. This is useful, for example, for narrowing down the section of code that you think may be trashing a variable in your program. Trap checksum is also on the "A000 trap functions" portion of the "User" window, and, like trap intercept, it takes four parameters, the last three of which are optional.

The line below the Trap checksum line allows you to define what range of addresses to perform the checksum on. The default values here are the beginning and ending addresses of the Macintosh ROM (which, of course, can never change, and so the checksum will never fail). Change these addresses in order to change the area that TMON will checksum.

Once you have defined the traps and PC range during which to checksum, and the memory range to checksum, execution of any code that contains the traps you chose and lies within the PC range that you chose will cause the memory that you chose to be checksummed. This checksum occurs before the trap is executed. If the checksum does not match the previous checksum, TMON will be entered and will display a message indicating that the checksum failed.

Leave TMON; queue events until mouse click

This function is a catch-all one that allows you to generate events which will be accepted into the application's event queue. TMON will regain control only after a mouse down event (i.e. only when you click the mouse). As the application executes its event loop, these events will eventually be acted upon. This is a good way to force an event, such as a menu selection, into an application while you are spying on it with TMON. This function is a user area function which is on the "control functions" portion of the "User" window. To use it, click on the "User" button in the button bar or hold down the ⌘ key and press the "U" key. If the first line ("Toggle pages") under the title does not say "{control functions}", click to the right of the colon on that line and press **Return** until it does. The window should look something like this:

```
☐ USER area starts at $01DBBE.  Physical size is $7600; logical size is $7600
Toggle pages {control functions}:
Look for labels between LINK/UNLK of A6 (0-6=register Ax):6
Label table ◇ (nLabels [loc]):
Label add/remove ◇ (lbl [adr [end]]):
Label file load:
Registers ◇ (0=save):
Leave TMON; queue events until mouse click:
:
Leave application (0=ExitToShell, 1=re-launch current application):
Shut down (0=re-boot, 1=unmount volumes and re-boot):
```

To use this function, click after the colon on the line that says "Leave TMON; queue events until mouse click:" and press **Return** or **Enter**. You will appear to be in the application. You can do anything that you'd like to generate events. When you click the mouse, you will be returned to TMON. When queueing events for your application, bear in mind that the Macintosh only maintains the number specified by your volume's boot blocks at boot time; any events beyond that number are lost. Once these events have been queued, the application will receive them as they are retrieved from the queue.

Trap Record

The "Trap record" function is another highly useful one for helping to determine where a problem arose. It allows you to define an area of memory which will be used to record important information about a set of traps. Trap record is located on the "A000 trap functions" portion of the "User" window, and like most of the A000 trap functions, it takes a trap number (or range) and an optional PC range. A secondary line allows you to indicate how many traps to record and whether recording should stop when the buffer is full. Optionally, this line allows you to indicate where the recorded information is to be stored. Generally it is best to leave this parameter out and let TMON allocate the space for you (TMON will allocate space in the system heap). Any time a trap within the given range occurs within the given PC range, it will be recorded in the buffer if fullStop is false, or if fullStop is true and the buffer is not full. Note that if you enter a non-zero value for the fullStop parameter, the function will enter TMON and give you a message when the trap record buffer is full. Otherwise the function will continue to record traps, overwriting the ones that were recorded before.

To disable the trap recording, just click to the right of the colon on the line that says "Record" and press **Return**. TMON will deallocate the block that contains the recorded information and return everything to normal.

Creating a buffer to record traps in assigns the address of the buffer to the V register, so that you can open a dump window anchored to the V register to see the recorded information. The information is layed out in rows starting with the most recent trap and ending with the oldest, and each line is as follows:

Byte	Description
0-1	A000 trap number
2-3	Low 16 bits of Ticks
4-7	Address of A000 trap
8-15	D0/A0 for OS traps, top eight bytes of stack for toolbox traps

Note that recording $10 traps is usually enough to be quite useful in tracking down the flow of the problem program.

Template

Template is an attempt to allow you to look at a some typical Macintosh data structures in a meaningful way. It takes an address and treats the data at that address as a record of whatever type you have chosen. (It's up to you to make sure that the data at the address you specify actually *is* what you are claiming that it is.)

The "Template" function is a user area function which can be found on the "memory functions" portion of the "User" window. It takes one parameter, the address of a data structure. If you press **Return** after the colon without entering a parameter, the function will cycle through the four data structures that it currently supports: WindowRecord, ControlRecord, TERec, and ParamBlock. Once you have chosen the appropriate data structure type, just type an expression that evaluates to the address of the data structure and press **Return** or **Enter**, and the "User" window will show the names and contents of the record's fields to the best of its ability. An example of this looks something like this:

```
☐ USER area starts at $01DBBE.  Physical size is $7600; logical size is $7600
Toggle pages (memory functions):
Block move (src dst len):
Block compare {} (adr1 adr2 len):
Fill (bgn end val [vLen]):
Find (byte aligned) {} (val [vLen [bgn [end]]]):
Template (WindowRecord @04FCA8) (addr):
:address=0FA700:rowBytes=40:bounds=FFD8 FFFE 012E 01FE:portRect=0000 0000 0123 01FC:
:visRgn=033534:clipRgn=033530:windowKind=0008:visible:hilited:goAway:
:strucRgn=03352C:contRgn=033528:updateRgn=033524:defProc=001534:dataHandle=033518:
:controlList=03350C:nextWindow=000000:refCon=00000009:title="TMON 2 Manual":
```

In this example, I have a data structure at $4FCA8 that I happen to know is a WindowRecord. (How do I know that? There's a TMON heap window that makes these things easy to find. We'll get to that later.) So, I typed in 4FCA8 after the colon and pressed **Return**, and the window above is what I got back from TMON. Among the interesting things that I can learn from this window are: the window is type 8 ("zoomDocProc," according to *Inside Macintosh*), the window was visible at the time that the WindowRecord was displayed, the window was highlighted, the window had a go away box, at least one control, and its title was "TMON 2 Manual." Other useful pieces of information are present as well. In particular, the handles to the window's various regions are given. Similar displays are generated for the other three data structures.

Stack Addresses

"Stack addresses" is a function on the "memory functions" portion of the "User" window. Its purpose is simply to take the values that it finds on the stack and attempt to "recognize" them (i.e. if they are return addresses within a piece of code that has a label associated with it, TMON will display the address as the appropriate label with the appropriate offset).

The user area starts off by conveniently providing you with a default parameter to this function of "SP." If you click to the right of this value (it is treated as a value) and press **Return** or **Enter**, TMON will look at the value that lies at that address and try to recognize it. The value will be displayed in the curly braces, and any label and offset combination that applies will be displayed in the parenthesis in the braces. Continuing to press **Enter** will continue to walk up the stack address by address. To start over, just click to the *left* of the address instead of to the right it and press **Return**. You can take advantage of the way in which pressing **Enter**

differs from pressing **Return** by clicking anywhere in the parameter and pressing **Enter** (which re-enters the whole line). Continuing to press **Enter** will cycle through the stack addresses list. Since the cursor remains to the left of the parameter when you do this, you can start over at any time by pressing **Return** instead of **Enter** (pressing **Return** at the beginning of the line clears the line). Since this is a user area function that returns a value, that value is also assigned to the V register.

Stack Crawl

The "Stack crawl" function is a lot like "Stack addresses" except that its purpose is to follow a chain of procedures or functions which have been activated but not yet completed. This is possible because in high level languages like Pascal and C (and in some 680x0 assembler programs as well) the A6 register is used in a LINK instruction at the beginning of the procedure and in an UNLK at the end. The LINK instruction creates what's known as a "stack frame," and the UNLK instruction destroys it. The Stack crawl function follows these stack frames in order to determine the return address of the procedure. It then sees if it can be identified by a label and, if so, does so.

It's harder to explain than it is to use. The function takes one parameter, the address register to assume is the stack frame pointer (in most Macintosh programs, it'll be register A6, expressed as the default parameter of RA6). If you click after the 6 and press **Return** or **Enter**, the function will determine what the return address for the currently active procedure is, display it in the curly braces, and display any matching label and offset in the parenthesis in the braces. Continuing to click to the right of the address displayed as the parameter will continue up the stack frame list until there are no more active procedures. The process can be cancelled at any time by clicking to the left of the address and pressing **Return** instead of clicking to the right of it. Note that the same **Return/Enter** shortcut that I described for "Stack addresses" also works here. The value displayed in the curly braces is also assigned to the V register.

Load Resource

This is kind of an obvious function, and also rather handy. It allows you to load into RAM a resource from any currently open resource file. It's particularly useful for loading resources of types other than CODE that do contain executable 680x0 code that you wish to debug (these types can be things like DRVR, INIT, CDEF, MDEF, WDEF, and so on).

The "Load resource" function is on the "memory functions" portion of the "User" window. It takes two parameters, the resource type and its ID. You can express the type as a four character value using single quotes ('CODE') or the type can be any valid longword expression. If the resource is successfully read from disk, the address of its handle will appear in the curly braces and also be assigned to the V register. Note that because this function expects the resource manager and disk I/O functions to work properly, it is a very dangerous function if the system has been left in an inconsistent state, either because of a problem with the application being debugged or because you've entered TMON at a time when the system is in the middle of an important operation. Always remember to hold down **Option** while pressing the **interrupt** button; precisely what this does is discussed in the **Trap Signal** section.

Leave application

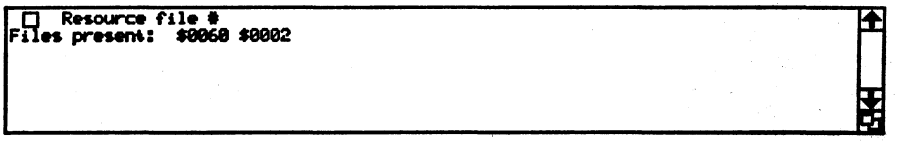
Leave application is a user area function that takes one of two parameters: zero to ExitToShell (launch the Finder) or 1 to re-launch the current application. All breakpoints are cleared by this function. Both of the choices (ExitToShell or current application) are useful for "starting over from scratch" because they close all open files and open resource files before launching. They're also useful for attempting recovery from certain kinds of system errors. For example, if the application that you're debugging consistently bombs into TMON with a System Error $1C (meaning that the stack and heap have collided) you may find it useful to try using Leave application to ExitToShell. Hopefully it will work and you will at least have some way to get out of the application which is trying so desperately to destroy the heap (exiting TMON and quitting the application won't work; by definition you'll wind up back in TMON with a System Error $1C almost immediately, unless you've played with the stack pointer to avoid the problem). You'll find Leave application on the "control functions" portion of the "User" window. Note that Leave application uses to re-launch the current application, so you can cheat and force TMON to launch anything by editing the string found in CurApName.

Shut down

Shut down, which is also among the "control functions" of the "User" window, bears a certain functional resemblance to Leave application. You use it to get out of a sticky situation. The difference is that Shut down does exactly what it says; it's like turning your computer off and back on. There are two flavors of this function: providing a parameter of 0 will re-boot; providing a parameter of 1 will unmount all currently mounted volumes and re-boot. Usually using parameter 1 will save you a lot of grief.

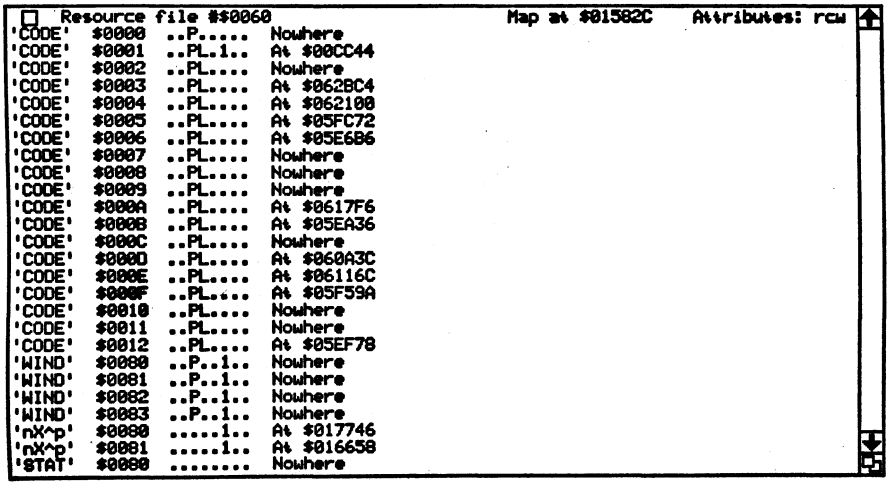
File

The "File" function is another resource-related one that's nice to have around. If you click on the "File" button in the button bar or hold down the ⌘ key and press the "F" key, a window listing all currently open resource files by file reference number will be shown. It looks something like this:



According to this window, there were two resource files open at the time: file number 2 is almost always the System file (the first resource file the Macintosh opens, by definition). In this case, as in most cases, the other file reference number refers to the application's resource file.

If we type "60" after the "Resource file #" prompt, we see something like this:



I deliberately made this window pretty large in order to show as much information as I could. This window is a (partial) resource list from the application that was running at the time.

The window provides us with some useful information. First it tells us the resource type. Then it tells us the resource ID (in hex, of course). Then it shows us the resource attributes in a format that's a lot more intelligible than just 1s and 0s. A period (".") is used if the flag is reset, otherwise a character appears. The characters are these:

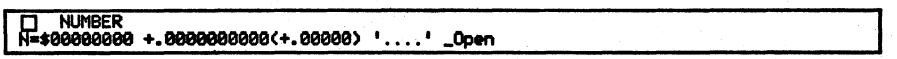
R	System reference
H	Load into system heap
P	Purgeable
L	Locked
T	Protected
1	Preloaded
W	Write into resource file
U	U flag set

Using this information, we can see from the window above that all of the CODE segments, for example, are purgeable and locked. In addition, CODE segment number 1 is preloaded (loaded into RAM immediately when the resource file is opened).

If you wish to look at a different file's resource list, just click before the file number and press **Return**. The list of file numbers will reappear. You can then choose another file number.

Number

The "Number" window is essentially TMON's answer to the programmer's calculator—you know, the ones that do decimal/binary/hex math and conversions. Well, TMON's is even more powerful than that. It will take any expression and evaluate it. Here's what the window looks like:



Looks pretty innocuous, doesn't it? It can be. You simply type any valid expression after the "NUMBER" prompt, and TMON will display the result of evaluating the expression in the line below (as a hexadecimal number assigned to the pseudoregister N, as a 32-bit decimal value, as a 16-bit decimal value, as a four character string, as whatever ROM trap the number can be taken to represent, and as a label with some offset, if the result happens to lie within a recognizable label range.

Fine. What's an expression? That's a reasonable question, and one which Waldemar has answered in great detail in his *Technical Reference*, which accompanies this one. Rather than reading a great deal of redundant information here, I suggest that you look at the Numbers section in the *Technical Reference*. You will learn just how powerful TMON's expression handling is.

Advanced Features

Ah, here we are! Now for the good stuff, the stuff that really comes in handy when the pressure's on and the bugs are starting to resemble Sherman tanks.

Trap Signal

The "Trap signal" function really could be considered a "smart interrupt" function. What it literally does is provide you with a way to press **interrupt** on the programmer's switch and not have that interrupt take effect until a single trap or trap within a range that you have specified occurs within a PC range that you have specified. This could also be considered a "basic" function, since it's used quite frequently in TMON. This function is on the "A000 trap functions" portion of the "User" window. It's been a while since we've seen that, so here it is again:

```
☐ USER area starts at $01DBBE.  Physical size is $7600; logical size is $7600
Toggle pages (A000 trap functions):
Trap record (t0 [t1 [PC0 PC1]]):
Record () (fullStop nMsg [loc]):
Trap (heap check, scramble) (zone#):
Heap () (zone#):
Trap discipline (lenient) (t0 [t1 [PC0 PC1]]):
Trap checksum (t0 [t1 [PC0 PC1]]):
Checksum (bgn end) (A426):400000 41FFFF
Trap intercept (t0 [t1 [PC0 PC1]]):
Trap signal (t0 [t1 [PC0 PC1]]):_GetNextEvent _EventAvail
```

Among the defaults that we've provided for TMON are the parameters for Trap signal. As you can see, we've set the trap range to the traps _GetNextEvent through _EventAvail (and provided no PC range, which means that any PC value will cause the function to work).

Once the parameters have been entered and TMON has been exited, you can get into TMON elegantly by holding down **Option** and pressing **interrupt**. This will indicate to TMON that you want the trap signal to go into effect. Now TMON will sort of keep an eye on things until a trap within the specified trap range and in the specified PC range occurs. If one occurs, TMON will be entered at that time—not at the time that **interrupt** was pressed. You're not likely to notice the delay, however, since things are happening so quickly. As far as you're concerned the time between your pressing **interrupt** while holding down **Option** and the Macintosh's entry into TMON is zero most of the time.

We recommend using **Option-interrupt** *any time* that you need to get into TMON, because it guarantees that the system will be in a consistent state when TMON is entered, whereas if you merely hit **interrupt**, you may interrupt the processor at a pretty bizarre time—like right smack dab in the middle of one of the system's interrupt tasks, for example. Note that this will only work if the trap(s) for which the Trap Signal function has been set are being executed somewhere after the time that you use **Option-interrupt**. If this is not the case, **Option-interrupt** will be ignored. You may use **interrupt** alone, although this is somewhat risky. If neither of those choices work, try **⌘-interrupt**, which tries rather desperately to enter TMON, even going so far as to clear some of its globals in the process. This causes TMON to display a message to the effect that it has been damaged, since TMON is constantly looking out for its own best interests by doing a checksum of itself as long as at least one window is open. Before using the Exit or GoSub functions, be sure to set up your registers and stack appropriately, since **⌘-interrupt** clears them. As a draconian measure of last resort (if, for example, **⌘-interrupt** causes TMON to hang), you may wish to use **⌘-Option-interrupt**, which does everything that **⌘-interrupt** does, and also clears the first 48 bytes of the user area so that a damaged user area won't screw TMON up. Of course, if you do this, the debugger will also complain that it has been damaged, and it will be virtually unusable for any purpose other than getting where you want to go by using Exit or setting PC to !_ExitToShell and Exiting.

Trap Discipline

Trap discipline is, in my opinion, TMON's most exciting feature. It represents a significant chunk of the user area's size and power. It is a front end to the Macintosh's ROM trap mechanism that, upon encountering a trap in the range specified within the PC range specified, checks the parameters to that trap and makes sure that they aren't obviously screwy.

Trap discipline has two strengths. I like to think of them as personal and industrial, but Darin Adler, who wrote the user area, refers to them as lenient and strict, and, I must admit, his terminology is at least consistent with the idea of discipline. To choose which strength to use, just click after the colon on the "Trap discipline" line and press **Return**. Once you have done that, you can set up Trap discipline with the same four parameters as Trap intercept or Trap signal require.

When you exit TMON with Trap discipline active, nothing obvious happens right away other than the dot in the upper left hand corner of the screen flashing on and off to let you know that TMON's up to something with the ROM traps. TMON does this whenever any of the A000 trap intercepting functions (Trap record, Heap check/scramble/purge, Trap discipline, Trap checksum, Trap intercept, or Trap signal) are in use.

If, in the process of examining the parameters to a ROM trap, TMON finds something that it doesn't think is reasonable, it will activate itself with the PC pointing to the trap whose parameters are questionable. You may then look at the stack and/or the registers to try to determine what TMON doesn't like. TMON will assist in identifying the problem as much as possible by giving you a special message window with some indicator as to what it thinks the problem is ("? NIL Handle," for example). Needless to say, Trap discipline is a real Godsend.

Look for Labels Between LINK/UNLK of Ax

This function allows you to define how local variable allocation works for the program that you are debugging. The LINK and UNLK instructions of the 680x0 family of microprocessors exist precisely to facilitate the creation of local variables. Most Macintosh programs use the LINK and UNLK instructions with address registers A6. Some Macintosh programs, however, use another address register because they use A6 for something else (in particular, programs written with the MACH 2™ 83-Standard FORTH development system use A6 as the data stack pointer and A2 as the LINK/UNLK parameter).

This function is on the "control functions" portion of the "User" window. Here's what it looks like:

```
☐ USER area starts at $01DBBE.  Physical size is $7600; logical size is $7600
Toggle pages (control functions):
Look for labels between LINK/UNLK of A6 (0-6=register Ax):6
Label table ◇ (nLabels [loc]):
Label add/remove ◇ (lbl [adr [end]]):
Label file load:
Registers ◇ (0=save):
Leave TMON; queue events until mouse click:
:
Leave application (0=ExitToShell, 1=re-launch current application):
Shut down (0=re-boot, 1=unmount volumes and re-boot):
```

To use this function, click after the colon on the line that says "Look for labels..." and enter the number of the address register to use. Note that, since so many programs use A6, "6" is the default parameter for this function.

Label Table

This function, which is also on the "control functions" portion of the "User" window, is used to allocate space to hold labels that refer to various points in a program. These labels are usually read with the "Label file load" function (explained later) but not always. Labels can be added and removed manually. (I'll explain how in a moment).

One thing that you should be aware of is that each label takes 16 bytes of RAM to store. Also, labels can be stored either as absolute or resource relative. These issues, as well as the label format, are discussed in detail in the Predefined User Area Functions section of the TMON Technical Reference Manual.

To use this function, click after the colon on the line that says "Label table" and type in the number of labels to make room for and, optionally, the location of the table (if you leave it out, TMON will allocate the space somewhere in the system heap for you). Press Return or Enter. The curly braces will contain a message indicating how many labels are currently loaded; to start with this number will, of course, be zero. The table can be deallocated by clicking to the right the colon and pressing Return without typing any parameters.

Since this user area function returns a value, that value is assigned to the V register, and you can dump the area allocated by opening a dump window and anchoring it to V.

Note that you don't need to use this function if your program contains embedded labels.

Label Add/Remove

This function allows you to add or remove a label to a label table on the fly. "Label add/remove" is on the "control functions" portion of the "User" window. Let's look at the window again:

```
☐ USER area starts at $01DBBE.  Physical size is $7600; logical size is $7600
Toggle pages (control functions):
Look for labels between LINK/UNLK of A6 (0-6=register #x):6
Label table {} (nLabels [loc]):
Label add/remove {} (lbl [adr [end]]):
Label file load:
Registers {} (0=save):
Leave TMON; queue events until mouse click:
:
Leave application (0=ExitToShell, 1=re-launch current application):
Shut down (0=re-boot, 1=unmount volumes and re-boot):
```

The label table to work on must already have been allocated. "Label add/remove" takes three parameters. If no parameters are provided the information in the curly braces is cleared. If just a label is given (it must be a label, in quotes, so that TMON knows it's a label) it will be removed from the table if it is present. Nothing will happen if the label is not present.

If a label and an address are given, the address is assigned to the label. If the "Scan resources" option in the "Options" window (more about that later) has been enabled and the address falls within a resource, the label is stored as resource-relative and needs no ending address. A message indicating that the label has been stored resource-relative will appear in the curly braces.

If all three parameters are given or the label could not be stored as resource-relative, it is stored as an absolute label starting at the address provided and ending at the ending address provided or, if no ending address was given, at the starting address + $800. Note that the ending address is considered the first byte *past* the recognition range, not the last byte *in* the recognition range. In other words, the ending address will not be recognized as "label"+offset. A message to the effect that the label has been added as absolute will appear in the curly braces.

If the label table is already full the label is not added and nothing appears in the curly braces.

One reason that you may wish to add labels manually is so that you can easily identify important sections of code. You may also wish to use it to name important locations at which to set breakpoints during debugging so that you don't have to remember breakpoint addresses.

Label File Load

The "Label file load" function is also on the "control functions" portion of the "User" window. As I mentioned earlier, some development systems create what we call ".MAP files." We call them this because the files have the string ".MAP" as the last four characters of the file name. The "Label file load" function can read a .MAP file into an allocated label table. Among the development systems that create TMON-recognizable .MAP files are Apple's MDS, Consulair's Mac C, TML Systems' MacLanguage Series Pascal (more commonly known as TML Pascal), and recent versions of Manx's Aztec C.

To use this function you must already have allocated a label table that is at least large enough to hold however many labels are in the file (and if you haven't counted them, you can always try to overestimate). Once the table has been allocated, just click after the line that says "Label file load:" and press Return or Enter. The

Macintosh's standard dialog box for reading a file will appear, and it will only show .MAP files. Open the one that you want. If TMON has trouble reading the file, or if you click the "Cancel" button instead of the "OK" button, the message "Bad load" will appear in a message window, otherwise the labels will be loaded into the table. If the table is too small to hold all of the labels, TMON will load as many as it can, and then stop. It will not provide a warning that it could not load all of the labels, so be sure that you have allocated enough space.

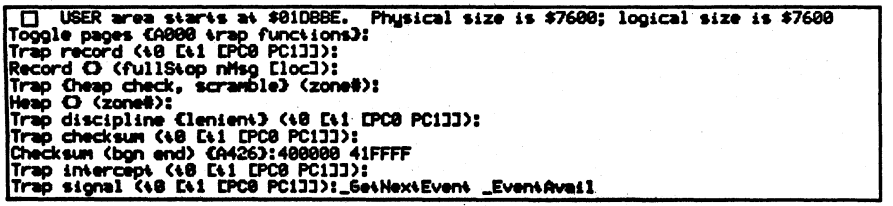
Using .MAP files is encouraged if your development system provides them, since the file will even contain labels for your development system's library routines, which is something that cannot be accomplished by embedding the labels directly in the code. Do be aware that this function is extremely dependent upon the system being in a reliable state. QuickDraw, the Dialog Manager, the File Manager, the Control Manager, the Font Manager, and TextEdit must all be working properly for this function to work properly! For this reason, you should use this function *immediately* upon entering TMON for the first time, i.e. before any other functions are used.

Heap Check, Scramble, and/or Purge

The Macintosh uses dynamic memory management. In other words, it has a pool of memory from which programs can request blocks of a particular size. These blocks may or may not move around in this memory pool (called the "heap") in order to allow allocation of large blocks. It can be quite confusing, and there are many opportunities for things to go wrong with the Macintosh's memory.

For this reason the user area includes a function that allows you to check the heap for consistency (the heap is actually a large, fairly complex data structure which may be "broken" by buggy programs), force a heap scramble anytime that one *may* occur, and/or purge all purgeable blocks from the heap.

The "Heap check, scramble, purge" function appears in the "A000 trap functions" portion of the "User" window. It looks like this:



The "Trap {heap check, scramble} (zone#):" line is the one you're looking for. If you click after the colon and press Return without typing in any parameters, the message in the curly braces will cycle through:

"heap check,"
"heap check, scramble,"
"heap check, purge," and
"heap check, scramble, purge."

Heap check examines the heap and makes sure that its structure is consistent. If it is not, TMON will be entered and a message window displayed which explains the problem. Note that with this function, a heap inconsistency is the only reason that TMON will be entered. The check is performed when any of the ROM traps listed in the next paragraph are encountered.

Heap scramble causes all relocatable blocks which can be moved to be moved. It only does so, however, when a trap that *may* cause relocation is called. The idea is to see if relocation *may* occur and, if it may, *force* it to occur. This will cause programs that rely on memory blocks *not* moving at particular times when they *may* move to fail consistently, since the blocks will consistently move. The traps that may cause relocation are _NewPtr, _NewHandle, _ReallocHandle, _SetPtrSize, and _SetHandleSize. Note that _SetPtrSize and _SetHandleSize may only cause relocation if the new size is greater than the old size. Note that the options chosen are performed before the trap is executed.

Scrambling the heap will also clear free blocks to an odd non-zero value and join consecutive free blocks. These facts go an extra step towards assuring that any program that has an invalid pointer will fail quickly.

Using the "purge" option of this function will purge all purgeable blocks from the heap before checking and scrambling. This is useful for causing programs that assume that a purgeable block will stick around to fail when it doesn't. It's interesting to note that *very* few programs can survive the purge option! Most Macintosh developers simply don't expect things to be purged out from under them, don't check, and therefore fail when they assume that something is still in RAM that has been purged.

Note that since the scramble and purge options occur whenever they *might* normally occur, they occur a lot more often than they do in "real life," and the result is that they slow the system down a great deal. If you use these, be prepared to have your system perform like someone poured molasses into its vents, and be aware that with these functions such behavior is normal.

Immediately beneath the "heap check, scramble, and/or purge" function is one that simply says "Heap {} (zone#):" If you click to the right of the colon, type a "0" for the system heap zone or a non-zero value for the application heap zone, and press **Return** or **Enter**, the curly braces will contain the total number of free bytes in the zone, the number of free contiguous bytes, and the amount that the zone can grow, in that order. If you enter a zone parameter to right of the "heap check, scramble, and/or purge" function's colon, the "Heap {} (zone#):" changes to "Heap check, etc. now" depending upon what options you chose. Clicking to the right of the colon and pressing **Return** or **Enter** will cause TMON to perform the chosen function immediately, rather than waiting until the next appropriate A000 trap. Note that scrambling and/or purging the heap when the memory manager is in an inconsistent state is extremely dangerous. You should only use this function if you have entered TMON via **Option-interrupt** or if you can somehow be sure that you are in the application environment at a moment when there are no dereferenced handles about to be manipulated. Hitting the application when it isn't looking isn't very sportsmanlike.

Heap

Speaking of heaps, TMON wouldn't be a very good Macintosh debugger if it didn't provide you with some means of taking a look at the system heap and application heap and seeing what was there. You can open a heap window by clicking on the "Heap" button in the button bar or by holding down the ⌘ key and pressing the "H" key. A heap window looks something like this:

```
☐ Application heap is at $011400-$0C0298.                    085F7C bytes free.
$019720 00011C 0  Handle at $018884 (lpr)
$019844 000012 0  Free
$01985E 000068 A  Handle at $0114A8 (lpr)  (Window @$018C22) TEHandle
$0198D8 000012 A  Handle at $018A10 (lpr)
$0198FC 000088 0  Handle at $0114EC (lpR)  File $011C 'MENU' ID=$0085
$01998C 000229 1  Handle at $011504 (lpR)  File $011C 'STR#' ID=$0080
$019BBE 00000A 0  Handle at $018840 (lpr)  (Window @$019008) ClipRgn
$019BD0 000006 0  Handle at $0114AC (lpr)  (Window @$018C22) WTitle
$019BDE 000008 0  Handle at $0187E8 (lpr)
$019BEE 00000A 0  Handle at $018890 (lpr)  (Window @$019008) ContRgn
$019C00 00000A 0  Handle at $01889C (lpr)  (Window @$019008) UpdateRgn
$019C12 000012 0  Handle at $018A1C (lpr)
$019C2C 000014 0  Handle at $0187DC (lpr)
$019C48 0000AA 0  Free
$019CFA 000012 0  Handle at $0187A4 (lpr)
$019D14 000002 6  Handle at $0187E0 (lpr)
$019D24 0000EC 0  Handle at $0114E0 (lpR)  File $011C 'MENU' ID=$0088
$019E18 00009C 0  Handle at $0114DC (lPR)  File $011C 'DITL' ID=$0081
$019EBC 00000A 0  Handle at $0188B8 (lpr)  (Window @$018AB8) ClipRgn
$019ECE 000010 0  Handle at $018818 (lpr)  (Window @$018CF8) WData
$019EE6 00000A 0  Handle at $0188B4 (lpr)  (Window @$018AB8) StrucRgn
$019EF8 000002 2  Handle at $018804 (lpr)
$019F04 00002A 0  Handle at $011510 (lpr)  (Window @$018CF8) Control
$019F36 00000A 0  Handle at $018814 (lpr)
```

This window shows the application zone. Which zone is being displayed is given on the first line of the window, along with the location of the zone (starting and ending addresses) and the total number of free bytes in the zone.

If you wish to see the information for the system zone, make sure that the cursor is on the first line of the window (by pressing the **Tab** key) and press **Return** or **Enter**. The window will change to show the system zone.

Each line of the heap window describes a block in that zone. First is a space if the block is relocatable or an asterisk ("*") if it is not. This is useful for easily seeing non-relocatable blocks in the middle of the heap, causing fragmentation. Next comes the address of the first byte of the block of data. In other words, the *pointer* to the data is next. After the pointer comes the size of the block. This will be the same as what was requested of the operating system. In other words, if the program executes a _NewHandle with a requested size of $2A bytes, a block size of $2A will appear in the heap window.

The next item—a single hex digit—takes a little explaining. It is the *size correction factor* for the block. This is the number of unused bytes in the block. If the pointer to this block is X and the pointer to the following block is Y, then the size correction value for the block pointed to by X is literally Y-X-8-Size(X), where Size(X) is the size given in the heap window.

Next comes one of four phrases: "Free," "Nonrel," "Handle at..." or "INVALID." "Free" means that the block is not currently allocated to anything. "Nonrel" means that the block is non-relocatable. "Handle at..." means that the block is normally relocatable and is referred to by a handle (a pointer to the pointer) which is at the address given. Following the address of the handle is a group of memory manager flags, L, P, and R. These flags are uppercase if set and lowercase if not. "L" stands for Locked, "P" stands for Purgeable, and R stands for Resource. Finally, "INVALID" means that the block failed the consistency check for some reason and that the zone is probably in *big* trouble.

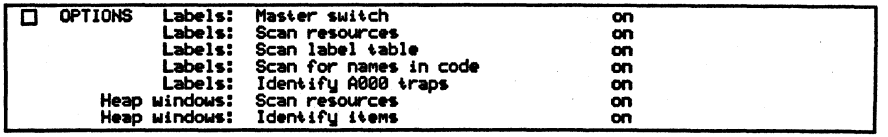
Both nonrelocatable and relocatable blocks may present additional information helpful in determining what they are. First of all, if the "R" flag is set, the handle is checked against the list of open resource files and their resources. If a match is found, the file reference number, resource type, and resource ID for the resource are shown. All other nonrelocatable and relocatable blocks are passed to a routine in the user area (the customizable part of the debugger). This routine should try to identify as many blocks as it can. In the examples above, the routine has identified many parts of windows (controls, TEHandles, visRgns, etc.). The system zone display identifies things like the FCBs list (File Control Blocks), the WDCBs list (Working Directory Control Blocks), DCEs (Device Control Entries), the UnitTable, and so forth. All of these are handled by a customizable user area routine, so if there's something that you'd like identified that isn't currently, feel free to rewrite the routine and add it!

You can set the V register to the address of a heap block by positioning the cursor somewhere on the line for that block and pressing **Return**. This is useful, for example, for entering short code sequences for quick and dirty patches: find a block marked FREE, put the cursor on that line, and press **Return**. Now open an "Asmbly" window anchored to V. You can enter a few bytes of code in this unused block that way.

Options

The "Options" window allows you to specify which of TMON's recognition features are used and which ones aren't. This is useful for a couple of reasons. First, many of these features are more reliant upon the system being in a consistent state than the heart of TMON is, and they may break and cause problems if the operating system is in an inconsistent state. Secondly, some of these features are somewhat slow, and if you don't need them you can turn them off and speed TMON up significantly.

Here's what the "Options" window looks like:



There are five switches for label functions and two for heap functions. The first label switch is the master switch; it defines whether or not labels are used at all. I find myself turning this one off whenever I am in a tight loop and have no convenient way to get out of it (i.e. it's in ROM or something like that). The labels

probably aren't doing me any good as long as I'm in this DBRA or whatever, so I can turn them off and let TMON move a bit faster (I usually close all windows except the "Regs" window, too, for maximum speed).

The second switch defines whether TMON will recognize addresses as being within a resource. In other words, if this switch is off, disassemblies will never indicate whether the code lies within a CODE resource or DRVR resource or what have you. Since TMON has no way of identifying code within resources, embedded labels are disabled by this option by definition (since TMON only looks for embedded labels if the code falls within certain resources).

The third switch defines whether or not TMON will look for code labels in a label table. If this switch is off, any label table that has been allocated will be ignored. If this switch is on, the table will be scanned for labels. Note that if the "scan for resources" switch is off, any resource-relative labels in the table will be unrecognizable, since resources won't be recognized.

The fourth switch defines whether TMON looks for labels embedded in the code for a program. If this switch is off, embedded labels will be ignored. Again, if resource recognition is off, then this option will be off by definition as well.

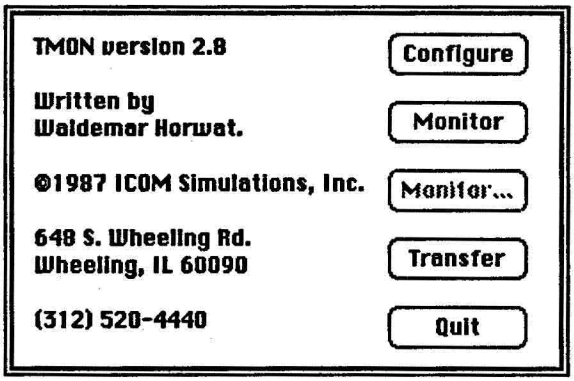
The fifth switch defines whether or not TMON will label code that belongs to an A000 trap. Normally TMON labels disassemblies of A000 trap code with !_TrapName+Offset, where "TrapName" is the name of the OS or Toolbox trap, and offset is the number of bytes away from the beginning address of the code. Turning this switch off will cause TMON not to label such code.

The sixth switch defines whether or not the heap window will identify blocks which are resources. You may wish to turn this capability off if, for example, a bug has corrupted a file's resource map, causing the function to fail.

The last switch defines whether or not TMON will attempt to identify heap blocks. Again, you may wish to turn off this switch in order to keep this function from failing in the event of some heap-related catastrophe, or just in order to improve debugger performance (assuming that, for the duration, you won't miss the feature).

The Main Dialog

The Main Dialog appears like this when there is a Monitor in memory. (The User's Guide discusses how it appears when there is no Monitor in memory.) This is the menu you will get if you double-click TMON or a User Area when there already is a Monitor in memory.



At this point you have a choice of five options. They may be executed by pressing the "C" key for **Configure**, the "M" key for **Monitor**, the "." key for **Monitor...**, the "T" key for **Transfer**, or the "Q" key for **Quit**. **Monitor...** is disabled if the Monitor is installed in memory, and **Configure** is disabled if there is no Monitor installed in memory. Most of these functions are discussed in the *User's Guide*. What follows is some additional information about certain functions.

Loading the Monitor

When there is no Monitor in memory, **Monitor** will load and enter the Monitor using either the file called "User Area" on the disk, or, if there is none, the built-in user area in TMON with the Monitor. More information on user areas can be found throughout this manual.

Monitor... is the same as **Monitor** except that it allows a user area other than the default one to be selected and used with the Monitor. Clicking **Open** will start the Monitor, while clicking **Cancel** will return to the Main Dialog. Hereafter these two options will be described together.

If a Monitor is already present in memory, the **Monitor** button will re-initialize the Monitor and reenter it without reloading it from the disk. Use the **Interrupt** button to reenter the Monitor without re-initializing it.

Do not use this button to re-initialize the Monitor if breakpoints have been set because the Monitor will forget about them even though they remain set. You will then be unable to clear the breakpoints.

Here is some terminology clarification: "TMON" is the program that loads the "Monitor" into memory. The Monitor, once loaded, is a debugger that remains in memory and spies on whatever program is currently running. You can easily tell when the Monitor is active because it has its own special screen format. The icons that you see on the disk if you examine it with the Finder belong to TMON. TMON, in addition to its own code, contains images of the Monitor and the "built-in" user area (see below).

Loading a User Area

The concept of a "user area" is used throughout this manual. A "user area" is a set of routines which are used to customize the Monitor to one's needs. It also contains several useful **Configuration** options and the current state of the Monitor's windows. In addition, there is a built-in user area embedded in TMON itself. The procedure for selecting the user area to be used with the Monitor is as follows:

1. If you double-click TMON or execute it as the startup application, the file "User Area" on the disk is used as the default user area. If there is no such file, the user area within TMON is used. That area cannot be removed or changed, and cannot be accessed unless there is *no* file called "User Area" on the disk.

   If you double-click a User Area, that user area will be used instead.
2. If you select **Monitor** in the Main Dialog while there is no Monitor in memory, the procedure is the same as in 1.
3. If you select **Monitor...** in the Main Dialog, you are asked to choose a user area. Once you do so, that user area is loaded with the Monitor.

User areas built with older versions of TMON (2.585) can not be loaded. An attempt to load an old user area will yield the error message, "The user area is too old." See Creating Your Own User Functions for details on modifying an old user area so it will work with this TMON.

The Monitor

When the Monitor starts, you will see a list of commands in a small bar at the top of the screen and a welcome message near the top of the screen. Now you can use any of the Monitor's features.

The Button Bar

There are fourteen commands in the button bar. To select a command, position the mouse over one of the commands and click the mouse button. It is also possible to use the keyboard equivalent of any of the commands in the button bar. For that hold down the ⌘ key and type the first letter of the command.

☞ Note that the button bar at the top of the screen behaves differently than the standard Macintosh menus; it is more similar to a collection of buttons. If you change your mind and don't want to select any function in the button bar, move the mouse below the button bar and then release it.

☞ One command does not appear in the button bar. It is the `Mouse Unfreeze` command. The only way to invoke it is to type ⌘M.

Windows

The commands `Dump`, `Assembly`, `Breakpoints`, `Registers`, `File`, `Heap`, `Number`, `Options`, and `User` use windows to do their functions. Executing one of these commands, either by clicking in the button bar or typing the keyboard equivalent, causes a corresponding window to appear or be uncovered. You can create only one each of `Breakpoints`, `Registers`, `Options`, and `User` windows; any number of `Dump`, `Assembly`, `File`, `Heap`, and `Number` windows can appear simultaneously on the screen. To create additional windows of one of these types you can't just use the corresponding command, because that will just move the cursor to the top of the window. Instead, you have to hold down the Shift key while either clicking in the button bar or typing the keyboard equivalent.

☞ Use the Shift key to create additional windows of the same type.

The windows behave similarly to normal Macintosh windows, although there are differences which will be explained here. All windows occupy the full width of the screen and therefore can't be moved horizontally. To close a window, click in the close box. To move it, press the mouse button down while the mouse is anywhere in the window (except in the close, resize, and scroll boxes), and drag the mouse up or down. `Dump`, `Assembly`, `File`, and `Heap` windows also have resize and scroll boxes on their right sides. Dragging the resize box in the lower right corner of the window will make the window bigger or smaller. Clicking one of the two scroll arrows will scroll the window one line, while holding down the mouse button there will repeatedly scroll the window. Note that the windows can be resized or scrolled even if they are partially covered by other windows, without being brought to the front. These windows can also be moved without being brought to the front by dragging them in the vertical area between the two scroll boxes. No more than nineteen windows can be open at any time; if you have exactly nineteen windows open, the last one may disappear if you save the user area, restart the Macintosh, and reload the Monitor with that user area.

☞ Windows can be dragged by pressing and dragging the mouse almost anywhere in the window, and not just in the title bar as with normal Macintosh windows.

☞ Resizing or scrolling a window does not automatically bring it to the front. Dragging a window also will not bring it to the front if the mouse is positioned in the area between the two scroll arrows.

The "active window" is a synonym for the frontmost window. It is a little more difficult to find which window is active in the Monitor than in the normal Macintosh windows. The active window, if there is one, is always the frontmost one and contains a flashing cursor. There is no active window if there is no window at all on the screen or if there is a special message displayed near the top of the screen. (An example of a special message is the welcoming message that appeared when you first entered the Monitor. Special messages can be distinguished by the fact that they have no close boxes and disappear on the first keystroke or mouse button.)

Refreshing of Windows

All windows on the screen are continuously being refreshed. This provides you with a unique real-time view of memory contents. The refreshing is faster if there are fewer and smaller windows on the screen. To see an example of refreshing, open a `Dump` window for locations $150-$180 or $800-$900. Also try opening an `Assembly` window starting at location $820.

☞ The line currently being edited (the line containing the cursor) is *never* refreshed. This is done to prevent the line from changing while it is edited.

➪ Refreshing can sometimes become extremely slow due to the long time needed to display the information in some windows. This is especially noticeable when there is a large `Assembly` window showing a portion of ROM on the screen. The villain in this case is the label routine, as it takes more than 70% of the time used to refresh the screen. See the **Labels** section for more information about this.

The Cursor and the Editing Facilities

The cursor is the flashing bar on the screen. It appears inside the active window. You can move it to a different place by clicking the mouse over the new place. If you experiment with this a little, you will find that some windows will not let you put the cursor in some positions because these positions do not contain anything that could be changed. Everything that you type appears at the cursor's position. If your keyboard has them, you may also use the left and right arrows to move the cursor left and right on the same line.

To enter text (more on that later) just type it after positioning the cursor to the correct place. The **Backspace** (or **Delete**) key may be used to delete characters. **Return** and **Enter** are used to enter the data you have typed. **Enter** enters the entire line, including any data you may have typed to the right of the current cursor position. **Return**, on the other hand, only enters the line, starting from the leftmost position and up to the cursor. **Clear** on the numeric keypad may be used to erase the entire line without entering it. More information about whether to use **Return** or **Enter** appears in the individual command description sections which follow. You don't have to follow these rules if you don't want to; every place where one of these two keys may be used, the other may be used as well.

If you decide that you don't want to enter the text, move the mouse to some other line and click it there. This will undo all changes you may have made on the original line.

The **Tab** key may be used to move the cursor to the top left corner of the window. It is useful for entering addresses in `Dump` and `Assembly` windows, entering the Program Counter value in a `Register` window, etc.

The button bar will flash after you have pressed **Return** or **Enter** if you made a mistake somewhere on the line.

☞ All text that is not between single quotes (') is converted to upper case; therefore, you may type anything except ASCII strings in either case without affecting the outcome. One exception to this rule is mentioned in the section on `Dump` windows.

Numbers

Whenever you are asked to enter a number, you may enter a number in either hexadecimal, decimal, binary, or ASCII, the value of a register, an indirection, an A000 trap name, a label, or any expression containing the above items. The default base is hexadecimal, but hexadecimal numbers may optionally be preceded by a dollar sign. Since spaces are often used to separate values, they may not be embedded in expressions except in ASCII values and labels. Decimal numbers are preceded with a period, and binary numbers are preceded with a percent sign. ASCII values must be enclosed by *single* quotes, and may contain single quotes themselves provided that the inside single quotes are doubled. (Example: To enter the string `a'b`, type `'a''b'`.) Labels must be enclosed by *double* quotes, and must not contain double quotes themselves. Labels may be any length, but only the first eight characters are significant. See the **Labels** section for more information about labels.

A000 trap names are entered by typing an underscore (_) followed by the name of the trap. The value generated by doing this will be the number of the trap as if the trap were entered into an `Assembly` window with one exception: the `Assembly` window allows the value of bits 8 through 11 to be set by following the trap name with a number (or expression), while trap names in expressions do not allow that. Since they are in expressions, however, the same effect may be achieved by adding the corresponding value to the trap name (Example: `_Open+$200`).

☞ There are differences between the usage of the A000 trap names in expressions and as `Assembly` window opcodes. The trap names may also be used in expressions in `Assembly` windows; they are not considered opcodes. An example of an A000 trap name in an expression in an `Assembly` window is `MOVE #_Open,D0`.

The registers may be used in two ways. They can either be used to provide values in expressions or to be interpreted as actual registers. The second option will be called passing registers as variables. Consider this example to clarify this distinction: Suppose you type the `MOVE A0,USP` instruction into an assembly window. Also suppose that the current value of the user stack pointer (as displayed in the `Registers` window) is $12345678. If the assembler interprets the `USP` as a value, the instruction will actually be assembled as `MOVE A0,$12345678`. If, on the other hand, the `USP` is interpreted as a variable, the instruction will be recognized as `MOVE A0,USP`. In this particular example the assembler would choose the latter option. The exact rules on whether register values or variables are used are described below.

☞ Make sure that you understand the difference between value and variable references.

Some registers have different names depending on whether they are used as values or variables. If that is the case, there is no ambiguity. If the same name is used for both the value and the variable, the variable will be selected whenever possible. Only if that is not possible will the value be selected. For example, in the instruction `MOVE A0,USP` the `USP` is interpreted as a variable. On the other hand, in `MOVE D0,USP` the `USP` is a value since it cannot be a variable (There is no `MOVE` instruction from a data register to the user stack pointer). In most expressions references by variable are prohibited so the values will be used.

☞ The only places registers may be referenced by variable are in `Assembly` windows and anchoring.

These names are used for the registers:

Variable	Value	Register name
A0 to A7	RA0 to RA7	Address registers.
D0 to D7	RD0 to RD7	Data registers.
SP*	SP	Same as A7 or RA7. *For anchoring windows only.
SSP*	SSP	System stack pointer. *For anchoring windows only.
USP	USP	User stack pointer. (Normally unused in the Macintosh)
PC	PC	Program counter.
SR		Status register.
CCR		Condition code register.
N	N	The result of the last `Number` calculation.
V	V	Result of `Find`, `Heap`, and other functions.
USER		The beginning of the user area.
DSPT		The address of the ROM A000 trap dispatcher.

Expressions can be made by combining numbers and register values using these binary operators:

+	Addition
-	Subtraction
*	Multiplication
/	Signed division
\	Signed modulo (The result has the same sign as the quotient would have.)
|	Logical OR
^	Logical exclusive OR
&	Logical AND

The following unary operators are also allowed:

~	Logical NOT
@	Indirection (The four-byte value at the given memory location. The given memory location must be even; if it isn't, the button bar will flash.)
+	Positive number
-	Negative number
!	Given an A000 trap number, return that trap's address

Consecutive unary operators are evaluated from right to left. Binary operators are evaluated according to this order of precedence:

*/\	Evaluated first
&	
+-	
^	
|	Evaluated last

Consecutive binary operators with the same order of precedence are evaluated from left to right. Triangular brackets (< and >) may be used to modify the order of evaluation. Depending on the complexity of the expression, between five and ten levels of parentheses are allowed. All operators use 32-bit arithmetic.

Here are a few sample expressions:

`0`	Evaluates as 0.
`A0`	Evaluates as $A0 unless it is interpreted as a variable reference to address register 0 by the assembler.
`$A0`	Always evaluates as $A0.
`2+2`	Evaluates as 4.
`-1*-.10`	Evaluates as 10 ($A). Both minus signs here are unary. The period indicates that the second number is decimal.
`RA2+RD3*2`	The value of address register 2 plus twice the value of data register 3.
`22/2|2+F&'A'`	Same as <22/2>|<2+<F&'A'>>, which is $13.
`_Open`	Evaluates as $A000.
`_Read+$400`	Evaluates as $A402.
`_Alert`	Evaluates as $A985.
`_NewHandle`	Evaluates as $A122.
`!_Open`	Gives the address of the ROM `_Open` routine.
`DSPT+16`	Gives the address of the 22nd byte of the ROM trap dispatcher.
`"WDef0000"+30`	Gives the address of the 48th byte of the `WDEF 0` resource. See the next section for details on labels.

Labels

Labels make the Monitor a symbolic debugger. They allow you to work using names instead of often meaningless or arbitrary numbers. The Monitor's label facility is powerful, but, most of all, it is flexible. If you have code labels that are not recognized by the Monitor, and if you have the necessary expertise, you may write a user area function that will recognize those labels. If you are not an expert, ask someone else to do it.

There are two basic operations that can be done with labels: convert a label into an address ("evaluate a label") and convert an address into a label plus an offset ("recognize the address"). The first operation is done whenever you use a label in any expression; the label is automatically converted into an address, and the expression is evaluated further. The second operation is done in `Assembly`, `Number`, and possibly `User` windows. `Assembly` windows display the label and offset corresponding to the current instruction address on the left side of the screen and the labels and offsets indicating any effective addresses on the right side. `Number` windows suppose that the value typed was an address and try to find the corresponding label and offset, which are displayed in the bottom right corner of the window if they were found. `User` functions may also identify addresses using labels.

☞ *Evaluating a label* is converting it into an address. *Recognizing* an address is converting it into a label plus an offset.

☞ Each label is assigned an address and a *recognition range*. The recognition range is the range of memory, beginning with the label's address, any references to which shall be recognized as the given label plus an offset.

⇨ In general, no recognition range should be greater than $FFFF. Also, most of the label routines dealing with resources will either ignore or truncate resources which are longer than $FFFF bytes.

Labels are always displayed as eight characters, possibly padded on the right side with spaces if there are fewer than eight characters. They are displayed exactly the way they appear in the label table or code, which means that they are not converted to upper case for the purpose of displaying. Labels are entered into expressions by enclosing them in double quotes ("). They must not contain double quotes themselves. More than eight characters may be typed for a label, but all but the first eight are ignored. If fewer than eight are typed, the remaining characters are set to spaces. Unlike ASCII constants, labels typed into the Monitor *are* converted to upper case. The Monitor ignores case while searching for labels.

☞ Type labels by enclosing them in double quotes. Although labels are *displayed* in their original case, case is not important when comparing labels, and you may type labels in either lower or upper case. You may enter more than eight characters, but only the first eight are significant.

There are three basic kinds of labels plus two kinds of pseudo-labels. The labels may be either *embedded name labels*, *resource/ID labels*, or *table labels*. Table labels may further be subdivided into *absolute labels* and *resource-relative* labels. Here are the explanations of these types of labels:

Embedded name labels use names placed in code resources to identify code routines. The names are placed there by some compilers and may also be included in assembly language routines by using the `DC.B` (or equivalent) assembler command. The specific method of recognizing embedded name labels is left to the user area. The default embedded name searching user routine searches for these labels in CODE, CACH, CDEF, DRVR, DSAT, FKEY, FMTR, INIT, LDEF, MDEF, NBPC, PACK, PDEF, PROC, PATC, PTCH, SERD, WDEF resources. (This can easily be changed in the user area.) A routine to which an embedded name label is assigned must begin with a `LINK A6,#____` instruction and end with either `RTS` or `JMP (A0)`. An `UNLK A6` must be present within 20 bytes before the `RTS` or `JMP (A0)`. The name of the routine must be placed immediately after the `RTS` or `JMP (A0)`, and must be at least eight characters long. Moreover, it must consist of eight valid ASCII characters (ASCII values $20 to $7E), but the first byte may optionally have its 7th bit set. Finally, because of performance considerations there is a limit on the length of the routine: it should not be longer than about 4000 bytes. If at least one of the above conditions is not satisfied, the label is not recognized. If recognized, the label is set to the address of the `LINK` instruction, and the recognition range extends to the `RTS` or `JMP (A0)`.

Note that the routine may have internal `RTS` or `JMP (A0)` instructions as long as there is no `UNLK A6` preceding them within 20 bytes. There can be no internal `LINK A6,#____` instructions, since any such instruction would be used as the beginning of the routine.

☞ If you are using a compiler to generate the embedded name labels, make sure that the necessary compiler options are turned on.

☞ Sometimes it is possible for the embedded name label routine to recognize spurious labels. This may happen if there are ASCII characters present after the end of a subroutine.

Resource/ID labels are a convenient way of identifying resources. A resource/ID label consists of the resource type in the first four characters and its ID in hexadecimal in the last four characters. Anything belonging to any resource is recognized using a resource/ID label containing the correct resource type and ID. For example, if you typed `"DITL0080"+34` on the top line of a `Dump` window, the address of the window would be set to the $34th byte of a DITL resource with an ID of $80. If you open `Assembly` windows to code segments with no other labels in them, they will be most likely identified by using resource/ID labels.

There is one rare circumstance when this function might not behave as expected. The Monitor converts the label to upper case, and the label evaluate routine then compares the resource type against the first four characters of the label. This works fine except when the resource type contains lower case characters. Such resources cannot be specified by using labels in expressions. Recognition works fine for all resources, though.

☞ Using resource/ID labels is a quick way to open windows to specific resources, as long as the resources are present in memory. If not, use the user area `Load resource` function. Also use `Load resource` to find resources with types containing lower case letters.

Table labels are the labels you may add or modify. You may also load them from .MAP files via a user area function (See **Label File Load**). You may add or remove table labels by using the `Label Add/Remove` user area function. Before you can do anything with table labels, however, you must allocate space for a label table by using the `Label Table` user function. As you can see, table labels are quite dependent on the user area, which also means that they can be changed in many aspects. The information given here applies to the predefined user area only.

Table labels may be either absolute or resource-relative. Absolute labels refer to an absolute memory location, while resource-relative labels refer to memory locations within certain resource blocks such as code segments. Absolute labels are most useful for identifying low-memory variables and ROM addresses. Resource-relative labels are used as an alternate way of identifying code routines, with the advantage that names do not have to be entered into the code and the disadvantage that the label table has to be allocated and loaded.

Absolute labels also have an ending address to prevent cases such as addresses around $7F000 being identified using a label pointing to $400. The ending address specifies the end of the recognition range (it *is* included in the recognition range). Resource-relative labels do not have explicit ending addresses, but their recognition ranges end at the ends of their resources. If an address could be identified using more than one table label, the one higher in memory is used (for recognition purposes, of course. Obviously, either one may be used for evaluation. An example should make it clear: suppose that there are two labels, `ALPHA` at $20 to $300 and `BETA` at $40 to $274. Address $10 would not be recognized at all, and neither would $500. $3F would be recognized as `"ALPHA "+$001F`, but $40 would be `"BETA "+$0000`. Finally, $274, $275 and $300 would be recognized as `"BETA "+$0234`, `"ALPHA "+$255`, and `"ALPHA "+$2E0`, respectively. On the other hand, nothing prevents you from *typing* `"ALPHA"+254` to specify $274).

☞ In recognizing labels, whenever there is a conflict between two or more *table* labels, the one highest in memory will be used. The results of a tie are unspecified.

Here is some more technical data on the storage format of both kinds of table labels. It is also helpful in understanding exactly what these labels can do. Each label record contains 16 bytes to make it easily readable in a Dump window. The bytes are assigned as follows:

Byte	0	1	2	3	4	5	6	7	8 to $F
Absolute	0	-begin address-			0	-end address-			label name
Resource-relative	---resource type---				--ID--		-offset-		label name

The user area has a built-in table of labels describing the Macintosh low-memory system globals. They are listed in **Appendix A**. Both the user area table and the user-defined table, if any, are scanned for labels.

The user area also recognizes labels for addresses in the A000 dispatch table from the A000 trap names it knows. The label of the four-byte address (two-byte if 64K ROMs are in use) in the dispatch table pointing to the code of trap _Open is labeled "jOpen", etc. If a trap has no name, it is recognized as "j$Axxx", where $Axxx is the trap's number in hexadecimal.

➪ These labels are *not* evaluated in expressions by the built-in user area.

Finally, there exist pseudo-labels which are not really labels but objects recognized by the address recognize routine. They are the A000 trap entry points and the `DSPT` variable. These items are recognized as if they were labels, but do not contain any quotes. To evaluate them, type `!_`*routine name* to find the address of the A000 trap routine and `DSPT` to find the address of the A000 trap dispatcher. Only ROM locations and system heap nonrelocatable blocks are searched while recognizing these pseudo-labels. To prevent excessive misidentification only the first $800 bytes after an A000 trap entry point are attributed to that label. Just like in table labels, the A000 trap whose entry point is highest in memory but not above the given address is attributed to that address.

One more issue remains to be resolved: what happens when several of the above types of labels may be used to *recognize* a particular address. The label highest on the order of recognition is given. The order of recognition is:

1. Table labels
2. Embedded name labels
3. Resource/ID labels
4. Pseudo-labels

Although this happens less frequently, it is possible for a given label to be *evaluated* in more than one way, for example, when there is a routine named `"CODE0001"`. In this case the order of recognition is again followed, which means that in the example the routine would be given and not the CODE resource. If there are several embedded name routines with the same name, one of them will be picked, but there are no guarantees as to which one. Under normal operation there cannot be two or more table labels with the same name, but that situation could arise if the label table is altered directly. Again, no guarantees will be made about which label will be used.

Since the labels depend on much of the system being in a consistent state, they may not always be reliable (although special precautions have been taken to avoid following things like odd or NIL pointers). In other cases it may be preferable to turn off one or more of the label types. Finally, some of the label recognition routines may not be particularly fast. Large `Assembly` windows with slow label routines tend to severely degrade the Monitor's performance. Try opening a large `Assembly` window to ROM and see how slowly the Monitor responds to typing and mouse clicking. On the bright side, the slowest label recognition routine, the pseudo-label recognition routine, is called only when disassembling ROM or system heap blocks. `Assembly` windows pointing to places like CODE blocks run at a normal speed unless you have several hundred labels defined in the label table.

Incidentally, you can time the relative speeds of having various windows open on the screen by opening a small `Dump` window pointing to Ticks ($16A). Make sure that the cursor is *not* on the second line of the `Dump` window, and observe the changes in the value of $16D. Open other windows elsewhere on the screen and see how the increments in $16D get less frequent but also greater. Open a large `Assembly` window pointing to ROM, and you can get an idea how much it slows the Monitor.

For the reasons outlined in the above paragraphs you have the option of turning off parts of or the entire label recognition system in the `Options` window. See the description of that window for more information.

☞ Use the `Options` window to disable the label system. Do it to prevent crashes when opening `Assembly` or `Number` windows when the system is in an inconsistent state, speed up the Monitor's pace, or if you simply do not wish to use labels.

Exiting the Monitor

When you want to leave the Monitor and either transfer to an application, go into **Configuration**, or exit to the Finder, use the `Exit` function, either by typing ⌘E or clicking in the `Exit` area in the button bar. The Main Dialog will reappear. The Monitor will stay in memory and can be called by **interrupt**.

Reentering TMON

Even after you have exited to the Finder or transferred to an application, you may re-execute TMON either by clicking on its or any user area's icon, although if you use a user area, it will be ignored. Doing this will display the Main Dialog, from where you may either re-initialize the Monitor, enter **Configuration** and save the user area, transfer to an application, or simply quit again.

Permanently Leaving the Monitor

The Monitor continues residing in memory and can be called by pressing **interrupt** until you either turn off the Macintosh or press **reset**. There is no other simple way to dispose of the Monitor.

The Monitor's Functions

Here is a discussion of some of the specifics of each of the Monitor's functions, most of which are accessible from the button bar.

Dump

The top line of the Dump window serves to allow you to enter the beginning address of the window. When the window is first opened, the address is set to zero. You can set the address by typing a number or an expression there and pressing **Return** or **Enter**. The address is automatically aligned to a word boundary. If the cursor is already in a Dump window, a quick way to get to its top is to press **Tab**.

If you type **Return** on the top line without entering an address, the previous address of the Dump window is incremented by two, and the window is updated. This is useful for fine adjustments of the window.

It is also possible to "anchor" Dump and Assembly windows to particular 68000 or Monitor registers. It is done by entering the name (variable reference name) of the register in parenthesis on the top line of the window, optionally *preceded* by an offset. All data and address registers may be used, as may SP, SSP, USP, PC, N, and V. The V register is particularly useful in anchoring windows during searches. Examples of anchoring are: (A0), (D5), -20(PC), 8(V), and RA0(A0). In the last example the RA0 was a value reference, while the A0 was a register variable reference.

On the left side of the screen are the addresses of the sixteen bytes displayed on each line of the window. After the colons some symbols may be present; they are:

P	The address of the line is the same as the program counter.
S	The address of the line is the same as the system stack pointer.
U	The address of the line is the same as the user stack pointer.
0 to 6	The address of the line is the same as the value of that address register.
*	One of the breakpoints is set at the address of the line.
N	The address of the line is the same as the value of the last Number calculation.
V	The address of the line is the same as the value of V.

No more than two symbols are displayed per line, even if more than two are applicable.

To enter data into memory, move the cursor into the data section, and change the numbers there. Of course you may also enter expressions, but all values have to be between $00 and $FF. When you are done changing your line, press **Return**. The cursor will then move to the first unchanged byte. Before you press **Return**, however, make sure that the cursor is still in the hexadecimal part of the dump.

You may also enter ASCII data by changing the ASCII part of the dump. Press **Return** or **Enter** to accept the new data, but make sure that before doing that the cursor is in the ASCII part of the dump. The cursor will then move to the ASCII value of the first unchanged byte.

☞ Do not enter the ASCII text in the ASCII portion of the window in single quotes. The text in ASCII portions of Dump windows is not converted to upper case even though it is not between quotes.

In ASCII portions of Dump windows *only*, in order to avoid entering extra spaces at the end of the line, **Return** is made to behave like **Enter** in that it does not clear the rest of the line.

Enter may also be used to accept the new hexadecimal values. Be careful, however, because it will also accept the values left in the remainder of the field as if you typed these values yourself. This may cause problems like this one:

```
Before Changes   00 11 22 33  44 55 66 77 88 99 AA BB  CC DD EE FF   ..”3DUfw........
Changes          00 11 .17 .18 .19 2+2 'x'88 99 AA BB  CC DD EE FF   ..”3DUfw........
After Changes    00 11 11 12  13 04 78 88 99 AA BB CC  DD EE FF FF   ......x.........
```

The best way to learn to enter data into a Dump window is to experiment, preferably on a block of unused memory. Locations between the top of the application heap (which may be found by looking at the top line of an application Heap window) and the bottom of the stack (which is pointed by register A7 in the Registers

window) are usually unused. In addition, it is sometimes convenient to locate a free block in the heap (using the `Heap` window), and experiment there.

`Dump` windows may be scrolled and resized as described in the **Windows** section.

Do not enter addresses above ROM in the top line of the `Dump` window unless you are sure you know what you are doing. If you know the addresses and functions of the I/O devices, you may sometimes be able to change their settings directly from a `Dump` window. Remember that since the screen is constantly being refreshed, the addresses shown in the `Dump` window are constantly being read. This prevents the displaying of SCC registers, as even *reading* the wrong address will reinitialize the SCC, freezing the mouse (See **Mouse Unfreeze**). VIA registers, however, may be displayed and even changed. Make sure that the byte you want to change is the first in the `Dump` window's line. Type a new hexadecimal value (or expression) over the old value and press **Return** (not Enter!). This way only the first byte of the line will be written. If you pressed **Enter**, all bytes on the line would be written. If you modified some byte other than the first and pressed **Return**, all bytes up to the one after which you pressed **Return** would be written. Both of these cases may cause some very undesirable effects.

On a 68020 machine, if the Monitor gets a bus error while reading memory to display in a dump window (or assembly or any other window), it will substitute a random number instead. This will cause a dump window pointing at nonexistent space to constantly change. If you have a Macintosh II, you can see this effect by opening a dump window to address 0 and then scrolling it backwards one line so it starts at address $FFFFF0.

Assembly

`Assembly` windows behave very similarly to `Dump` windows except that, of course, they contain assembly language data. The address of the window on the top line is set in the same way as in `Dump` windows. The symbols after the colon after the address of each line are also the same as in `Dump` windows.

As in a `Dump` window, typing **Return** on the top line without entering an address increments the previous address by two. Use this if the window appears to be misaligned in the instruction stream.

The `Assembly` windows attempts to use labels to recognize the address of the instruction and any effective address that the instruction may contain. The recognition of the instruction address is displayed just before the instruction itself, and may not be edited. If the instruction contains an absolute, relative with offset, or relative with index and offset addressing mode, the address specified by that mode is recognized on the right side of the screen in the form of a comment. If there is more than one such addressing mode present, only the *last* one is recognized. The recognition information will not be displayed if there is not enough room on the line.

Sometimes you may want to type an instruction on the line but cannot fit it due to the label on the left side of the screen. In that case, use the `Options` window to temporarily turn off labels, assemble your instruction, and turn the labels on again. Incidentally, the instruction that causes the longest disassembly is: `MOVEM.L ^_____(D0.L),D0/D1/D3/D4/D6/D7/A0/A1/A3/A4/A6/A7`. (Note the missing registers in the register list. If `D2`, `D5`, `A2`, or `A5` were present, the disassembler would use register ranges.)

Recognition of labels can be very convenient, but it may also slow the window. See the **Labels** section for information on the usage of labels and fixes for some problems.

To assemble instructions, type or change the instructions listed and press **Return** or **Enter**. If you entered an entirely new instruction, you will probably want to use **Return**. If you just changed an immediate value or an address in an existing instruction, it will probably be more advantageous to use **Enter**. Again, if you are unsure about the consequences, experiment on an unused area of memory.

No spaces are necessary before the mnemonics, but their presence there will not harm anything. At least one space is necessary between the mnemonic and the operands, if there are any. Spaces are *not* allowed anywhere else except at the end of the line and in ASCII constants and labels. The line may also have an optional comment at the end of the line. The comment must begin with a semicolon (;). The primary reason comments are allowed is so that the assembler will ignore the recognition information provided by the disassembler.

The standard Motorola mnemonics are used for the disassembler. In the assembler the use of the `Q` (quick), `A` (address), and `I` (immediate) suffixes after the mnemonic is optional. If a suffix is not included, the quick form of the instruction is going to be used whenever possible. The mnemonic may, in some cases, be followed by a `.B`, `.W`, `.L`, or `.S` extension. Extensions are prohibited in instructions that have either no or only one size.

☞ Extensions are prohibited in instructions that have no size or only one size. This includes instructions such as `MOVE DO,SR` and `SWAP DO`.

If the A000 trap names have been loaded, the A000 traps are displayed by their names. All trap names begin with an underscore. Unlike the 68000 mnemonics, the trap names contain both upper and lower case. In some cases the names are followed by a single hexadecimal digit; this occurs if bits 8 (10 for stack-based traps) through 11 of the trap word have been set in a nonstandard way. The standard values of bits 8/10 through 11 are:

1 for `_GetZone`, `_MaxMem`, `_NewPtr`, `_NewHandle`, `_HandleZone`, `_RecoverHandle`, `_GetTrapAddress`, `_PtrZone`, `_PurgeSpace`, and `_NewEmptyHandle`.
2 for `_HFSDispatch`.
0 for all other OS traps.
8 to B for all toolbox traps.

You may also enter A000 trap names to be assembled. If you wish, you may enter the previously mentioned hexadecimal digits after the names. They may even be expressions as long as they evaluate between 0 and $F. Of course, you do not have to worry about upper and lower case when entering A000 trap names, since all input not between quotes is converted to upper case anyway.

There are two extra instructions which do not have Motorola's mnemonics. They are `????` and `ROM`. They both have one argument that is a number between 0 and $FFFF. They are mnemonics used for unimplemented instructions. The disassembler uses the `ROM` opcode for unnamed A000 traps. `????` is used for all other unimplemented instructions. The assembler doesn't distinguish between the two opcodes. It also allows the use of these opcodes to assemble A000 traps even if the names of the corresponding A000 traps have been loaded.

Most addressing modes are standard. Numeric expressions may be used anywhere numbers are allowed except in the *names* of data and address registers. There are nevertheless several differences from the standard, and those are explained below.

The PC relative with offset and PC relative with index and offset addressing modes may each be specified in one of four ways: `^destination`, `*`, `*+offset`, and `*-offset` for the PC relative with offset addressing mode and `^destination(Rn)`, `*(Rn)`, `*+offset(Rn)`, and `*-offset(Rn)` for the PC relative with index and offset mode. `*` stands for the current program counter value (the address of the first word of the instruction).

☞ The `^` symbol may be omitted if doing so would not make the instruction ambiguous. Specifically, it may be omitted in all branch and `DBcc` instructions. Omitting it in an operand of a `MOVE` instruction would probably cause the addressing mode to be interpreted as absolute or register indirect with offset.

☞ Keep in mind that labels may be used anywhere in expressions.

Be careful about using `USP` as an addressing mode. It may only be used as a variable in `MOVE` instructions between the user stack pointer and an address register. Since these instructions have only one size, long, a mnemonic extension is prohibited. An attempt to use `USP` in any other instruction, for instance `MOVE USP,DO`, will cause the `USP` to be interpreted as a value and evaluated as an absolute addressing mode whose address is the current value of the user stack pointer. See the Numbers section for an explanation. The same problem will occur if you try to use `SSP` or `SP` in an operand.

The register list after a MOVEM instruction is standard. The register may be listed in any order and separated by either dashes to indicate ranges or slashes to separate two registers. All these generate the same instruction:

```
MOVEM.L D0/D1/A3-A0/D6-D7,-(A7)
MOVEM.L D0-D1/D6-A3,-(A7)
MOVEM.L D0/D1/D6/D7/A0/A1/A2/A3,-(A7)
```

As in the second example, if a range contains both an address and a data register, the address registers are considered to "follow" the data registers. You may not enter a list that doesn't contain any registers.

Here is a summary of the addressing modes:

`Dn`	Data register direct
`An`	Address register direct
`(An)`	Register indirect
`(An)+`	Postincrement register indirect
`-(An)`	Predecrement register indirect
offset`(An)`	Register indirect with offset
offset`(An,Rns)`	Register indirect with offset and index
address	Absolute
`^`*address*`, *,` `*+`*offset*`, *-`*offset*	Relative with offset
`^`*address*`(Rns),` `*(Rns),` `*+`*offset*`(Rns),` `*-`*offset*`(Rns)`	Relative with offset and index
`#`*number*	Immediate
`USP, SR, CCR`	Implied register

`Rns` is an abbreviation for either a data or an address register optionally followed by either a word or longword size indication. Some examples of `Rns` are `D0`, `A0`, `D3.W`, and `A7.L`.

Next is a table of the possible ranges of numbers. For the -128 to 255 and -32768 to 65535 ranges the values above 127 and 32767, respectively, are just positive equivalents of the negative values.

1 to 8	$00000001 to $00000008	Values in ADDQ, SUBQ.
0 to 15	$00000000 to $0000000F	TRAP value and value after A000 trap name.
-128 to 127	$FFFFFF80 to $0000007F	MOVEQ value.
-128 to 255	$FFFFFF80 to $000000FF	Byte immediate values, offset in register indirect with offset and index addressing mode.
-32768 to 32767	$FFFF8000 to $00007FFF	Absolute short addressing mode.
-32768 to 65535	$FFFF8000 to $0000FFFF	Word immediate values, offset in register indirect with offset addressing mode.
0 to 65535	$00000000 to $0000FFFF	???? instruction.
all	$00000000 to $FFFFFFFF	Absolute long addressing mode and long immediate values.
*-126 to *+129		Offset in relative with index and offset addressing mode.
*-126 to *+129 (excluding *+2)		Offset in a short branch.
*-32766 to *+32769		Offset in a relative with index addressing mode or a long branch.

During reverse scrolling the disassembler tries to find the preceding instruction, but that is not always possible. To make a guess at the length of the preceding instruction it goes up to 128 words back in memory. If it finds that no preceding instruction exists, it goes back one word and disassembles from that location. If more than one preceding instruction is possible, it chooses one of them. In both cases it will refresh the entire window, significantly slowing the reverse disassembly.

The assembler currently supports only the 68000 instruction set; 68020, 68030, 68881, and 68851-specific instructions and addressing modes are neither assembled nor disassembled.

Breakpoints

Only one `Breakpoints` window can be open at a time. It contains up to seven breakpoints. To enter or change an address of a breakpoint move the cursor to the appropriate place on the lower line, enter the new address of that breakpoint, and press **Return**. Avoid having two breakpoints at the same address. To delete a breakpoint do the same thing except instead of an address type a dash. You may also change more than one breakpoint at a time by separating the addresses and/or dashes with spaces. For instance:

Before Changes	007000	007432	--------	012FFE	--------	--------	--------
Changes	7410 - 7422 (Return pressed here)						
After Changes	007410	--------	007422	012FFE	--------	--------	--------

The breakpoints are implemented as `TRAP #$F` instructions. They are not put into memory until *after* the first instruction is executed after you leave the Monitor. This prevents you from having to remove a breakpoint in order to continue your program after it hit that breakpoint. Unfortunately this also has some side effects; see **Trace Flag Side Effects** for more information and some warnings. See also **Breakpoints in the Exception Handling** section.

You can cause many problems by setting breakpoints in sections of code that will be subsequently moved or written to the disk. The breakpoints will remain in the code, but they will be no longer recognized by the Monitor as breakpoints; instead, the Monitor will treat them as `TRAP #$F` instructions. Moreover, you will not be able to remove these breakpoints unless you know what instructions were "under" them before they were set.

The same problem as stated above may happen if you have breakpoints set and click the **Monitor** button in the Main Dialog to re-initialize the Monitor. Fortunately, this should not happen often since there is little reason to set breakpoints while running the TMON loader.

The difference between a breakpoint and a `TRAP #$F` instruction is that after a breakpoint has been encountered the next instruction to be executed is the instruction "under" the breakpoint, while after a `TRAP #$F` instruction the next instruction to be executed is the next instruction in the code.

Registers

The `Registers` window contains saved values of all 68000 registers. All registers are displayed as 8-digit hexadecimal values except the status register, which is displayed in binary with the flags indicated as either upper or lower case letters. The interrupt mask in the status register is displayed as three binary digits. To change any of the registers move the cursor to the appropriate register, enter a new value, and press **Return** or **Enter**. See **The Cursor and the Editing Facilities** for an explanation. As in `Breakpoints`, more than one register can be changed at a time.

When changing the status register enter the name of the flag in lower case, a `0`, or a period to clear a flag, or the name of the flag in upper case or a `1` to set the flag. Beware of changing the `S` flag. If its state is changed and **Enter** is pressed, in the next field the other stack pointer is going to be expected, causing the button bar to blink. Therefore, it is best to press **Return** after changing the supervisor flag.

A trick is used to disable the conversion of flags to upper case. The entire status register is enclosed in quotes, allowing you to use upper case to set flags and lower case to clear them.

The Monitor won't allow odd values in the program counter or either stack pointer when running on a 68000. On a 68020, odd values are allowed in the stack pointers.

The Registers window currently displays only the 68000 registers; the extra 68020, 68030, 68881, and 68851 registers are not shown and may not be changed.

Heap

The `Heap` window displays the contents of a heap zone. When it is first opened, it displays the application heap zone, but it may be toggled between the application and system heap zones by pressing **Return** or **Enter** with the cursor on the top line. The top line displays, in addition to the name of the current heap zone, its location and number of free bytes, both in hexadecimal. The addresses of the zones are stored in SysZone ($2A6) for the system zone and ApplZone ($2AA) for the heap zone. These two locations are maintained by the Macintosh operating system; you should normally not be changing them.

The Monitor will crash if the contents of either of these locations (SysZone or ApplZone) is odd or does not point to RAM.

The `Heap` window displays the position, length, and other data for each block in the heap zone. For each block the following information is displayed:

- An asterisk if the block is immovable or a space otherwise.
- The address of the beginning of the block in hexadecimal. Memory Manager's block header for this block is stored in the 8 *preceding* memory locations.
- The logical length of the block in hexadecimal.
- The size correction as a single hexadecimal digit. The size correction is the difference between the logical length and physical length-8. The physical length is the difference between the address of the next heap block and the address of this one. The size correction represents the number of unused bytes in this heap block.

One of the following phrases:

`Free`	a free block.
`Nonrel`	a nonrelocatable block.
`Handle at $.. (LPR)`	a relocatable block. The address of the handle is given. The three letters in parenthesis are the flags found in the high byte of the handle, set if they are in upper case or clear if in lower. **L** is set if the block is locked, **P** is set if the block is purgeable, and **R** is set if the block is a resource.
`INVALID`	any block that is not consistent. See the comment on reliability below.

Nonrelocatable and relocatable blocks may then contain further information helpful in identifying them. All such information except the resource file information is generated by a user routine, and is therefore subject to change and customization. If you find the identification of blocks inadequate, you are welcome to try to change it to suit your needs. Information on doing this is listed in **Creating Your Own User Functions**.

All relocatable blocks with the R flag set are checked against the list of open resource files and their resources. If they are present, their resource file number, type (ASCII letters in quotes), and ID number are displayed. All other blocks are passed to the user routine which should identify as many of them as possible. The default routine identifies the following:

UnitTable	$11C	A block containing all of the device control blocks.
DSAlertTab	$2BA	The Dire Straits alert table.
FCBs	$34E	A block containing all of the file control blocks.
WDCBsPtr	$372	A block containing all of the working directory control blocks.
Scrap	$964	Memory scrap.
WMgrPort	$9DE	A grafPort used by the Window Manager.
OldStructure	$9E6	A saved structure region used by the Window Manager.
OldContent	$9EA	A saved content region used by the Window Manager.
GrayRgn	$9EE	The rounded region defining the desktop.
SaveVisRgn	$9F2	A region used by the Window Manager.
MenuList	$A1C	The current menu bar list.
ParamText0-3	$AA0	The parameters in the last ParamText call.
TEScrap	$AB4	TextEdit scrap.
FinderInfo	CurrentA5+$10	The Finder information handle (in system heap).
VCB $..		Volume control block.
Resource map $..		Resource map of the given resource file.
Driver storage $..		Storage for the given driver.
Window #$.., kind $..		A window found by following the window list. The first number is the number of the window (0 is the frontmost window, 1 the next one, etc). The second number is the value of windowKind for that window.

(The hexadecimal numbers are either handles or pointers to the items above.)

In addition to the items listed above, the components of WMgrPort and all windows on the window list are identified. These components are identified first by either the phrase (Window @$..), which indicates the location of the window to which they belong, or (WMgrPort), indicating that they belong to the WMgrPort. After one of these two identifications one of the following messages is displayed:

VisRgn	The region of the window which is visible on the screen.
ClipRgn	The clipping region.
PicSave	Data for a picture being saved.
RgnSave	Data for a region being saved.
PolySave	Data for a polygon being saved.
StrucRgn	Structure region of window.
ContRgn	Content region of window.
UpdateRgn	Update region of window.
WData	Window-defined data.
WTitle	The title string.
WPic	Window's picture used for updating.
DlgItemList	Item list (dialog windows only).
TEHandle	TextEdit record (dialog windows only).
Item #$.. type $..	An item in the window's DlgItemList (dialog windows only). The first number is the item number (first item is 0, second 1, etc.), and the second number is the item type.
Control	Any control belonging to the window. This is displayed only if the control could not be identified as an item in that window's dialog list.

☞ If the Heap window crashes, "hangs", or causes the screen to flicker when opened or scrolled, the identification routines are getting lost, and you should turn them off. You also may want to turn them off for other reasons; for example, it is possible that identification might slow the Monitor too much. That is most likely to happen if you write your own heap identification routines, and do it inefficiently; the standard identification routine is quite fast. Anyway, if for any reason you want to turn off either identification of non-resource objects or identification of resources, use the Options window.

A `Heap` window may be scrolled up or down by using the arrows on the right side of the window. There are two quick ways to scroll the window up to the top of the heap: either close the window and then reopen it or type **Tab, Return, Tab, Return**. This will toggle the window between the two heaps and back, with the side effect that at each toggle the window will display the top portion of the heap. Since there is no quick way to scroll the window to the bottom of the heap, it is recommended that `Heap` windows be left open as long as they are needed. No harm will be done if the window is left open while the information in the heap changes; the window will just update itself. It is sometimes interesting to watch an open `Heap` window while the `Scramble now` user area function is repeatedly executed.

☞ More than one `Heap` window may be opened by using the Shift key. Two or more windows may even be opened to the same heap zone without harmful effects.

☞ Sometimes it is possible for a `Heap` window to turn completely blank except for the top line. This will happen if the window was initially displaying an area near the bottom of the heap and then the heap contracted. In that case you can either do nothing, scroll the window up until you see the heap blocks, or follow one of the procedures described above for quickly moving the heap window to the top of the heap. For a demonstration of this phenomenon move the `Heap` window to the bottom of the application heap and then use the `Heap` function in the user area, causing some blocks to be purged.

You cannot enter any information into `Heap` windows. Any changes to the heap structure have to be done by opening `Dump` windows to the appropriate places and changing data in them. A `Heap` window may then be used to verify that the changes were done correctly.

There is a quick way of scanning the heap on either a `Dump` or `Assembly` window (or in both). Open one of these windows and anchor it to `V` by typing `0 (V)`. Then place the cursor in a `Heap` window on the line of the block at which you would like to look and press **Return** or **Enter**. That action will cause `V` to be set to the address of the block at which the cursor was located and *then* the cursor to be moved to the next block of the `Heap` window. By repeatedly pressing **Return** or **Enter** the entire heap may be scanned without having to type the addresses of the blocks. Remember, however, that the cursor in the `Heap` window will always be on the line after the line containing the block currently being displayed in the `Dump` or `Assembly` window.

☞ Finally, here is some information about error checking and reliability: aside from the values of SysZone and ApplZone, the `Heap` window does *complete* error checking of the heap blocks it displays. When addresses of handles are displayed, the handles are checked to make sure that they point to the block. All pointers are checked to make sure that they are even and non-NIL and that they point to real memory. Any odd physical block lengths are rejected as are blocks with impossibly large sizes. The heap zone pointers in the block headers of relocatable blocks have to point to the heap zone which is currently displayed. Any blocks which do not pass the error checking are displayed as `INVALID`. You may use a `Dump` window to find why they are considered invalid.

Only one area may cause problems with reliability of `Heap` windows. In order to make the speed of these windows acceptable, the resource file maps are not fully checked when relocatable heap blocks are being identified. Should a damaged map claim that it has 10,000 resources, all will be examined in search of the one that is to be identified. Nevertheless, as with the heap zones themselves, all applicable pointers and handles in the resource file maps are rejected if they are NIL or odd.

The supplied user routine does error checking on all parts of structures it examines. In this case, error checking means that all pointers are checked for NIL or odd. The routine will not stray on invalid data structures for long. However, should you still encounter problems with the identification routine, you may turn it off by using the `Options` window. One indication of these is the screen becoming fuzzy when a `Heap` window is opened; this is caused by some memory accesses above ROM on the Macintosh 512K, 512Ke, Plus and SE.

⇨ Blocks which are displayed valid in the `Heap` window are usually valid. There is one important exception: if, for one reason or another, some of the handles which are traced by the `Heap` function have been disposed but not set to NIL, they may still point to master pointer blocks. If another handle is allocated using one of the master pointers, it will be erroneously identified as the heap object to which that master pointer used to belong. This problem will usually be encountered if not all managers have been initialized. In this case some heap blocks may be incorrectly identified.

File

A `File` window displays the contents of any open resource file. When a `File` window is first opened, the numbers of all open resource files are displayed on the second line of the window. You may then enter the resource number of the file you wish to examine on the top line. The system file is usually file $2 and the application file is usually the next higher number. If, after you are done examining one file, you wish to switch to another file, you may either enter the number of that file on the top line of the window or press **Return** there, causing a listing of the numbers of all open resource files to reappear.

Once you have entered a number of an open resource file, the contents of the file's resource map are displayed in the window. The top line of the window contains the address of the resource map as well as three flags which apply to the entire map. These three flags are stored in the 22nd byte of the resource map and are displayed as follows:

`r`	The map is read-write	`R`	The map is read-only
`c`	No compaction necessary	`C`	The map will be compacted
`w`	The map was not changed	`W`	The map was changed and will be written to disk

For each resource its type, ID number, flags, location, and optionally name and system reference data are displayed. The type is displayed first, in ASCII between single quotes, followed by the resource ID displayed as a four-digit hexadecimal number. The flags follow. They are displayed as upper case letters if they are set or periods if they are clear. The following abbreviations are used for the flags:

.	Local reference	`R`	System reference (64K ROMs only)
.	Load into application heap	`H`	Load into system heap
.	Not purgeable	`P`	Purgeable
.	Not locked	`L`	Locked
.	Not protected	`T`	Protected
.	Not preloaded	`1`	Preloaded
.	Don't write into file	`W`	Write into file
.	U flag clear	`U`	U flag set

After the flags either the memory location of the resource is displayed or `nowhere` if the resource is not in memory. The name of the resource (also in quotes) is then displayed if it is present.

`File` windows are very similar to `Heap` windows in the aspects of scrolling. In fact, the last few paragraphs of the description of `Heap` windows apply to `File` windows as well with the difference that `File` windows do not use any identification user routines. There are two more minor differences: in order to quickly move a `File` window to the top of the file, instead of typing **Tab, Return, Tab, Return**, type **Tab** and then **Enter**. This will re-enter the current file number into the top line of the window, also causing the window to be moved to the top of the file. Finally, when browsing through the file using a `Dump` or `Assembly` window anchored to `V`, you will find that blocks which have not been loaded into memory are ignored. Pressing **Return** or **Enter** on a line containing one will not change the value of `V`.

Just as in the `Heap` windows, any information displayed in the `File` windows is checked for accuracy. NIL or odd pointers will not cause the `File` windows to crash, although there may be more elaborate ways of crashing them. Any invalid files are simply not displayed. Any invalid resources are displayed as `nowhere`.

Exit, GoSub, Step, and Trace

These four functions all leave the Monitor, starting execution at the current PC. They differ in the duration before they return to the Monitor. `Trace` returns to the Monitor as soon as the next instruction is executed; `Step` is just like `Trace` except that it treats A000 traps as units; if the next instruction is an A000 trap, it is allowed to complete, and only then does control return to the Monitor. `GoSub` also like `Trace`, except that both A000 traps and `JSR` or `BSR` instructions are treated as units. `GoSub` is a quick way of skipping subroutines and executing just the main body of the program or a procedure. Finally, `Exit` leaves the Monitor indefinitely; that is, until the next exception.

All four functions restore all registers, the screen, and the cursor before they leave the Monitor. They do not put breakpoints into memory until the second instruction executed to allow continuing after a breakpoint.

When the Monitor is reentered after GoSub, Step, or Trace, one of two messages may appear near the top of the screen. "Trace interrupt at $_____" will usually appear; if, however, an entire subroutine or A000 trap was executed, "The A000 trap or subroutine has returned" will appear instead.

It is possible for the Monitor to refuse to proceed, displaying the interesting message, "I don't want to execute the next instruction." This will usually happen when continuing would not normally be to your best advantage. Specifically, the Monitor will refuse to continue if the next instruction is _SysError. (Exception to the exception: _SysError will be executed if the word value of D0 is a "harmless" system error value, where "harmless" is defined by the user area. In the default user area values less than 0, 30, 31, 42, and greater than 99 are harmless.) Also, for GoSub and Step only, _LoadSeg, _Launch, _Chain, and any A000 traps with both the stack-based and auto-pop bits set cause the Monitor to refuse to execute the next instruction. These latter traps are restricted for GoSub and Step because these functions do not know where these traps return; unlike other A000 traps, these do not resume at the instruction following the trap.

☞ Use either Exit with a breakpoint or Trace if you want to continue execution without losing control after hitting a _LoadSeg, _Launch, or _Chain. Traps with auto-pop bits set should only present a problem in older code; the auto-pop bit is not used any more. If you do find one, examine the stack to find the return address, set a breakpoint there, and use Exit.

➪ Due to the way these routines function internally, be careful if the next instruction is MOVE SR, *dest*. The trace bit will be set in the saved value of SR, and you should clear it before continuing. This problem arises because these exit functions, including Exit, use the trace flag to single-step the next instructions, and *then* put in the breakpoints and perform other tasks (like re-entering the Monitor).

☞ Even though the Monitor uses the trace flag for its internal stepping purposes, it will work correctly if you set it in the Registers window. A trace interrupt will be generated after every instruction, even if you use Exit. Keep in mind that the trace flag is cleared by any A000 traps encountered.

Although GoSub and Step are quite clever, you should be aware of some of the problems they may cause. In order to obtain control after the subroutine or A000 trap returns, they save the address of the next instruction in the Monitor's variables and pass a dummy address pointing to a Monitor routine to the trap or subroutine about to be executed. Obviously, if that subroutine examines or changes the return address it got on the stack, this scheme will fail. You will have to avoid skipping through subroutines or A000 traps that examine or change the return address. A few A000 trap routines examine the return address and execute patches if they were called from a particular place in ROM; this is used as a method of fixing ROM bugs. This is a mild version of the problem described above, as it will cause trouble only if you use GoSub or Step on the A000 trap in the ROM place to which the patch is attached; the patch will not be executed.

Another potential area of problems is interrupting the subroutine or A000 trap that was called by GoSub or Step. The interrupt could be a press of the interrupt button as well as a breakpoint, illegal instruction, address error, user area A000 trap exception, or another exception. Whatever the cause, after that interrupt you may wish to step through your code at the new PC and use GoSub or Step again, on another subroutine. *This will work correctly* up to *eight* levels of recursion. Also, nothing harmful will happen if one of the levels of recursion is never completed. The Monitor can keep track of up to eight pending subroutine or A000 trap returns, and it will halt execution as it encounters each one.

Finally, in a case similar to the above one, suppose that the subroutine or A000 trap called by GoSub or Step is interrupted. After the interruption you choose to single-step through the rest of the subroutine, including the final RTS, JMP (A0), (or whatever) statement. Instead of getting the expected "Trace interrupt at $_____" message you will instead get "The A000 trap or subroutine has returned," but everything else will work as expected. The subroutine's return address will be removed from the Monitor's eight-entry list of return addresses mentioned above. If you wish to examine it, the eight-entry list is stored at USER-$2F0 [longword array], and the order of priorities of assigning the next entry is at USER-$2F8 [byte array].

Options

The Options window lets you enable/disable seven features of the Monitor. You may want to disable them because they crash, take too much time, or don't do anything useful. Whatever the reason, you may disable and re-enable any of the features by moving the cursor to the correct line and pressing Return or Enter.

Master switch disables all label recognition.

Scan resources disables the scanning of resource files by the label routines. This turns off resource/ID resources completely. Since there is now no way of distinguishing CODE segments, the embedded name labels are also inoperative. Resource-relative table labels can no longer be evaluated or recognized.

Scan label table disables the user area label table routines.
Scan for names in code disables the user area embedded name label routines.
Identify A000 traps disables the pseudo-label Monitor routine.
Scan resources disables the identification of resource types, IDs, and files in Heap windows.
Identify items disables the identification of other Heap window items via a user routine.

Number

The Number window asks you for a number or an expression, evaluates it, and displays it in hexadecimal, 32 and 16 bit decimal, ASCII, as an A000 trap name, and as a recognized address. It also sets the N variable to the value entered. The second line of the window contains, in order from left to right, N= (if the value displayed is equal to the current value of N), the value in hexadecimal, the value in signed 32-bit decimal, in parentheses the lower 16 bits of the value in decimal, and in quotes the value in ASCII. The A000 trap name corresponding to the lower bits of the value is next, followed by the label recognition information that would be generated if the value typed were an address. If the particular trap has no name, the number is displayed instead. If there is no recognition information available for the particular address in the window, the recognition field is left blank.

More than one Number window may be open on the screen at a time. Several Number windows can serve as temporary memories to remember values which you would otherwise have to write on paper.

The N= indicator is not as extraneous as it may initially seem. Try opening two Number windows and entering different numbers into them. If no Number window contains the N= indicator and you want to see the current value of N, enter N into any Number window.

User

The user area is one of the most powerful features of this Monitor. It allows you to create your own functions or use the predefined ones. This section is just an overview of using the functions; information on creating new ones and an explanation of the predefined ones are included later.

The top line of the user area contains the beginning address of the user area, which is also the value of the predefined variable USER. Afterwards the physical and logical sizes of the user area are listed. The physical size is set at boot time and cannot be changed. It indicates how much memory is reserved for a user area. The logical size is used by the Configuration functions only and indicates how much disk space the user area would take if it were saved. The logical size may be changed by typing a new value in its place and pressing Return or Enter. It must, however, be nonzero, a multiple of $100, and not greater than $7F00. More information on logical and physical sizes is in the Configuration section.

To use a function in the user area, move the cursor to the appropriate line. Type as many parameters after the colon as the function requires (some require none), and press Return. The function will be executed and the cursor will return to the position after the colon. The function may return results in other places on the line, usually within the curly ({ and }) brackets. Some functions also put numbers after the colon so that you can just press Enter to execute the function again. Still others affect Monitor's registers, usually V. Functions can be created, like Leave TMON in the predefined area, that leave the Monitor altogether, while others may just provide additional parameters to be used by other User functions.

Most functions within the predefined area give a listing of the parameters they expect within parenthesis before the colon. If there is no such listing, the function probably doesn't need any parameters and is executed just by pressing **Return**. If the wrong number of parameters is supplied, the button bar will flash.

Sometimes the function name itself is displayed in parentheses. That means that several functions have been encoded on a single line, and you can cycle through them by pressing **Return** immediately after the colon. A good example of such a function is `Print` (which should not be confused with the `Print` command in the button bar).

Print

This function sends the contents of the active window to a serial port. See the **Configuration** section for more information about which port is used and how to change data like the baud rate. You may stop the printing by pressing the mouse button. If a handshaking protocol is active, you may have to hold the mouse button for a long period of time (possibly 20 seconds or more), because the state of the mouse button isn't checked while the Macintosh is waiting for the other RS232 device to allow it to send another character. Also note that since the mouse click is executed, if you tell it to print by clicking the mouse button over the `Print` area in the button bar, you will give the Monitor another command to print the topmost window, which is just what you wanted to avoid! If an error occurs while printing, its error code will be stored on the `Print` line in the user area if you are using one of the predefined user areas. An error code of 1 will be stored if the printing is interrupted by the mouse button. An error code of 2 is displayed after an attempt to use the user area's `Print file` function on a nonexistent resource file. See also the explanation of the `Print` function in the user area and the **Printing Problems** section for some of the possible communication problems.

Never use ⌘-Interrupt to stop a long printout; press the mouse button instead. If you do not heed this advice, the Monitor may behave in ways *stranger than could be imagined.*

Mouse Unfreeze

This is the only function that does not appear in the button bar at the top of the screen. It can be executed from the keyboard only by holding down the ⌘ key while typing M. This function unfreezes the mouse and at the same time turns off both serial ports. The mouse is probably frozen if it appears on the screen but doesn't move or doesn't appear on the screen at all. The Monitor must respond to keyboard commands if you want to use the unfreeze. The main cause of mouse freezing is accessing memory locations above ROM. Programs that crash or otherwise go out of control frequently do that. Accesses to such memory locations tend to reset the SCC chip, turning off the mouse interrupts. This option will turn those interrupts back on. This is most useful on a Macintosh 512K, 512Ke or Plus.

Note that the use of Mouse Unfreeze leaves the serial ports in an unstable state. In particular, using any AppleTalk function after having used Mouse Unfreeze may result in a system which is in a state where re-entry to TMON is not possible.

If you should happen to freeze the mouse by opening a `Dump` or `Assembly` window with an address in that memory range, *first* move the cursor to the top of that window by typing **Tab** or using ⌘D or ⌘A, then change the address of that window to zero and press **Return**, and finally use the **Mouse Unfreeze** function. In `File` windows do the same thing but press **Return** alone, causing just the list of file numbers to be displayed. If using the unfreeze now doesn't help, follow the procedure for `Heap` windows.

If a `Heap` window causes the mouse freezing or if any of the above procedures doesn't work, you have to close the offending window. That requires you to move the mouse to the close box. In most cases that can be done by moving the mouse while repeatedly typing ⌘M. The mouse will move in small jumps, but with luck you will be able to close the window. If even this doesn't work, hold down the ⌘ key and press the **Interrupt** button. Afterwards you can use the unfreeze function. This will reset the Monitor, possibly damaging it, but now at least you can use it.

If this option was performed successfully, a message window will appear near the top of the screen informing you that the mouse has been unfrozen and that you should not try to use the serial ports. You may, however, use the Monitor's `Print` functions.

Exception Handling

The Monitor intercepts all vectors except the ones essential to the functioning of the Macintosh, although you may change which vectors are intercepted by modifying the user area. For extra security the Monitor's vectors are stored again into low memory every time the Monitor is entered unless a **Configuration** option is changed (See **Configuration**). The Monitor also has a self-check feature that will tell you if the Monitor has become unreliable. If it detects an error, it will reset the entire Monitor and display a message. You may then take an appropriate course of action. Finally, the user area functions may themselves generate exceptions that are intercepted by the Monitor.

Normal Exception Messages

When an exception is intercepted by the Monitor, the registers are saved and breakpoints removed. Then the current screen, cursor, and cursor position are saved, the hardware is switched to display the main screen page if the alternate one was used on a Macintosh 512K, 512Ke, Plus or SE or swapped into one-bit mode on a Macintosh II, the Monitor's window and background are displayed along with a message explaining why the Monitor was entered, the user area's initialization routine is executed if present, and control of the computer is turned over to the Monitor. This section deals primarily with the messages that appear at this time.

The messages for most exceptions are self-explanatory. There is one thing, however, that may require an explanation. When the message gives the current value of the program counter, it sometimes states that the exception happened *before* that value and sometimes *at* that value. The program counter reported is always the value saved by the 680x0 upon handling the exception. The 680x0, however, sometimes saves the address of the next instruction and at other times the current instruction. This is the reason for the difference.

Address and Bus Errors

The messages for address and bus errors give extra data on 68000 machines. The program counter is given along with the address that was accessed when the error happened. The program counter value most likely isn't the address of the instruction that caused the error; in most cases it is somewhere nearby that instruction, although branch and jump instructions may affect the saved value of the program counter. The function code is displayed in parenthesis after the access address and tells what type of access that took place:

Message displayed	Function code saved by the 68000
`user data`	001
`user program`	010
`supervisor data`	101
`supervisor program`	110
`exception`	111
`illegal`	000, 011, or 100

The next field, `instruction`, tells whether the processor was executing an instruction at the time of the error. The only time it is *not* executing an instruction is during fetching an exception vector. The last field, `mode`, tells whether the access was a read or a write.

On a 68020, only a simple bus or address error message is reported.

Breakpoints

The breakpoints are `TRAP #$F` instructions. TMON distinguishes them from true `TRAP` instructions by checking the program counter against its list of breakpoints. If it is present in that list, a breakpoint message is given; otherwise, a trap message is given. See the **Breakpoints** section in **The Monitor's Functions** for more information.

System Error

The system error vector is also intercepted by the Monitor. All system errors except the ones listed in the User Area (30, 31, and 42 are the default) will cause the Monitor to be entered with an appropriate message.

Interrupt Button

The **Interrupt** button on the programmer's switch on the side of the Macintosh generates interrupt exceptions with priority levels between 4 and 7 (only 7 is generated on a Macintosh II). If the Monitor isn't currently executing, it will be started and an appropriate message will appear at the top of the screen. If it is currently executing, the interrupt will be ignored.

Sometimes pressing the **Interrupt** button may have no effect. If something goes wrong with the Monitor and you desperately want to regain control, hold down the ⌘ key while pressing the **Interrupt** button. This may re-initialize the Monitor to its original state, but that action may also erase some of the Monitor's variables, causing a message stating that the Monitor has been damaged to appear on the screen. Do not use `Exit` or `GoSub` after you have pressed **⌘-Interrupt** unless you initialize the program counter, stack pointer, status register, and any other necessary registers.

The ⌘-Interrupt action is one of last resort and should not be used often, as it may cause unpredictable damage to the Monitor.

On the Macintosh 512K, 512Ke, Plus, and SE, the processor's interrupt level can be set high enough so that neither Interrupt nor ⌘-Interrupt will work. When this happens, there is no way to stop the machine. Either press the reset button, or turn the system off and on again.

Self-Check

Whenever there is a window on the screen the Monitor does a self-check . If the self-check detects some signs of damage to the Monitor's code or a stack overflow, it will re-initialize the Monitor, just as if you pressed **⌘-Interrupt** (see the previous section). There are three things that could cause the Monitor to display a message informing you that it has been damaged:

One of the ways the Monitor could be damaged is if its code has been modified. If that happens, you will get only a message stating that the Monitor has been damaged. In this case some function of the Monitor will no longer work because its code has been changed.

The Monitor also checks to make sure that its stack stays within the area of memory designated to it. If it overflows, you will get a message stating that the Monitor has been damaged.

In this case some user functions will no longer work because the stack is located immediately after the user area.

The third way the Monitor could be damaged is if an exception occurs while the Monitor is executing. The Monitor is executing anytime the Monitor's screen is visible. This includes execution of a user function by pressing **Return** in the user window. This does *not* include execution of user routines by calling them from an outside program. If this type of an error occurs, you will get two messages, one stating the nature of the exception and the other stating that the Monitor has been damaged. The moral of this is to make sure that your user routines are "safe" by using code to prevent exceptions such as address errors from happening in them.

User Exceptions

The user routines that are called from an outside program may enter the Monitor by executing `TRAP #$F` instructions from within the user area, displaying the message "`User trap:`" followed by text from the user area. Details on this are included in the **User Routines Entering the Monitor** section. To see an example of this, use the `Leave TMON` function in the standard user area.

Possible Problem Areas

Here is a discussion of some of the problems you might encounter while using the Monitor.

Mouse Freezing

See the **Mouse Unfreeze** section in **The Monitor's Functions**.

Interrupting the Vertical Retrace

When you press the **Interrupt** button to enter the Monitor, there is a small chance that you will interrupt the vertical retrace handler routine. This could have a number of effects, ranging from none to a system crash when you leave the Monitor. Although this possibility can't be eliminated, its probability can be significantly decreased by making your program's vertical retrace queue short if it has any and by not moving the mouse when you press the **Interrupt** button.

Can't Regain Control of the Monitor

If **Interrupt** does not seem to work, hold down ⌘ and press **Interrupt**. Should this still be unsuccessful after a few tries, you probably will not be able to regain control of the Monitor and should give up and press **Reset**. See the **Interrupt Button** section for more information on this topic.

Trace Flag On

If you keep getting a trace interrupt after leaving the Monitor, you are either single-stepping or the trace flag in the Status Register is set. Clear it. Remember to clear the trace flag in the saved SR if you use `Exit`, `GoSub`, `Step`, or `Trace` when the next instruction is `MOVE SR,` *dest*.

Windows Crash or Are Too Slow

If the `Assembly`, `Number`, or `User` windows crash when opened or scrolled, turn off the labels using the `Options` window. If you can isolate the label routine that is causing the crashes, you may turn if off and leave the others on. If the `Heap` windows crash, you will have to turn off one or both of the options in the `Options` window dealing with `Heap` windows.

Follow the above procedure if you believe that `Assembly` or `Heap` windows are too slow to be convenient. Remember that a large slow window *visible* on the screen will slow all of the Monitor's operations, not just the ones dealing with that window. Covering such windows or decreasing their sizes may help.

Printing Problems

If you encounter problems with printing, make sure that the information in the **Communications** menu of **Configuration** is correct. Make sure that the Chooser desk accessory does not think that the serial port you're trying to use is connected to AppleTalk. It is also possible that the Monitor may not print correctly if some other program has opened and used the serial port designated for printing from the Monitor. If this is the case, try using the other port for doing printing from the Monitor. Also, since printing uses the Memory Manager, it may not work if the heap is in an inconsistent state.

Debugging Existing Applications

Using the Monitor with existing applications presents an array of problems. The Monitor is, nevertheless, flexible enough to allow it to be used with almost any program written for the Macintosh. That does not mean that you will not have to make adjustments. Some common changes which have to be made with some applications are disabling the **Vector Refresh** option or loading the Monitor into the system heap instead of high memory. The first of the actions listed should be done if you encounter `TRAP` exceptions; the last if you find that the program you are debugging uses the alternate graphics or sound page on a Macintosh 512K, 512Ke, Plus or SE.

The Monitor particularly dislikes changing the interrupt vectors and a few of the A000 trap vectors. Avoid changing the vectors `_SysError` and `_PostEvent`. If `_SysError` was changed, the Monitor will put it back to its previous address as soon as it regains control unless **Vector Refresh** was disabled. Do not change `_PostEvent` in such a way that events such as key down, mouse down, and mouse up are not posted; if you do, the Monitor will fail.

The ability of loading the Monitor into the system heap is a controversial one. If all Macintosh programs were well-written, they should not be affected by having a Monitor in the system heap. It appears, however, that some bugs in programs appear only if the system heap exceeds a certain size. For that reason the option of loading the Monitor into high memory was provided. An example of a bug that will appear only if the Monitor is in the system heap is in an application running under the 64K ROMs that calls `_SetTrapAddress` to a routine in the application heap. That shouldn't be done, but if it is, it will cause no problems unless the routine is more than $10000 bytes above RAMBase (which points to the beginning of the system heap). You, of course, will avoid all such errors, and might even find it constructive to load the Monitor in to the system heap to make sure that none are present.

In some cases you still might have little idea why the Monitor can't be used with some programs. In that case, use `Trap intercept` to stop the program at the beginning (intercepting `_InitGraf` works well), and use `GoSub` to trace the program execution. If you find that the program crashes in one subroutine, you now know where that subroutine is and can try the process again except that the next time step through that subroutine. You can then use this divide-and-conquer approach to quickly find the instruction that gives the Monitor indigestion. Once you know what is wrong, you may be able to bypass or fix it.

Using the Disk Cache, RAM Disks, and Other High-Memory Drivers

TMON respects the space reserved for the disk cache, RAM disks, and other programs which reside in high memory (hereafter called "RAM disks") as long as they do not interfere with TMON's exception vectors and code. If there is a RAM disk in high memory, and if it has allocated its space properly, TMON will load below it and set the top of memory address (BufPtr) below itself. RAM disks, on the other hand, should respect TMON; if they don't, they will cause problems. In particular, they should not assume that BufPtr points to the beginning of their memory space, as it does *not* once TMON has been loaded.

If you experience problems with using TMON and a RAM disk at the same time, try reversing the order in which you load them. If you loaded TMON first, load the RAM disk first, or vice versa. If the problems persist, try to load TMON into the system heap either before or after loading the RAM disk (See **Loading Position**).

⌘-Shift-1 to ⌘-Shift-4 Usage in the Monitor

These function keys are not active in the Monitor. Depending on which ROMs and system software you're using, when you use a function key, either nothing may happen or it may be saved and executed after you leave the Monitor.

The Configuration Menus

This chapter contains some additional information about the various items in the configuration menus.

The File Menu

Save User Area saves the current user area, which includes user area code, current settings of parameters to the user routines, and the state of the Monitor's windows. If you **Save** a user area with the name "User Area", it will become the default user area and be loaded every time you start the Macintosh with the TMON Startup loader. It is also used every time open the TMON icon from the Finder, unless you use another user area's icon to start TMON from the Finder or use the **Monitor...** button. If there is no default user area on the disk, an internal copy of the user area with all functions present will be used instead.

The user areas are saved using the logical length given in the user area. If the logical length is greater than the physical length, the extra space is filled with zeros. If that user area is later loaded with the Monitor, its logical length will be used to set the user area's physical length.

☞ The procedure above must be followed in order to enlarge user areas.

The Options Menu

The settings from the **Options** menu are stored at the beginning of each user area, and may be loaded and saved by loading and saving user areas. The **Communications**, **Vector Refresh**, and **VBL Tasks** options take effect immediately; the others will take effect only if you save their desired states in a user area, restart the computer, and load that user area with the Monitor. That can be done by either saving the user area with the name "User Area", or starting the Macintosh with another disk and opening that user area's icon from the Finder, or using **Monitor...** to select that user area to boot the Monitor. **Monitor Size** is not really an option because it does not allow you to change anything.

Communications

There are some communication options available which are not present here; see **User Configuration Area** for a more technical description of the other options.

Any changes you make are effective immediately.

It is possible to change the settings displayed here directly from the Monitor, which may be advantageous in the middle of debugging when this window cannot be invoked. See **User Configuration Area** for the locations of the bits in the user area that can be changed.

Vector Refresh

Use this option only in extraordinary circumstances when the program you are debugging desires to handle some of its own exceptions. If you select **Refresh**, which is the default choice, the Monitor will keep storing its own exception vectors every time it gains control. If you select **Don't refresh**, the Monitor will store its vectors only once when it is initialized. The program that you are debugging may then replace the Monitor's vectors with its own vectors.

Actually there are other times when the Monitor will store its exception vectors. It will do that anytime it thinks that it has been damaged and every time you enter it using the **Monitor** button on the Main Dialog. No program except the Monitor, however, may intercept the A000 vector, trace vector, and TRAP #$F vector.

VBL Tasks

This option allows you to keep vertical blanking tasks running while the Monitor is active. If you select **Suspend VBLs**, which is the default, only the disk driver VBL task will be left running, turning off the floppy disk motor after a few seconds if it is spinning when the Monitor is entered. Leaving VBLs running is usually undesirable except in some special cases where network drivers are timing out while the Monitor is active.

Loading Position

The only reasons *not* to load the Monitor into high memory are if you are debugging a program that uses the alternate sound or video page (on the Macintosh 512K, 512Ke, Plus, or SE) or if the program interferes with the Monitor in high memory, which should not happen. Loading the Monitor into the system heap causes the bugs in some existing applications to reveal themselves, and may be a good way to check that the program you are debugging does not have the same problems.

☞ Changing this option will have no effect unless you save the new setting in a user area and use that user area to boot the Monitor.

Auto-Quit

Auto-Quit is most useful when used in conjunction with the TMON Startup loader. If you set this option, instead of stopping at the Monitor welcome screen, TMON displays in the middle of the screen a message stating that the Monitor has been installed and then automatically exits to the Finder. The first time the Monitor is entered it displays the reason for entering along with the welcoming message.

It is quite easy to override this feature. There are two ways to do this:

- If TMON is just loading, hold down either **Shift**, **Option**, or ⌘ or the mouse button. If you hold one down while the message "Welcome to Macintosh" disappears, you will see the Main Dialog or, if you are using TMON Startup, the Monitor's welcome screen.
- If, on the other hand, you hold down **Shift**, **Option**, ⌘, or the mouse button *after* TMON begins to load but before the "The Monitor is installed" message appears, the loading process will stop when the Monitor's welcome screen appears.

The auto-quit option function also works if you start TMON by double-clicking on the TMON or a user area icon in the Finder.

☞ Hold down **Shift**, **Option**, ⌘, or the mouse button while TMON is booting to temporarily override the auto-quit function.

☞ Changing this option will have no effect unless you save the new setting in a user area and start the Monitor with that user area.

Memory Size

This function just prints a summary of the Monitor's memory usage. The only way to change any of the sizes shown is to change the corresponding **Configuration** option, save the user area, reset the Macintosh, and then use the user area to boot the Monitor.

☞ Remember that this option shows the amount of memory actually used by the Monitor the way it is presently configured; it may or may not correspond with the configuration in the current user area. In other words, the other functions in the **Options** menu display and change the configuration in the user area, while this function displays the configuration in the code of the Monitor presently in memory, which cannot be changed without resetting the Macintosh.

Built-In User Area Functions

This chapter will not explain the internal structure of a user area or how to create your own user functions; that is covered in the next chapter.

In the explanations below the name of the function is in boldface followed by the unabbreviated name. Any parameters used are at the end of the line.

Do not execute any of the functions listed below with addresses above the end of ROM. Although the functions *should* reject them, avoid giving negative lengths.

If you have more than one A000 trap function active at a time, the functions are executed in the following order: `Trap record`, `Trap scramble`, `Trap discipline`, `Trap checksum`, `Trap intercept`, and `Trap signal`. If one of the functions fails and enters the Monitor, the remaining ones are not executed. Also remember that these functions are not executed on the first instruction after leaving the Monitor should that instruction happen to be an A000 trap. This is to avoid a situation where having `Trap intercept` set would prevent you from leaving the Monitor.

The pixel in the upper left corner of the screen is turned on while one of the A000 trap intercepting functions is executing (except on a Macintosh II). If you press the **Interrupt** button at that time, the interrupt will not be executed immediately but will be made pending. After the completion of the intercepting function, if an interrupt was pending, the Monitor is entered with the message "`User trap: interrupt`". In rare circumstances it is possible to interrupt the user area A000 trap intercepting dispatcher. If that happens, you will know it because the PC will point to the user area. Use `Exit`, and press **Interrupt** again.

Be very careful with using `Label file load`, `Load resource`, and `Leave TMON` while one of the A000 trap intercepting routines is active. Since these routines leave the Monitor, the A000 trap intercepting functions will be executed on any traps they execute! `Trap intercept`, for example, will intercept the traps and go back to the Monitor with the PC and registers set to the user area. If that happens, deactivate the offending trap intercept routines and `Exit` the Monitor.

A much worse problem than the one described in the previous note arises when one of the A000 trap intercepting routines intercepts a trap executed from an interrupt handler. One particularly common case is the interception of _PostEvent due to a mouse click or keyboard activity. *Usually* this is harmless if you realize what is happening, but sometimes the consequences can be strange indeed. Consider the following scenario, which happened to me several times: you have `Trap intercept` set to intercept all traps and are single-stepping through a program by typing ⌘S. Suddenly, however, you release the key at just the right time to generate a key-up event outside the Monitor (If you released the key while the Monitor was executing, nothing would happen since the trap intercepting routines are disabled then). After the `_PostEvent` was intercepted, the PC points to a place in ROM, and you are wondering what happened—all you did was single-step an instruction! You `Exit`, and only then do you arrive at the instruction following the one you stepped, but with one difference: the trace flag is now set in the status register! The Monitor uses the trace flag to single-step instructions, and it normally turns it off after the instruction is complete; you don't see it being used. In this case the Monitor got somewhat confused because it was entered at an unexpected time. Afterwards it no longer remembered to clear the trace flag. No large harm has been done; you may proceed once you clear the trace flag. The moral of the story is to avoid indiscriminately intercepting traps such as `_PostEvent`.

`Toggle Pages` (Toggle *memory/control/A000 trap* functions)

Switch among the three user area pages. The current page's function types are displayed. The three user area function types are `memory functions`, `control functions`, and `A000 trap functions`.

`Block Move` *src dst len*

Move a block of memory Len bytes long from Src to Dst. The source and destination ranges may overlap without adverse effects.

`Block Compare` *adr1 adr2 len*

Compare two blocks of memory against each other. If they match, the result is **Match**. If they don't match, the address in *adr1* of the mismatch is displayed and the computer looks for the first match after that memory location and puts that address after the colon along with a corresponding address from *adr2* and the number of bytes remaining. The numbers after the colon are initialized so that you can look for the next area of mismatch just by pressing **Enter** or **Tab**.

☞ The value of `V` is set to the address of the mismatch. You can anchor a `Dump` or `Assembly` window to `V` and then you won't have to type the address of the mismatch to see the area of memory around the mismatch. What's more, you can actually anchor *two* windows, one to `V`, and the other one to `V` plus an offset which is equal to the difference between *adr2* and *adr1*. This way you can look at both blocks at the positions of the mismatch.

If you have a block of memory that seems to be filled with a single hexadecimal value, you can use `Block Compare` to measure how far it extends by giving it the following parameters: *Starting address Starting address*`+1 7FFFF`.

`Fill` *bgn end val [vLen]*

Fill a block of memory with a value. The bgn and end numbers specify the boundaries of the block, inclusive, and val is the number that is to be stored into the block. The fill is a byte, word, or longword fill depending on val. If val is less than 256, a byte fill is performed; if it is less than 65536, a word fill is done; otherwise, the fill is a longword fill. The vLen value may be used to override that by explicitly giving the length of val: 1 for a byte fill, 2 for a word fill, and 4 for a longword fill.

`Find` (Find *byte/word* aligned) *val [vLen [bgn [end]]]*

Search for a pattern in memory. bgn and end specify the boundaries of the block that is to be searched and val is the target pattern. vLen is used in the same way as in `Fill`, but a length of 3 is now allowed. Also, if there is no *vLen* and *val* is between 65536 and 16777215, inclusive, the length will be assumed to be 3.

The default for *bgn* is 0. The default *end* is the end of RAM, which depends on the size of memory in the Macintosh. All parameters except *val* may be omitted, in which case the entire RAM is searched for *val*.

☞ If no parameters are supplied, the `Find` toggles between a byte and word aligned search. Word-aligned search is faster and is usually used to search for specific 680x0 assembly language instructions, which obviously must be word-aligned. *vLen* is forced to be either 2 or 4 in word-aligned search.

☞ Do not use word-aligned search to search for handles or pointers because the high bytes may contain flags which cause some valid matches to be missed. Handles and pointers should be searched with a *vLen* of 3.

☞ The value of `V` is set to the address of the match. You can anchor a `Dump` or `Assembly` window to `V` so you won't have to type the address of the match to see the area of memory around it.

If *val* isn't found in the specified range, `No Match` will be displayed. If it is found, the address is given, `V` is set to that address, and the numbers after the colon are adjusted to allow the searching for the next occurrence of that *val* by simply pressing **Enter**.

Aside from using labels, one of the most common ways of finding subroutines in your program is to search for a string that the subroutine is known to contain. In that case *val* is usually a quoted four-letter string.

`Template` (*WindowRecord/ControlRecord/TERec/ParamBlock*) *adr*

This function displays many of the pertinent fields of the following Macintosh data structures: `WindowRecord`, `ControlRecord`, `TERec`, and `ParamBlock`. Enter an expression which refers to the data structure. The field names and their values will appear in the window below the **Template** line.

☞ If no parameters are supplied, the **Template** toggles among the `WindowRecord`, `ControlRecord`, `TERec`, and `ParamBlock` data structures.

`Stack Addresses` *adr*

This function takes an address, checks to see if it is within RAM, even, and below **CurStackBase**. If it is, it checks the value at the address to make sure that it too is even. If it is, it displays the address and attempts to recognize it. If recognition is successful, the result is also displayed. adr+4 is left to the right of the colon so that you may continue through the stack by pressing **Return** after the address or by pressing **Enter** anywhere in the line.

☞ The address returned by Stack Addresses is also assigned to the V register so that you can anchor a window to V in order to view the data around that address.

☞ This function uses a default parameter of SP, which refers to the current value of the stack pointer.

`Stack Crawl` *adr*

This function takes an address and treats it as a pointer to a stack frame created by the 680x0 family's **LINK** instruction. It checks the value that is the saved linkage register value in a stack frame. If this is a valid address (non-nil and even) it is assigned to the N register.

The function that is the return address in a stack frame is checked next. If it is valid it is assigned to the V register and displayed. If the address is within range of a label and the appropriate labeling switch is on, the label and offset will also be displayed.

The value that is the address of the next deeper stack frame in a stack frame is checked. If it is valid it is displayed after the colon so that clicking after it and pressing **Return** or clicking anywhere within it and pressing **Enter** will repeat the process for the next deeper stack frame.

☞ The address returned by Stack Crawl is also assigned to the V register so that you can anchor a window to V in order to view the data around that address.

☞ Perhaps the easiest way to use Stack Crawl is to position the cursor to the right of the colon and press Enter to cycle through the various stack frame addresses until the parameter becomes the default (RA6) again.

`Load resource` *type id*

Load a resource into memory. Search for the given Type and ID in all open resource files in the order defined by the Resource Manager. If the resource doesn't exist, do nothing. If the resource is already in memory, give its address. Otherwise attempt to read the resource from the disk. If there is no error, load the resource and give its address.

This routine leaves the Monitor for technical reasons (to prevent problems with crashes and disk swapping). It reenters the Monitor as soon as the resource is read.

☞ If the resource is in a file on a disk not currently in any drive, you may be asked to insert the disk.

Do not press the **interrupt** button while this routine is executing. Also, any mouse clicks or key presses made after the Monitor has been left are saved in the event queue. See `Leave TMON` for details.

The heap must be in a consistent state for this routine. This routine may cause heap compaction.

`Print` (Print *Dump*) *bgn end*
`Print` (Print *Disassembly*) *bgn end*
`Print` (Print *File*) *file#*
`Print` (Print *Heap*) *heap#*

Print the requested information. Press the mouse button if you would like to interrupt printing in progress. This function is useful for printing memory ranges that are bigger than a single `Dump`, `Assembly`, `File`, or `Heap` window.

The *heap#* is zero for the system heap and non-zero for the application heap.

You can cycle through the four print routines by pressing **Return** without typing any parameters. These four functions were placed on the same line to save screen space.

The error number is stored here by both the above four routines and the `Print` routine in the Monitor. See the description of the Print command for an explanation of the error numbers.

`Look for labels between LINK/UNLK of Ax` *addressReg#*

This function defines which address register will be used when the label recognition routine looks for embedded labels in routines enclosed within LINK and UNLK instructions. Since most Macintosh programs use A6 as the LINK/UNLK parameter the default value for this function is 6.

`Label table` *[nLabels [loc]]*

This function must be used to allocate a table for table labels. Without a table the table labels cannot be used. Whenever you invoke this function, the old label table, if present, is cleared, and, if it was previously allocated as a system heap block, it is deallocated.

If you supply no arguments, the old label table is deallocated. If it was a system heap block, the block is released. If you supply one argument, a table of size 16*nLabels is allocated on the system heap. If there was enough memory for the table, its address is then displayed in loc. If there was not enough memory, no table is allocated, and the line is cleared.

If you supply two arguments, the old table is deallocated and the new table is assumed to begin at loc. No checks of legality of loc are made. loc must be even!

V is set to the address of the table, if one is present.

nLabels should not exceed $7FF. If it exceeds $7FF, it is treated modulo $800. If it is zero, the function behaves as if no parameters were given.

The system heap must be in a consistent state for this function to be used. Also, *loc*, if given, must be even and point to unused RAM.

Beware of pressing Enter on the line after allocating a table on the system heap. This will deallocate the table on the system heap and make the Monitor assume that the new table is in the same location with the same length as the old one. Unfortunately, the table now resides in a free block instead of a nonrelocatable block in the system heap, and may corrupt both the system heap and the Monitor if something is later allocated in the system heap.

`Label add/remove` *[lbl [adr [end]]*

Add and remove labels in the label table. The label table must be allocated. If no parameters are given, do nothing except clear the result information. The result information is displayed between the two curly brackets and is either blank if nothing was done or gives information about the last operation completed. lbl must be a label enclosed in double quotes (not an expression!). It is the label upon which the operation is being done.

If only lbl is given, it is removed from the table if it was present; otherwise, nothing happens. The appropriate information is displayed between the curly brackets.

If lbl and adr are given, lbl is added (or replaced if it already exists) to the table. adr is the address assigned to it. If the Scan resources option in the **Options** window is set and *adr* falls inside a resource, the label is stored as a resource-relative label in the table, and needs no explicit ending address. A message stating that the label has been added relative a given resource type is shown between the curly brackets.

If all three parameters are provided, or the label could not be stored as resource-relative, it is stored as absolute starting at *adr* and ending the recognition range at *end*. If *end* was not provided, it is set to *adr*+$800. *end* is the first byte *past* the recognition range, not the last byte of it. A message stating that the label has been added is shown between the curly brackets.

If the table is already full, the label is not added, and nothing appears between the curly brackets.

☞ When adding labels this routine automatically determines whether a label should be stored resource-relative or absolute. If you think you need more control, modify the label table directly. Add some dummy labels to it and then use a `Dump` window to change their data.

`Label file load` (Load .MAP label file)

This function reads a .MAP file and extracts from it labels that are inserted into the label table. Labels must be enabled and a label table must be allocated. When executed, this function leaves the Monitor and shows a standard file selection dialog. All files of type `TEXT` are displayed. If you press **Cancel**, the Monitor is reentered with the message "Bad load". If you select a file, it is opened and read. Any errors cause a return to the Monitor with the "Bad load" message. If the label table becomes full, any extra labels in the file are ignored.

The function has definitely not been optimized for speed in order to keep it simple. It should not be difficult to change it to read other file formats.

This is the format of the `TEXT` file. All spaces are completely ignored; they are not used as delimiters of any kind and are removed from wherever they appear. This means that they can appear in the middle of numbers or labels without being detected. Any string of characters beginning with a character below $20 or above $7E and ending with an equal sign (=) is considered a label. The two delimiters are, of course, not included in the text of the label. The label, in order to be entered into the label table, must be followed by two hexadecimal numbers separated by another delimiter, usually a colon. The first number is 0 for absolute labels and gives the `CODE` segment number for `CODE` resource-relative labels. The second number gives the value for absolute labels and gives the offset inside the `CODE` resource minus 4 for the relative labels. No range checking is performed. The recognition range ending address for absolute labels is set to the label address plus $800. If the label already exists in the label table, the old one is replaced. Resource-relative labels relative to resources other than `CODE` cannot be specified in .MAP files.

This function is by far the least reliable one in the entire Monitor. It requires most of the managers like QuickDraw, the Window Manager, Dialog Manager, Font Manager, TextEdit, and others to be initialized. Moreover, it uses the standard file dialog to select the file, which is very risky to do from a Monitor. For all these reasons it is recommended that you use this function only from *TMON when the Monitor first appears*. The only way to *assure* a consistent state is to execute this function the first time the Monitor screen appears (with the welcoming message) or to click **Monitor** in the Main Dialog.

This function does not restore the screen; instead, it uses the activate/update event mechanism of the program currently executing. That mechanism must be capable of supporting redrawing of the area of the screen containing the standard file dialog.

Any mouse clicks or key presses made while this function is executing are saved in the event queue. See `Leave TMON` for more details.

`Registers` *selector*

This is a set of three functions dealing with the alternate register set located in the user area.

The three functions were placed on the same line to save screen space. Selector is used to identify the function to be executed.

If Selector is zero, the Monitor's register set is copied to the user area's register set and the other two options become available.

If Selector is one, the user area's register set is copied to the Monitor's register set.

If Selector is two, the two register sets are exchanged.

If Selector is none of the above, nothing happens.

The PC in the user area's register set is shown in curly brackets.

This function is very useful in case you are anticipating a system crash in a routine you are debugging. You can save the registers and then execute the routine. Then, even if it crashes, you can restore the registers and do what you were doing previously. Also use this routine if you are testing small pieces of code from TMON. By saving registers and later restoring them you can still use Exit to exit to the Main Dialog without worrying about making your routine preserve registers.

`Leave TMON; queue events until mouse click`

This function leaves the Monitor and stays in a routine that does nothing except check for a mouse button click and reenter the Monitor when the mouse button is pressed. This function also allows you to move the mouse cursor in the program you're debugging without having the program react to that motion. That could be useful sometimes.

This function will fail with a system error $1C if the stack pointer is below the top of the application heap.

Do not press the interrupt button while this routine is executing.

Any mouse clicks or keystrokes are recorded as events, but since the Monitor does not use the event queue, they will remain queued until you exit the Monitor. This includes the mouse click which you use to return to the Monitor. Although it may be annoying at times, this side effect can also be very useful for testing how the program you are debugging responds to a rapid succession of events. You could, for example, use `Leave TMON` to click on several buttons of a dialog (you would have to use `Leave TMON` several times for this), and then exit the Monitor and see how the program responds. This method can discover some very subtle errors.

`Leave application` *selector*

This function clears all breakpoints, closes all open files including resource files but excepting the System file, and if the selector is 0 launches the Finder, or if the selector is 1 re-launches the currently active application.

`Leave application` uses the `_ExitToShell` trap to return to whatever application is named in the low-RAM global `FinderName` when the selector is 0. When the selector is 1, it passes the contents of the low-RAM global `CurApName` to the `_Launch` trap.

This function relies upon the file manager and resource manager data structures being in a consistent state in order to close all open files successfully.

`Shut down` *selector*

If the selector is 0, this resets the Macintosh. If the selector is 1, it will eject all mounted disks and reset the Macintosh. This is better than pressing the reset button because it ejects and unmounts all disks. In order for this to succeed with a selector of 1, the volume queue and file variables must be in a usable state.

`Trap record` *[t0 [t1 [PC0 PC1]]]*

Whenever an A000 trap in between t0 and t1 located between PC0 and PC1, both inclusive, is found, record it in a table. The defaults are the value of t0 for t1, 0 for PC0, and $FFFFFF for PC1. A null parameter line turns off the recording. The table into which the traps are to be recorded must be specified in the `Record` function; otherwise, nothing happens.

The traps are recorded in a table of 16-byte entries. In each entry the first word (bytes 0,1) is the A000 trap that was recorded. The second word (bytes 2,3) contains the low 16 bits of the value of `Ticks` ($16A) when the trap occurred. The following longword (bytes 4-7) is the address of the A000 trap. The values of the last eight bytes vary depending on the setting of bit 11 of the trap. If the bit was 0, the trap was probably a register-based trap, and the longword values of D0 and A0 are stored in the remaining eight bytes. On the other hand, if that bit was 1, the eight bytes from the top of the stack are given.

New entries are added to the beginning of the table. All remaining entries are shifted to make room for the new entry. The last entry is forgotten. Note that unless you specifically clear the table before you exit the Monitor, the old entries will remain in it. If fewer traps than the table size were recorded while out of the Monitor, the old traps will still remain at the bottom of the table. You can distinguish them from the new traps by the time value in bytes 2 and 3.

The indicator in the curly brackets on the `Record` line shows the number of new traps recorded since the last time the Monitor was exited.

There are several uses for this function. One is to generally view the sequence of traps executed by the program to troubleshoot it. Another is to intercept a single trap such as `_GetResource` and see which resources are required by the program. Finally, this function may be used for limited performance analysis because it records the times of execution of the traps. This function does not significantly affect the running time of the program; moreover, its execution time is independent of the size of the table; recording a new trap and shifting the table is just as fast for a 2000-entry table as for a 20-entry table. Try it! The other A000 trap intercepting functions, notably `Heap scramble` and `Checksum` on large ranges, do slow the program, however.

You will probably want to use a `Dump` window to view the record and a `Number` window to find the trap name from the number given in the records.

See also `Record` and the note at the beginning of this chapter for general information about the A000 trap intercepting routines.

`Record` (Where to record traps) *[fullStop nMsg [loc]]*

This function must be used to allocate a table for `Trap record`. Whenever you invoke this function, the old label table, if present, is cleared, and, if it was previously allocated as a system heap block, it is deallocated.

If you supply no arguments, the old label table is deallocated. If it was a system heap block, the block is released. If you do not supply *loc*, a table of size 16**nMsg* is allocated on the system heap. If there was enough memory for the table, its address is then displayed in *loc*. If there was not enough memory, no table is allocated, and the line is cleared.

V is set to the address of the table, if one is present.

If you supply all arguments, the table is assumed to begin at *loc*. No checks of legality of *loc* are made. *loc* must be even!

nMsg should not exceed $7FF. If it exceeds $7FF, it is treated modulo $800. If it is zero, the function behaves as if no parameters were given.

fullStop is a flag for use by `Trap record`. If it is zero, recording takes place until the Monitor is entered again. If it is nonzero, the Monitor is automatically invoked via a user trap at the moment the table overflows. The trap that would cause the table to overflow is not recorded.

If you have already allocated the table using a *loc*, you may clear it by pressing Enter on the line. You *do not* want to press Enter on the line if it was allocated as a system heap block, because the routine will believe that you are now giving it an address! To clear a table that is a system heap block, press Return with the cursor between *nMsg* and *loc*.

The indicator in the curly brackets shows the number of new traps recorded since the last time Monitor was exited.

The system heap must be in a consistent state for this function to be used. Also, *loc*, if given, must be even and point to unused RAM.

Beware of pressing Enter on the line after allocating a table on the system heap. This will deallocate the table on the system heap and make the Monitor assume that the new table is in the same location with the same length as the old one. Unfortunately, the table now resides in a free block instead of a nonrelocatable block in the system heap, and may corrupt both the system heap and the Monitor if something is later allocated in the system heap.

`Trap heap check, scramble, purge` *[zone#]*

Scramble the heap on A000 traps. Unlike the other A000 trap intercepting routines, this one does not give you a choice of traps on which it is executed. The heap is scrambled whenever *a trap that might trigger a heap compaction* in this range is intercepted; the Monitor is not entered. The traps that might trigger a heap compaction are: `_NewPtr` ($A01E), `_NewHandle` ($A022), `_ReallocHandle` ($A027), and `_SetPtrSize` ($A020) and `_SetHandleSize` ($A024) if the new length is greater than the old length. The Monitor is entered if the heap is somehow damaged. (If that happens, the heap will have been partially scrambled up to the location of the error.)

zone# selects the heap to be scrambled. Use 0 for the system heap and any non-zero number for the application heap. The scramble is enabled when the zone number is visible. Press Return on the line once to turn off the scrambling.

The "heap scramble" in the preceding two paragraphs may not really be a heap scramble. There are four choices possible; they are chosen by consecutively pressing Return on the line.

The *Check* choice just checks the heap for consistency without modifying it. It is useful for locating routines in your program that somehow damage the heap.

The *Check, purge* choice first purges all purgeable blocks from the heap and then checks it.

The *Check, scramble* choice checks and moves as many relocatable blocks around the heap as possible. This is used for finding handle dereferencing errors in programs, which are surprisingly common. The check and scramble are done simultaneously; a part of the heap may have already been scrambled when an error is found; nevertheless, the heap area immediately before and after the error is never scrambled. This option is highly optimized for speed within the constrains of space in the user area; still, it is quite slow.

Note that this option also clears all free blocks in the heap except in some cases the last one (to make the speed bearable; the last one is usually very large), but not to zeros. This provides additional assurance that dereferencing errors are eliminated.

The *Check, scramble, purge* choice does all three: a purge, a check, and a scramble.

A heap scramble consists of moving as many unlocked relocatable blocks in the heap as possible. This way a heap compaction is simulated every time one *could* happen. Any handle dereferencing errors which otherwise would be rare and random and very difficult to find because they would occur only on heap compactions are now made to occur consistently every time, making them much easier to find.

In addition to moving the free blocks in the heap the heap scramble erases (to a nonzero, odd value) all free blocks and consolidates any consecutive ones. The only free block that may not be completely erased is the last one. This is done for performance reasons.

It is possible that some relocatable blocks will not be moved. This will occur in the rare situation that one relocatable block is caught between two immovable ones.

`Heap` *zone#*

If no parameters have been passed to Trap check, scramble, and/or purge, this function displays the total amount of free bytes on the heap, the maximum number of contiguous bytes, and the number of bytes the heap may grow. The system heap is used if Zone# is zero, otherwise the application heap is used.

☞ The heap zone is compacted and all purgeable blocks are purged from it.

Do not use this function if the heap zone is inconsistent (has invalid blocks in it). Also avoid using it if the application program you are debugging does not expect the heap zone to change.

If parameters were passed to Trap check, scramble, and/or purge, this function changes to `Check, scramble` now which, when chosen, checks and possibly scrambles or purges the heap as described above according to the options currently set in `Trap heap scramble`. The action is performed immediately. The last heap zone entered into `Trap heap scramble` is used.

Be careful with this function (`Check, scramble` now). If the program you are debugging has any handles dereferenced at the time you invoke it, that program may later fail. If you are not sure whether this is the case, use `Trap intercept` to find the nearest A000 trap that might cause a heap scramble (see previous section). Before such a trap it is definitely permissible to scramble the heap.

`Trap discipline` (*lenient/strict*) *[t0 [t1 [PC0 PC1]]]*

This examines the parameters being passed to any traps about to be executed that lie within the defined trap range and optional PC range. If any of the parameters are questionable or incorrect, the Monitor will be entered with an appropriate message being displayed. The message will begin with a question mark ("?") and will provide the Toolbox type of the parameter in question, sometimes with additional information (e.g. `? selector, ? NIL address, ? odd address, ? address, ? string, ? string length, ? StringPtr, ? NIL StringPtr, ? jump table, ? THz, ? Zone, ? Ptr, ? Handle, ? empty Handle`, etc.). Note that when the monitor is entered due to discipline, the PC points to the trap whose parameters are questionable, and that the parameters are still on the stack or in the appropriate registers, where they can be examined so as to glean some idea as to what is wrong with them.

☞ If no parameters are passed to this function, it toggles between using lenient and strict discipline. If lenient discipline is being used, NIL string pointers will be converted to pointers to a zero-length string. If strict discipline is being used, the Monitor will be entered with an appropriate message being displayed if a rectangle is invalid (top is greater than bottom and/or left is greater than right) and the Monitor will be entered with an appropriate message being displayed if procedures passed to _SetTrapAddress are not in the system heap zone.

`Trap checksum` *[t0 [t1 [PC0 PC1]]]*

Whenever an A000 trap in between t0 and t1 located between PC0 and PC1, both inclusive, is found, do a checksum on the range specified in the `Checksum` range. The defaults are the value of *t0* for *t1*, 0 for *PC0*, and $FFFFFF for *PC1*. A null parameter line turns off the checksumming. When an A000 trap is encountered, if the result agrees with the checksum shown in the `Checksum` line, the function does nothing. Otherwise, it drops into the Monitor with a user trap message stating that the checksum has failed.

Note that once the checksum fails, it is *not* automatically recomputed. If you want to see when the area of memory specified in the `Checksum` line changes again, you have to press **Enter** on the `Checksum` line to recalculate the checksum.

`Checksum` was not written with emphasis on performance. It works very fast on small ranges, but was not really designed for checksumming large ones.

See also `Checksum` and the note at the beginning of this chapter for general information about the A000 trap intercepting routines.

`Checksum` *bgn end*

`Checksum` generates and displays a checksum generated from that memory range. It is used to check if a memory range has been changed through time or for comparing two memory ranges. This checksum will find most transposition errors as well as substitution errors.

The *bgn*, *end*, and generated checksum values are also used in `Trap checksum`, above. The default values are the beginning and end of the ROM, which is convenient, since the ROM doesn't change while the machine is on.

`Trap intercept` *[t0 [t1 [PC0 PC1]]]*

Intercept all A000 traps with the trap number between t0 and t1, inclusive, and which are in the block of memory from PC0 to PC1. The defaults are the value of t0 for t1, 0 for PC0, and $FFFFFF for PC1. A null parameter line turns off the intercepting. When a trap is encountered, the Monitor is entered with a user trap message. See also the note at the beginning of this chapter for general information about the A000 trap intercepting routines.

`Trap signal` *[t0 [t1 [PC0 PC1]]]*

Like the other A000 trap intercepting functions, this one executes on A000 traps with the trap number between t0 and t1, inclusive, and which are in the block of memory from PC0 to PC1. The defaults are the value of t0 for t1, 0 for PC0, and $FFFFFF for PC1. A null parameter line turns off this function.

Unlike the other trap intercepting functions, this one does nothing most of the time. It remains dormant until you press **interrupt** while holding down the **Option** key. Once you do that, this function will drop into the Monitor as soon as it is executed. It is very useful for stopping your program at a specific point as opposed to anywhere. Some possibilities are setting *t0* to `_SystemTask` or `_GetNextEvent` or setting *t0* and *t1* to all traps but restricting the PC to your main program.

This function learns that **Option-interrupt** was pressed because the Monitor tells it that. If you press **Option-interrupt** while this function is disabled or the Monitor is active, nothing happens.

Avoid pressing the ⌘ key too, which would have disastrous consequences (It does a complete Monitor reset).

See also the note at the beginning of this chapter for general information about the A000 trap intercepting routines.

Creating Your Own User Functions

The user area is a variable-length block of memory reserved by the Monitor. It has two purposes: to allow you to use the predefined and add your own functions to the Monitor and to store the **Configuration** and Monitor windows settings. The **Configuration** setting, a few other parameters, and the user area identification number are all stored at the beginning of the user area. They are followed by a linked list of names and other parameters for the routines. The routines themselves are at the end of the user area.

The user area routines must be relocatable. The user area is placed between Monitor's variables and the Monitor's stack, which could be anywhere in memory. Once the user area is loaded, however, it is never moved (but it could be saved and loaded into a different place).

This chapter is organized roughly in the order of the data in the user area. The description of the configuration bytes is first, followed by the descriptions of some special numbers and vectors that are also stored in the **Configuration** area. Then the names and finally the routines are described.

It is difficult to learn how to create your own user routines without looking at examples. You are encouraged to look at default user area source file (supplied on the disk) for any ideas.

All numbers are in decimal unless preceded by a dollar sign or the context implies that the number is hexadecimal. Bit 7 is the most significant bit of a byte and 0 the least significant.

If you have an old (TMON 2.585) user area, you must edit it before this TMON will read it. Re-assemble the user area, changing the configuration settings (bytes 0-683) at the beginning to make sure they make sense. At the same time set bit 3 of the byte at offset 4 in the user area; this indicates to TMON that this is a new user area.

The User Configuration Area

As stated earlier, the first 48 bytes of a user area are used to hold the current **Configuration** setting and some other interesting data. Here is a summary of this data; some of the items will be explained in more detail later.

Byte	Bit Range	Description
0-1	all	The length of the user area. It must be a multiple of 256, less than $8000, and can not be zero.
2-3	all	This user area's version/ID number. This number is for identification purposes only; no part of the program references it.
4	7	0 if the vector refresh is on; 1 if it is off.
4	6	0 if VBL tasks are to be disabled; 1 if VBL tasks must be left running.
4	5	0 if the Monitor is to be loaded into high memory upon booting; 1 if system heap.
4	4	1 if auto-quit is enabled; 0 if not.
4	3	1 if this is a new (TMON 2.8) user area, 0 if it is an old (TMON 2.585) one.
4	2-0	Unused; formerly was the amount of screen compression used; 0 indicated saving the entire screen (21888 bytes on a Macintosh 512K, 512Ke, Plus or SE), 1 a 10K compression, 2 a 4K compression, and 3 no compression. The values from 4 to 7 were illegal.
5	7	Set to 1 to inhibit calling the user identification routine for heap windows.
5	6	Set to 1 to inhibit scanning resources for heap windows.
5	5	Set to 1 to inhibit table labels.
5	4	Set to 1 to inhibit embedded name labels.
5	3	Set to 1 to inhibit scanning resources for label routines.
5	2	Set to 1 to inhibit pseudo-label identification.
5	1	Set to 1 to inhibit all labels (master switch).
5	0	Reserved. Must be zero.
6	7	The port used for printing. 0 is the printer port; 1 is the phone port.
6	6-0	Reserved. Must be zero.
7	7-2	Reserved. Must be zero.
7	1-0	The handshake used for printing. 0 is none, 1 is XOn/XOff, and 2 is hardware. A value of 3 will cause unpredictable results.
8	7-6	The number of stop bits used by the serial printing routines. 1, 2, and 3 are 1, 1.5, and 2 stop bits, respectively.
8	5-4	0 and 2 are no parity; 1 is odd, and 3 is even parity for printing.
8	3-2	0, 1, 2, and 3 are respectively 5, 6, 7, and 8 bits per byte for printing.
8-9	1-0;7-0	A constant determining the baud rate being used. The value is 115200 / baud rate - 2. For instance, use 4 for 19200 baud, 10 for 9600 baud, and 46 for 2400 baud. The constant is ten bits long.
10-11	all	Offset in the user area to an A000 hook routine; 0 if there is no such routine. For example, if the user area starts at $7000, and the A000 handler routine within the user area starts at $732D, the value stored here would be a $032D.
12-13	all	Offset in the user area to the location to store the `Print` error code or 0 if there is no such location. Whenever the `Print` function is used from the button bar or the predefined user area, the error code is stored at this word in the user area.
14-15	all	Offset to the first user routine's description block or 0 if there is none.
16-17	all	Offset to the heap window identification routine or 0 if there is none.
18-19	all	Offset to the user area initialization routine or 0 if there is none.
20-21	all	Offset to the user area entry routine or 0 if there is none.
22-23	all	Offset to the user area exit routine or 0 if there is none.
24-25	all	Offset to the user area label table recognize routine or 0 if there is none.
26-27	all	Offset to the user area embedded label recognize or 0 if there is none.
28-29	all	Offset to the user area label table evaluate routine or 0 if there is none.
30-31	all	Offset to the user area embedded label evaluate or 0 if there is none.
32-33	all	Offset to the packed table of user area A000 name additions.

34-35	all	Offset to the user area table of system errors to be passed through to the system.
36-37	all	Offset to the user area level 7 entry routine or 0 if there is none.
38-39	all	Offset to the user area level 7 exit routine or 0 if there is none.
40-43	all	Reserved for internal cursor patch routines.
44-63	all	Unused; reserved for future expansion. Must be zero.
64-71	all	Bitmap of exception vectors to be intercepted when the Monitor is first loaded or re-initializes itself.
72-79	all	Bitmap of exception vectors to be intercepted every time the Monitor gains control.
80-683	all	Internal data pertaining to the current state of the Monitor's windows.

Names and Local Storage in the User Area

At the 14th byte of the user area is a pointer to a linked list of user routine descriptors. The list is composed of entries described below:

First is the offset from the beginning of the user area to the next routine's descriptor in the list. The offset is a word. Next is another word offset from the beginning of the user area, this time to the starting address of a user routine. It is followed by the length of the routine's name and then by the name itself (described in the next section). An extra byte may be inserted after the last character of the name to make the next field on a word boundary. The parameter count byte is after the name. It contains a bit map indicating the number of parameters accepted by this routine. After that is a byte containing the length of the user routine's local variable space followed by the variable space itself. (The length of the local variable space is no longer used. It was once used by earlier versions of TMON, but the current one ignores it. It may be used again in the future.) The data after the variable space can consist of anything; in particular, it may be the routine code, or the name record of the next routine.

What's in a Name?

The names are usually not pure ASCII strings, although they may be. Usually they are much more complicated, including displaying of variables, ASCII values, or even conditionals. This is a powerful feature of the Monitor that simplifies creating your own user areas. Some functions described below refer to a pointer called *P*. It is an internal Monitor variable that is initialized to the beginning of the user routine's local variable space every time before the name is printed. In the following section "printing" means displaying on the screen, not the printer. Some excellent examples of control sequences in names are present in the source code of the default user area. You are encouraged to look at these to better understand "names" as described below. Here is a table of the "control codes" that can be used in a name:

`$00` `EndIf`
Cancel a preceding `IfElse`, `IfPos`, or `IfNeg`.

`$01` `IfElse`
Flip the condition of the last `IfPos` or `IfNeg`.

`$02` `IfPos`
If the byte at P is positive, execute the next section of code. If it is negative, skip until the next `EndIf` or `IfElse`. In either case increment P. The conditionals may be nestable to any reasonable level. Each `IfPos` may be optionally followed by `IfElse`, but must be followed by `EndIf` with one exception: it is not necessary to put an `EndIf` before the end of the string.

`$03` `IfNeg`
If the byte at P is negative, execute the next section of code. If it is positive, skip until the next `EndIf` or `IfElse`. In either case increment P. The conditionals may be nestable to any reasonable level. Each `IfNeg` may be optionally followed by `IfElse`, but must be followed by `EndIf` with one exception: it is not necessary to put an `EndIf` before the end of the string.

`$04` `Colon`
This is one of the more powerful commands in names. It prints a colon, but has a much more profound side effect. Everything printed after this command will appear to the right of the colon and provide a default for the user's editing. Some pre-defined user functions that use this control code are `Block Compare`, `Find`, `Checksum`, `Load resource`, and `Trap intercept`. No more than one `Colon` can be interpreted, although more than one may be present (using conditionals). Any `Colon`(s) encountered after the first one is interpreted, are ignored. If there is no `Colon` in the name, an implied one is inserted after the end of the name, so that there always is a colon on the line.

`$05 to $0E` `Skip`
Increment P by 1 to 10 bytes. $05 increments it by 1 byte, $06 by two bytes, etc.

`$0F to $16` `PrHex`
Print from one to eight hex nibbles from the memory pointed by P. P is incremented past the byte that contains the last nibble printed. For example, $13 causes the least significant nibble of the byte at P and both nibbles of the two following bytes to be printed as a five-digit hex number.

`$17 to $1E` `PrASCII`
Print 1 to 8 ASCII characters from memory starting at P. P is incremented past that memory block. ASCII values lower than $20 or higher than $7E are printed as tiny periods.

`$1F` `NoOp`
No operation.

`$20 to $7E` Print that ASCII character.

`$7F` Print a tiny period.

`$80` `DisAsm0`
Print the name of the A000 trap whose number is in the word at P. P is incremented past the word. The number at P is decoded in the same manner as in a `Number` window: bits 0 to 11 contain the trap number, and bits 12 to 15 are ignored. If the trap names are not present or the trap given has no name, its number (plus $A000 or $A800, depending on the trap) is displayed instead, preceded by a dollar sign.

`$81` `DisAsm1`
This is the same as `DisAsm0` except that the word at P is decoded in the manner an `Assembly` window would show it as opposed to a `Number` window. All 16 bits are significant; bits 12 to 15 should contain $A. A space and the hexadecimal digit indicating the value of bits 8 to 11 are printed when that value differs from the default. If the trap names are not present or the trap given has no name, this function is equivalent to the sequence `'$',PrHex+4`.

`$82` `Recognize`
Call the label recognize routine on the address given in the longword at P, and report the results. Up to 23 bytes of the destination string might be used. Nothing is displayed if the address could not be recognized.

`$83 to $FF` Unimplemented. Currently behave as `NoOps`.

The line is truncated to 84 characters including the colon. Names that are too long may provide too little editing space for the user.

Parameter Count

The parameter count byte defines the number of parameters allowed by the routine. A parameter is a number or an expression typed by the user on that routine's line. The parameters are separated by spaces and are automatically evaluated by the Monitor. The parameter count byte is actually a five-bit bit map contained in the least significant bits of the byte. The three most significant bits are flags, which should be set to 0. If bit 0 of the byte is set, the user routine may be called with no parameters; if bit 1 is set, it may be called with one parameter, etc. If the routine allows more than one amount of parameters to be present, it may find the number of parameters actually given and provide defaults for the missing parameters.

Bit 7 of the parameter count serves a special function. If it is set, a label is expected instead of the first two parameters. The label is checked for syntax but not evaluated. The first four characters of the label are given as parameter 0 (in D0), and the second four are given as parameter 1 (in D1). If there are less than eight characters, the missing ones are padded with blanks; if there are more, the extra ones are ignored. The label counts as two normal parameters.

Bits 6 and 5 are unused and reserved for future use.

Register Conventions

Upon entry to the routine the following values are present in the 68000 registers. All values are longwords.

D0 to D3	Parameters provided by the user or zeros if not present.
D7	The number of parameters supplied by the user (0 through 4).
A0	A pointer to this user routine's local variable storage.
A1	This subroutine's starting address.
A2	The user area's starting address.
A5	A pointer to the Monitor's variables (described later).
A7	Monitor's stack pointer. At least 200 bytes are available on the stack.

All other registers contain zeros. The user routine does not have to preserve any registers except, of course, A7. The interrupt level is set to zero upon entry to the user routine, but the user routine may set it to anything it wishes. The status register does not have to be preserved either.

There are, however, certain restrictions on the user routines that are called from the Monitor. They must not cause any exceptions; in particular, this includes address and illegal instruction errors and trace interrupts. There are some ways user routines may get around these restrictions if it is absolutely necessary, as described in the next sections.

The A000 Trap Intercepting Hook

At the beginning of the user area there is a word that contains either zero or an offset to a user A000 trap intercepting routine. This makes functions like `Trap intercept` and `Trap discipline` possible. The user routine pointed by the vector is executed before every A000 trap *except* traps that occur while the Monitor is executing (See the **Self-Check** section for a definition of when the Monitor is executing) and traps that occur while the first instruction is executing after leaving the Monitor (See **Trace Flag Side Effects** for more information on this topic).

The A000 trap routine must preserve all registers except the CCR. Upon entry all registers are the same as before the A000 trap except A5, which has been saved on the stack. A5 is initialized to the beginning of Monitor's variables. The standard user area contains a routine which is usually linked to this hook; it is called `A000Hook`. It obtains the trap number and PC and dispatches any user A000 intercept routines that are active.

The trap routine must either return to the routine that called it using `RTS` or restore all registers to their original states (this includes the A5 that was saved on the stack) and execute a `TRAP #$F` instruction, as described in **User Routines Entering the Monitor**. Examine the `A000Hook` routine for a safe way of doing that.

User Routines Leaving the Monitor

A user routine may leave the Monitor if it wishes to do so. It must initialize the register area in the Monitor's variables to the values the registers are to assume after the Monitor has been exited. This includes in particular the program counter, stack pointer, and status registers. After doing this the routine must execute a `JMP -12(A5)` instruction. A5 must contain the address of the Monitor's variables. At that time the Monitor executes a kind of an automatic `Exit` function. The register area in Monitor's variables will be described in a later section. Refer to the `Leave TMON` and `Load resource` routines in the user area source for examples of leaving the Monitor and re-entering it later, although that is not the clearest example because before leaving the Monitor the return status register and program counter values are already pushed onto the stack in anticipation of the `TRAP #$F` instruction that will be executed after the mouse button is pressed.

User Routines Entering the Monitor

User routines that are called from outside the Monitor or have exited the Monitor may reenter it by pushing a longword and a word on the stack and executing `TRAP #$F` *inside the user area*. That `TRAP #$F` instruction is interpreted as an attempt to reenter the Monitor only if it is located within the user area.

The longword pushed onto the stack is the value that will be loaded into the program counter in the `Registers` window. The word pushed onto the stack after the longword is the status register value that will be placed into the same window. The other registers in the `Registers` window come directly from the values left in the registers. Some examples of the usage of this feature are the `Leave TMON` routine and the routines that exit to the Monitor after a trap has been intercepted or an error in the heap has been found.

The Heap Window Identification Routine

Locations $10 and $11 of the user area contain a word offset from the beginning of the heap to the `Heap` window identification routine that is described in the `Heap` window section. The routine is used to identify heap blocks and may be customized. The routine is given a flag indicating the type of block in D3. A zero is a non-relocatable block, one is a non-resource relocatable block, and two is a relocatable block that has already been identified as a resource. Free and invalid blocks are not passed to the identification routine. The user identification routine will probably want to ignore type-2 blocks.

- D3 contains the flag described above.
- A0 contains the address of the beginning of the user area.
- A1 contains the address of the heap zone to which the block belongs.
- A2 contains a pointer to the area in which the text identifying the block is to be stored. Look at the listing of the default user identification routine to learn how to handle A2. Make sure that you do not run off the right edge of the string at A2.
- A3 is a pointer to the heap block to be identified.
- A4 is another pointer to the destination area. The difference between it and A2 is that while A2 points to the free space after the last word already present in the string, A4 always points to the same place: the position after the second space after the size correction digit.
- A5 points to Monitor's variables.
- A6 contains the address of the handle for relocatable blocks *only*.
- A7 is the stack pointer. At least 60 bytes are free on the stack.

The routine may destroy the contents of any registers except A2, A5, and, of course, A7. If the routine was able to identify the block, it should move A2 past the information it has written into the destination.

The routine should not take much more than 1/100th of a second to execute; longer times will tend to excessively slow the Monitor.

Make sure that the routine does not write outside the designated destination area. See the listing of the supplied routine for details.

The routine should check all data structures it uses to avoid address errors and following NIL pointers. It should not assume that anything is correct except in extremely time-critical cases.

The User Initialization Routine

There exists a routine in the user area that is executed immediately *before* the Monitor is first initialized and immediately before the Monitor is reinitialized after clicking the **Monitor** button in the Main Dialog if the Monitor is already present. Locations $12 and $13 of the user area contain a word offset from the beginning of the user area to the beginning of this initialization routine. These two bytes contain zeros if there is no initialization routine.

This routine is called *before* the Monitor initializes itself; this means that none of the Monitor's variables contain valid information. This also means that the Monitor's self-checking has not yet begun and, therefore, the Monitor can be patched without generating the message stating that the Monitor has been damaged. In fact, the main purpose for including this routine is to allow user areas that modify the Monitor to be made. A secondary purpose is to allow self-initializing user areas.

The routine does not have to preserve any registers except the high byte of SR and A7. On entry D0 contains 0 if the routine is called the first time and -1 every time thereafter (It can be called more than once if the **Monitor** button is used to re-initialize the Monitor). A5 points to the beginning of the Monitor's variables. None of the variables themselves, however, have been initialized. A7 points to the application program's stack, not the Monitor's stack.

This routine is *not* called if ⌘-**Interrupt** is pressed or the Monitor reinitializes itself due to one of the conditions listed in the **Self-Check** section of **Exception Handling**.

Make sure you know exactly what you are doing before attempting to patch the Monitor! Remember that none of Monitor's variables contain valid information when the initialization routine is executed. Your initialization routine also should not call any of the Monitor's routines if you are not sure whether such routines depend on the initialization of Monitor's variables.

The User Enter and Exit Routines

These user routines are called after entering and before exiting the Monitor. Their offsets are in bytes $14 and $15 for the enter and $16 and $17 for the exit routine. The routines do not have to preserve any registers except the high byte of SR and A7. On entry A5 points to the beginning of the Monitor's variables, and A7 points to the Monitor's stack. Look in the standard user area for examples of usage of these routines.

An alternate set of enter and exit vectors is located at offsets $34 through $37 in the user area. These vectors are identical to the above ones except that they are entered with interrupt level 7. The level 7 entry routine is called before the normal entry routine, and the level 7 exit routine is called after the normal exit routine. The interrupt level is maintained at 7 from the time an exception causing an entry into the Monitor takes place to the time of the level 7 entry routine call and from the level 7 exit routine call to the actual exit from the Monitor.

Some very large problems could arise if the user enter routine causes an address error. The Monitor will re-initialize itself, and, in the process, will call the enter routine again, causing another error. The cycle will thus continue, and you will be unable to regain control.

The User Label Routines

There are four user routines which are used by the Monitor's label system for label evaluation and recognition. Two of these routines do recognition: `_LSCAN`, at $18 and $19, for label table recognition; and `_CSCAN`, at $1A and $1B, for embedded name recognition. `_LFIND`, at $1C and $1D, evaluates table labels; and `_CFIND`, at $1E and $1F, evaluates embedded name labels. These routines are called only if labels are enabled and if they are not inhibited by `Options`. The register conventions are listed in the user area source code. These routines should take measures to avoid crashing on address errors. They should also be designed efficiently, as slow ones will excessively slow the Monitor.

The User A000 Name Table

If the word at offset $20 in the user area is nonzero, it is assumed to be an offset into the user area table of A000 trap names. The A000 name assembly and disassembly routines scan both that table and the Monitor's internal table; when a name is present in the user area table, it overrides the name in the Monitor's table. The format of the table is documented in the user area but subject to change.

The Monitor does not do error checking on the table. It is very sensitive to the format of the table and will crash if the word at offset $20 points to the wrong place or if there are any errors in the table.

The System Error Table

The word at offset $22 in the user area is an offset to a table of system errors that are passed by the Monitor to the system. These include the disk-switch dialog ("Please insert the disk...") and the power-off dialog. The table is composed of longwords terminated by a zero longword. Each longword is a range of error numbers to pass to the system. The high word is the lower end of the range inclusive, and the low word is the higher end of the range inclusive. All error numbers are signed. The ranges must be listed in ascending order. An error number not present in any of the ranges causes entry into the Monitor.

The Window List

Locations $50-$2AB of the user area are used by the Monitor for storage of the current state of the Monitor's windows. Byte $53 contains the current number of windows (including alerts) on the Monitor's screen. Locations $54-$2AB contain an array of up to twenty 30-byte records corresponding to up to twenty windows on the screen. The first window is the topmost, the second one is just behind the first one, etc. Each window record is 30 bytes long, and its first byte indicates the type of a window: $00 `Alert`, $01 `Registers`, $02 `Breakpoints`, $03 `User`, $04 `Number`, $85 `Dump`, $86 `Assembly`, $87 `File`, $88 `Heap`, and $09 `Options`. Type $0A is used internally. The other bytes of a window record indicate the window's length and position on the screen, and in some cases the data displayed in the window. Assembly window records, for instance, contain the lengths of the instructions displayed in them.

The Exception Vector Bitmaps

There are two bitmaps of exception vectors in the user area at offsets $40 and $48. Each bitmap consists of 64 bits corresponding to the approximately 64 exception vectors possible on the Macintosh. The first bitmap indicates the exception vectors into which the Monitor should put its vectors when it is initialized, and the second bitmap indicates the exception vectors that should be refreshed. Setting a bit indicates that the Monitor should take over the corresponding vector. The Monitor will never take over vectors at $00 and $04 (reset), $64, $68, and $6C (interrupt levels 1-3), or $F8 and $FC (used internally by the Monitor), regardless of the settings of the bits in the bitmap.

Clearing some of the bits (especially the ones corresponding to the `TRAP #$F`, A000, and trace vectors) will likely crash the Monitor.

The Monitor's Variables

This is an incomplete list of the Monitor's variables. Only the more useful variables are shown. You can learn how to use them by examining the user area source file. The locations of variables are offsets from A5. The lengths of the variables follow the locations.

$15 (B) `MONEXECUTING`
This byte is used to decide whether the Monitor is currently executing or not. $6B means it is executing, $29 means that the first instruction after an exit of the Monitor is executing, and any other value means that the Monitor isn't executing. Do not change this value unless you are sure what you are doing; the A000 trap intercept routine in the user area source contains an example of changing `MONEXECUTING`.

$1A (L) EVENTINTERCEPT
Saved address of the OS Event Manager routine while the Monitor is executing.

$1E (L) DESNIFF
Saved value of StkLowPt ($110) while the Monitor is executing.

$23 (B) BREAKPTMAP
A bitmap of the breakpoints. The MSB is used internally by the Monitor to indicate if the breakpoints are set at the present time or not. The seven least significant bits are set if the corresponding breakpoints have been set.

$24 (L*7) BREAKPOINTS
The addresses of the breakpoints or zeros if the breakpoints are reset.

$40 (W*7) BREAKSAVES
The values of the words "under" the breakpoints.

$4E (L) REG.PC

$52 (W) REG.SR

$54 (L) REG.USP

$58 (L) REG.D0
Registers D1 through D7 follow.

$78 (L) REG.A0
Registers A1 through A6 follow.

$94 (L) REG.A7

$98 (L) REG.NUM
The current value of N.

$9C (L) REG.V
The current value of V. Many user area routines change this location.

$1F7 (B) MONTRACETIME
Information on what to do if a trace interrupt occurs and MONEXECUTING is $29. 0 means enter the Monitor with trace flag clear, 1 enter the Monitor with trace set, $80 put in the breakpoints and leave, and $81 is used by GoSub to step through the JSR or BSR instruction.

$1F8 (L) SYSERRVECTOR
The saved system error vector. Jump to the address stored here to generate a system error.

$200 (L) SYSA000VECTOR
The saved value of the system A000 line exception vector.

$207 (B) USERIINFORM
This variable is cleared every time the Monitor exits. It is set to $FF if the interrupt button is pressed (without Option or ⌘ keys) while MONEXECUTING is $6B. USERIINFORM is also set to $01 whenever Option-interrupt is pressed (without the ⌘ key). This variable is examined by the user area INTERCEPT routine to determine if the interrupt button was pressed while one of its dispatched routines was executing. It is also used by the Trap Signal function.

$208 (8*B) ALCPCORDER
The order of allocation of ALCPCVALUES. 0 is the next to be allocated.

$210 (8*L) ALCPCVALUES
Up to eight return addresses for up to eight recursive invocations of GoSub or Step. See the section on GoSub and Step for more details.

$500 USER
This is the beginning of the user area.

The Monitor's Vectors

This a list of Monitor's vectors which may be accessed by jumping to or calling subroutines at *offset(A5)*.

-$08(A5) `_PRINT1`
This is for use of the printing routine only. See `PRINT` in the listing of the default user area.

-$0C(A5) `_EXITMON`
This routine is used for exiting TMON and has been described earlier.

-$14(A5) `_PRINT2`
This is for use of the printing routine only. See `PRINT` in the listing of the default user area.

-$18(A5) `_PUTASCII`
Given a byte in D0 and A2 pointing to a destination area, `_PUTASCII` stores either D0 if it is a valid ASCII character or $7F (a tiny period) if it isn't in (A2). A2 is incremented one byte. D0 is destroyed.

-$1C(A5) `_PUT1DIG`

-$20(A5) `_PUT2DIG`

-$24(A5) `_PUT4DIG`

-$28(A5) `_PUT6DIG`

-$2C(A5) `_PUT8DIG`
These routines take either the hex digit, byte, word, three bytes, or long word in D0 and put it in hexadecimal format at (A2). A2 is incremented by the corresponding number of digits. D0 and D1 are destroyed.

-$30(A5) `_NEXTCRESFILE`
Find the next file in the linked list of resource files. Check for NIL handles and address errors. Ignore files with address errors caused by accessing type maps. On entry D1 contains a handle to the next file. On exit, if Z if set, there is no next resource map. If Z is clear, A1 points to the type list, D0.W contains the file reference number, and D1 has a handle to the next file. No other registers are affected.

-$34(A5) `_FINDRES`
Find a resource given its type and ID. Check for address errors and NIL handles. On entry D2 contains the type and D3.W the ID. On exit, if D0 is -1, the resource was not found (maybe not loaded). If D0 is 0, D2 points to the resource, and A0 has the handle to the resource. D1, D4, A0, and A1 are destroyed.

-$38(A5) `_RECOGNIZE`
Call the recognize routine. On entry D2 contains the address to be recognized, and A2 must point to a destination string at least 23 bytes long. Upon exit, A2 is advanced past the recognition data stored in the string, and D0 is either 0 if the recognize was successful, or -1 otherwise. D1-D4, A0, and A1 are destroyed. All of the `Options` switches are obeyed.

-$3C(A5) `_CALLDISA000`
Call the A000 disassembly routine. On entry D2 contains the A000 trap to be disassembled (only the low 12 bits are significant), and A2 must point to a destination string at least 23 bytes long. The low byte of D0 is either $00 to display the digit indicator or $FF to inhibit it. Upon exit, A2 is advanced past the disassembly of the instruction, which is either the trap's name preceded by an underscore or the the trap's hexadecimal four-digit number preceded by a dollar sign. D0-D4, A0, and A1 are destroyed.

The Startup Loader

The TMON diskette contains a file named "TMON Startup." This file contains an INIT resource which loads the Monitor into RAM during the execution of INIT resources. This is useful in debugging any INIT resources which are executed after TMON Startup. To use TMON Startup, place it in your System Folder along with a copy of TMON and any user area file that you wish to use. When the system is re-booted, TMON Startup will load the Monitor into RAM according to the configuration found in the user area. The name of the resource file that TMON Startup uses to load the Monitor into RAM is in resource STR with an ID of 1000. Normally this string contains "TMON." It can be changed with a resource editor.

TMON Startup can be prevented from loading the Monitor into RAM by holding down the mouse button before TMON Startup executes. Conversely, TMON can be entered immediately after having been loaded by TMON Startup by holding down the Shift, ⌘, or Option key.

TMON Startup indicates its progress by displaying an icon near the bottom left corner of the screen. If it displays the TMON icon, all is well. If it displays the TMON icon with a question mark superimposed, it failed to find the file named in STR 1000. If it displays the TMON icon with a slashed circle superimposed, there was a resource error in opening the file named in STR 1000. If it displays the TMON icon with a skull and crossbones superimposed, there was a disk error in reading the Monitor into RAM.

It may be desirable to have TMON Startup be the first INIT file executed. If that is the case, its name must be changed so that it comes first in the System Folder, because external INITs are executed in alphabetical order by file name. For example, "! TMON Startup", will come before *any* INITs with names beginning with a letter.

Appendix A—Quick Reference

This is a compilation of the frequently used tables in the manual. See the appropriate sections of the manual for more detailed information.

Keys that May be Used in the Monitor

Tab	Move cursor to the top left position of the current window
Return	Process the contents of the line left of the cursor.
Enter	Process the entire line.
Clear (keypad)	Clear the line.
→	Move cursor left.
←	Move cursor right.
⌘A	Bring assembly window to front.
⌘B	Bring breakpoints window to front.
⌘D	Bring dump window to front.
⌘E	Exit.
⌘F	Bring file window to front.
⌘G	GoSub.
⌘H	Bring heap window to front.
⌘M	Unfreeze mouse.
⌘N	Bring number window to front.
⌘P	Print contents of frontmost window.
⌘R	Bring registers window to front.
⌘S	Step.
⌘T	Trace.
⌘O	Bring options window to front.
⌘U	Bring user window to front.
⌘-Shift	⌘-Shift-A, ⌘-Shift-D, ⌘-Shift-F, ⌘-Shift-H, and ⌘-Shift-N generate additional windows of the indicated type. See corresponding ⌘ keys above.

Keys that May be Used outside the Monitor

interrupt	Enters the Monitor.
Option-interrupt	Activates the user area signal function.
⌘-interrupt	Reinitializes the Monitor in emergencies.

Operators Allowed in Expressions

Binary Arithmetic	Binary Logical	Unary	Precedence
+ Addition	| Logical OR	+ Positive number	<>
- Subtraction	^ Logical exclusive OR	- Negative number	unary +-~@!
* Multiplication	& Logical AND	~ Logical NOT	*/\
/ Signed division		@ Long word indirection	&
\ Signed modulo		! A000 trap address	+-
			^
< and > may be used as parentheses.			|

Register References

Variable	Value	Register name
A0 to A7	RA0 to RA7	Address registers.
D0 to D7	RD0 to RD7	Data registers.
SP*	SP	Same as A7 or RA7. *For anchoring windows only.
SSP*	SSP	System stack pointer. *For anchoring windows only.
USP	USP	User stack pointer. (Normally unused in the Macintosh)
PC	PC	Program counter.
SR		Status register.
CCR		Condition code register.
N	N	The result of the last `Number` calculation.
V	V	Result of `Find`, `Heap`, and other functions.
USER		The beginning of the user area.
DSPT		The address of the ROM A000 trap dispatcher.

Dump Window Flags

P	Program counter	*	Breakpoint
S	System stack pointer	N	N register
U	User stack pointer	V	V register
0 to 6	Address register		

Assembly Window Addressing Modes

`Dn`	Data register direct
`An`	Address register direct
`(An)`	Register indirect
`(An)+`	Postincrement register indirect
`-(An)`	Predecrement register indirect
offset`(An)`	Register indirect with offset
offset`(An,Rns)`	Register indirect with offset and index
address	Absolute
`^`*address*, `*`, `*+`*offset*, or `*-`*offset*	Relative with offset
`^`*address*`(Rns)`, `*(Rns)`, `*+`*offset*`(Rns)`, or `*-`*offset*`(Rns)`	Relative with offset and index
`#`*number*	Immediate
`USP, SR, CCR`	Implied register

Items Identified by the Heap Window

UnitTable	$11C	A block containing all of the device control blocks.
DSAlertTab	$2BA	The Dire Straits alert table.
FCBs	$34E	A block containing all of the file control blocks.
WDCBsPtr	$372	A block containing all of the working directory control blocks.
Scrap	$964	Memory scrap.
WMgrPort	$9DE	A grafPort used by the Window Manager.
OldStructure	$9E6	A saved structure region used by the Window Manager.
OldContent	$9EA	A saved content region used by the Window Manager.
GrayRgn	$9EE	The rounded region defining the desktop.
SaveVisRgn	$9F2	A region used by the Window Manager.
MenuList	$A1C	The current menu bar list.
ParamText0-3	$AA0	The parameters in the last ParamText call.
TEScrap	$AB4	TextEdit scrap.
FinderInfo	CurrentA5+$10	The Finder information handle (in system heap).
VCB $..		Volume control block.
Resource map $..		Resource map of the given resource file.
Driver storage $..		Storage for the given driver.
Window #$.., kind $..		A window found by following the window list. The first number is the number of the window (0 is the frontmost window, 1 the next one, etc). The second number is the value of windowKind for that window.

Heap Window Handle Flags

l L Locked p P Purgeable r R Resource

File Window Map Flags

r R Read-only map c C Map will be compacted w W Map will be written to disk

File Window Resource Flags

- . R System reference (64K ROMs only)
- . H Load into system heap (opposite is application heap)
- . P Purgeable
- . L Locked
- . T Protected
- . 1 Preloaded
- . W Write into resource file
- . U U flag set

A000 Traps in Numerical Order

A000:	Open	A00D:	SetFileInfo	A11A:	GetZone	A027:	ReallocHandle
A001:	Close	A00E:	UnmountVol	A01B:	SetZone	A128:	RecoverHandle
A002:	Read	A00F:	MountVol	A01C:	FreeMem	A029:	HLock
A003:	Write	A010:	Allocate	A11D:	MaxMem	A02A:	HUnlock
A004:	Control	A011:	GetEOF	A11E:	NewPtr	A02B:	EmptyHandle
A005:	Status	A012:	SetEOF	A01F:	DisposPtr	A02C:	InitApplZone
A006:	KillIO	A013:	FlushVol	A020:	SetPtrSize	A02D:	SetApplLimit
A007:	GetVolInfo	A014:	GetVol	A021:	GetPtrSize	A02E:	BlockMove
A008:	Create	A015:	SetVol	A122:	NewHandle	A02F:	PostEvent
A009:	Delete	A016:	InitQueue	A023:	DisposHandle	A030:	OSEventAvail
A00A:	OpenRF	A017:	Eject	A024:	SetHandleSize	A031:	GetOSEvent
A00B:	Rename	A018:	GetFPos	A025:	GetHandleSize	A032:	FlushEvents
A00C:	GetFileInfo	A019:	InitZone	A126:	HandleZone	A033:	VInstall

A034: VRemove
A035: Offline
A036: MoreMasters
A037: ReadParam
A038: WriteParam
A039: ReadDateTime
A03A: SetDateTime
A03B: Delay
A03C: CmpString
A03D: DrvrInstall
A03E: DrvrRemove
A03F: InitUtil
A040: ResrvMem
A041: SetFilLock
A042: RstFilLock
A043: SetFilType
A044: SetFPos
A045: FlushFile
A146: GetTrapAddress
A047: SetTrapAddress
A148: PtrZone
A049: HPurge
A04A: HNoPurge
A04B: SetGrowZone
A04C: CompactMem
A04D: PurgeMem
A04E: AddDrive
A04F: RDrvrInstall
A050: RelString
A051: ReadXPRam
A052: WriteXPRam
A053: ClkNoMem
A054: UprString
A055: StripAddress
A057: SetAppBase
A058: InsTime
A059: RmvTime
A05A: PrimeTime
A05B: PowerOff
A05D: SwapMMUMode
A260: HFSDispatch
A061: MaxBlock
A162: PurgeSpace
A063: MaxApplZone
A064: MoveHHi
A065: StackSpace
A166: NewEmptyHandle
A067: HSetRBit
A068: HClrRBit
A069: HGetState
A06A: HSetState
A06C: InitFS
A06D: InitEvents
A06E: SlotManager
A06F: SlotVInstall
A070: SlotVRemove
A071: AttachVBL
A072: DoVBLTask
A077: CountADBs
A078: GetIndADB
A079: GetADBInfo
A07A: SetADBInfo
A07B: ADBReInit
A07C: ADBOp
A07D: GetDefaultStartup
A07E: SetDefaultStartup
A080: GetVideoDefault
A081: SetVideoDefault
A082: DTInstall
A083: GetOSDefault
A084: SetOSDefault
A090: SysEnvirons

A0B5: GoDriver
A0B6: WaitUntil
A0B7: SyncWait
A0B8: SoundDead
A0B9: Disptch
A0BA: IAZInit
A0BB: IAZPostInit
A0BC: LaunchInit
A0BD: CacheFlush
A0BF: Lg2Phys
A0C0: FlushCache
A0C1: GetBlock
A0C2: MarkBlock
A0C3: RelBlock
A0C4: TrashBlocks
A0C5: TrashVBlks
A0C6: CacheWrIP
A0C7: CacheRdIP
A0C8: BasicIO
A0C9: RdBlocks
A0CA: WrBlocks
A0CB: SetUpTags
A0CC: BTClose
A0CD: BTDelete
A0CE: BTFlush
A0CF: BTGetRecord
A0D0: BTInsert
A0D1: BTOpen
A0D2: BTSearch
A0D3: BTUpdate
A0D4: GetNode
A0D5: RelNode
A0D6: AllocNode
A0D7: FreeNode
A0D8: ExtBTFile
A0D9: DeallocFile
A0DA: ExtendFile
A0DB: TruncateFile
A0DC: CMSetUp
A0DE: DtrmV1
A0DF: BlkAlloc
A0E0: BlkDealloc
A0E1: FileOpen
A0E2: PermssnChk
A0E3: FndFilName
A0E4: RfNCall
A0E5: AdjEOF
A0E6: Pixel2Char
A0E7: Char2Pixel
A0E8: HiliteText
A0EE: CkExtFS
A0EF: DTrmV3
A0F0: BMChk
A0F1: TstMod
A0F2: LocCRec
A0F3: TreeSearch
A0F4: MapFBlock
A0F5: XFSearch
A0F6: ReadBM
A0F7: DoEject
A0F8: SegStack
A0F9: SuperLoad
A0FA: CmpFrm
A0FB: NewMap
A0FC: CheckLoad
A0FD: TETrimMeasure
A0FE: TEFindWord
A0FF: TEFindLine
A801: SndDisposeChannel
A802: SndAddModifier
A803: SndDoCommand
A804: SndDoImmediate

A805: SndPlay
A806: SndControl
A807: SndNewChannel
A808: InitProcMenu
A809: GetCVariant
A80A: GetWVariant
A80B: PopUpMenuSelect
A80C: RGetResource
A80D: Count1Resources
A80E: Get1IxResource
A80F: Get1IxType
A810: Unique1ID
A811: TESelView
A812: TEPinScroll
A813: TEAutoView
A814: SetFractEnable
A815: SCSIDispatch
A816: Pack8
A817: CopyMask
A818: FixAtan2
A819: XMunger
A81A: XGetZone
A81B: XSetZone
A81C: Count1Types
A81D: XMaxMem
A81E: XNewPtr
A81F: Get1Resource
A820: Get1NamedResource
A821: MaxSizeRsrc
A822: XNewHandle
A823: XDisposHandle
A824: XSetHandleSize
A825: XGetHandleSize
A826: InsMenuItem
A827: HideDItem
A828: ShowDItem
A829: XHLock
A82A: XHUnlock
A82B: Pack9
A82C: Pack10
A82D: Pack11
A82E: Pack12
A82F: Pack13
A830: Pack14
A831: Pack15
A832: XFlushEvents
A833: ScrnBitMap
A834: SetFScaleDisable
A835: FontMetrics
A836: GetMaskTable
A837: MeasureText
A838: CalcMask
A839: SeedFill
A83A: ZoomWindow
A83B: TrackBox
A83C: TEGetOffset
A83D: TEDispatch
A83E: TEStyleNew
A83F: Long2Fix
A840: Fix2Long
A841: Fix2Frac
A842: Frac2Fix
A843: Fix2X
A844: X2Fix
A845: Frac2X
A846: X2Frac
A847: FracCos
A848: FracSin
A849: FracSqrt
A84A: FracMul
A84B: FracDiv
A84C: XCompactMem

A84D: FixDiv
A84E: GetItemCmd
A84F: SetItemCmd
A850: InitCursor
A851: SetCursor
A852: HideCursor
A853: ShowCursor
A855: ShieldCursor
A856: ObscureCursor
A858: BitAnd
A859: BitXor
A85A: BitNot
A85B: BitOr
A85C: BitShift
A85D: BitTst
A85E: BitSet
A85F: BitClr
A861: Random
A862: ForeColor
A863: BackColor
A864: ColorBit
A865: GetPixel
A866: StuffHex
A867: LongMul
A868: FixMul
A869: FixRatio
A86A: HiWord
A86B: LoWord
A86C: FixRound
A86D: InitPort
A86E: InitGraf
A86F: OpenPort
A870: LocalToGlobal
A871: GlobalToLocal
A872: GrafDevice
A873: SetPort
A874: GetPort
A875: SetPBits
A876: PortSize
A877: MovePortTo
A878: SetOrigin
A879: SetClip
A87A: GetClip
A87B: ClipRect
A87C: BackPat
A87D: ClosePort
A87E: AddPt
A87F: SubPt
A880: SetPt
A881: EqualPt
A882: StdText
A883: DrawChar
A884: DrawString
A885: DrawText
A886: TextWidth
A887: TextFont
A888: TextFace
A889: TextMode
A88A: TextSize
A88B: GetFontInfo
A88C: StringWidth
A88D: CharWidth
A88E: SpaceExtra
A890: StdLine
A891: LineTo
A892: Line
A893: MoveTo
A894: Move
A895: ShutDown
A896: HidePen
A897: ShowPen
A898: GetPenState

A899: SetPenState
A89A: GetPen
A89B: PenSize
A89C: PenMode
A89D: PenPat
A89E: PenNormal
A89F: Unimplemented
A8A0: StdRect
A8A1: FrameRect
A8A2: PaintRect
A8A3: EraseRect
A8A4: InverRect
A8A5: FillRect
A8A6: EqualRect
A8A7: SetRect
A8A8: OffsetRect
A8A9: InsetRect
A8AA: SectRect
A8AB: UnionRect
A8AC: Pt2Rect
A8AD: PtInRect
A8AE: EmptyRect
A8AF: StdRRect
A8B0: FrameRoundRect
A8B1: PaintRoundRect
A8B2: EraseRoundRect
A8B3: InverRoundRect
A8B4: FillRoundRect
A8B5: ScriptUtil
A8B6: StdOval
A8B7: FrameOval
A8B8: PaintOval
A8B9: EraseOval
A8BA: InvertOval
A8BB: FillOval
A8BC: SlopeFromAngle
A8BD: StdArc
A8BE: FrameArc
A8BF: PaintArc
A8C0: EraseArc
A8C1: InvertArc
A8C2: FillArc
A8C3: PtToAngle
A8C4: AngleFromSlope
A8C5: StdPoly
A8C6: FramePoly
A8C7: PaintPoly
A8C8: ErasePoly
A8C9: InvertPoly
A8CA: FillPoly
A8CB: OpenPoly
A8CC: ClosePgon
A8CD: KillPoly
A8CE: OffsetPoly
A8CF: PackBits
A8D0: UnpackBits
A8D1: StdRgn
A8D2: FrameRgn
A8D3: PaintRgn
A8D4: EraseRgn
A8D5: InverRgn
A8D6: FillRgn
A8D8: NewRgn
A8D9: DisposRgn
A8DA: OpenRgn
A8DB: CloseRgn
A8DC: CopyRgn
A8DD: SetEmptyRgn
A8DE: SetRecRgn
A8DF: RectRgn
A8E0: OffsetRgn
A8E1: InsetRgn
A8E2: EmptyRgn
A8E3: EqualRgn
A8E4: SectRgn
A8E5: UnionRgn
A8E6: DiffRgn
A8E7: XorRgn
A8E8: PtInRgn
A8E9: RectInRgn
A8EA: SetStdProcs
A8EB: StdBits
A8EC: CopyBits
A8ED: StdTxMeas
A8EE: StdGetPic
A8EF: ScrollRect
A8F0: StdPutPic
A8F1: StdComment
A8F2: PicComment
A8F3: OpenPicture
A8F4: ClosePicture
A8F5: KillPicture
A8F6: DrawPicture
A8F8: ScalePt
A8F9: MapPt
A8FA: MapRect
A8FB: MapRgn
A8FC: MapPoly
A8FD: Printing
A8FE: InitFonts
A8FF: GetFName
A900: GetFNum
A901: FMSwapFont
A902: RealFont
A903: SetFontLock
A904: DrawGrowIcon
A905: DragGrayRgn
A906: NewString
A907: SetString
A908: ShowHide
A909: CalcVis
A90A: CalcVBehind
A90B: ClipAbove
A90C: PaintOne
A90D: PaintBehind
A90E: SaveOld
A90F: DrawNew
A910: GetWMgrPort
A911: CheckUpdate
A912: InitWindows
A913: NewWindow
A914: DisposWindow
A915: ShowWindow
A916: HideWindow
A917: GetWRefCon
A918: SetWRefCon
A919: GetWTitle
A91A: SetWTitle
A91B: MoveWindow
A91C: HiliteWindow
A91D: SizeWindow
A91E: TrackGoAway
A91F: SelectWindow
A920: BringToFront
A921: SendBehind
A922: BeginUpdate
A923: EndUpdate
A924: FrontWindow
A925: DragWindow
A926: DragTheRgn
A927: InvalRgn
A928: InvalRect
A929: ValidRgn
A92A: ValidRect
A92B: GrowWindow
A92C: FindWindow
A92D: CloseWindow
A92E: SetWindowPic
A92F: GetWindowPic
A930: InitMenus
A931: NewMenu
A932: DisposMenu
A933: AppendMenu
A934: ClearMenuBar
A935: InsertMenu
A936: DeleteMenu
A937: DrawMenuBar
A938: HiliteMenu
A939: EnableItem
A93A: DisableItem
A93B: GetMenuBar
A93C: SetMenuBar
A93D: MenuSelect
A93E: MenuKey
A93F: GetItmIcon
A940: SetItmIcon
A941: GetItmStyle
A942: SetItmStyle
A943: GetItmMark
A944: SetItmMark
A945: CheckItem
A946: GetItem
A947: SetItem
A948: CalcMenuSize
A949: GetMHandle
A94A: SetMFlash
A94B: PlotIcon
A94C: FlashMenuBar
A94D: AddResMenu
A94E: PinRect
A94F: DeltaPoint
A950: CountMItems
A951: InsertResMenu
A952: DelMenuItem
A953: UpdtControl
A954: NewControl
A955: DisposControl
A956: KillControls
A957: ShowControl
A958: HideControl
A959: MoveControl
A95A: GetCRefCon
A95B: SetCRefCon
A95C: SizeControl
A95D: HiliteControl
A95E: GetCTitle
A95F: SetCTitle
A960: GetCtlValue
A961: GetMinCtl
A962: GetMaxCtl
A963: SetCtlValue
A964: SetMinCtl
A965: SetMaxCtl
A966: TestControl
A967: DragControl
A968: TrackControl
A969: DrawControls
A96A: GetCtlAction
A96B: SetCtlAction
A96C: FindControl
A96D: DrawControl
A96E: Dequeue
A96F: Enqueue
A970: GetNextEvent
A971: EventAvail
A972: GetMouse
A973: StillDown
A974: Button
A975: TickCount
A976: GetKeys
A977: WaitMouseUp
A978: UpdtDialog
A979: CouldDialog
A97A: FreeDialog
A97B: InitDialogs
A97C: GetNewDialog
A97D: NewDialog
A97E: SelIText
A97F: IsDialogEvent
A980: DialogSelect
A981: DrawDialog
A982: CloseDialog
A983: DisposDialog
A984: FindDItem
A985: Alert
A986: StopAlert
A987: NoteAlert
A988: CautionAlert
A989: CouldAlert
A98A: FreeAlert
A98B: ParamText
A98C: ErrorSound
A98D: GetDItem
A98E: SetDItem
A98F: SetIText
A990: GetIText
A991: ModalDialog
A992: DetachResource
A993: SetResPurge
A994: CurResFile
A995: InitResources
A996: RsrcZoneInit
A997: OpenResFile
A998: UseResFile
A999: UpdateResFile
A99A: CloseResFile
A99B: SetResLoad
A99C: CountResources
A99D: GetIndResource
A99E: CountTypes
A99F: GetIndType
A9A0: GetResource
A9A1: GetNamedResource
A9A2: LoadResource
A9A3: ReleaseResource
A9A4: HomeResFile
A9A5: SizeRsrc
A9A6: GetResAttrs
A9A7: SetResAttrs
A9A8: GetResInfo
A9A9: SetResInfo
A9AA: ChangedResource
A9AB: AddResource
A9AC: AddReference
A9AD: RmveResource
A9AE: RmveReference
A9AF: ResError
A9B0: WriteResource
A9B1: CreateResFile
A9B2: SystemEvent
A9B3: SystemClick
A9B4: SystemTask
A9B5: SystemMenu
A9B6: OpenDeskAcc
A9B7: CloseDeskAcc
A9B8: GetPattern
A9B9: GetCursor
A9BA: GetString

A9BB: GetIcon
A9BC: GetPicture
A9BD: GetNewWindow
A9BE: GetNewControl
A9BF: GetRMenu
A9C0: GetNewMBar
A9C1: UniqueID
A9C2: SysEdit
A9C3: KeyTrans
A9C4: OpenRFPerm
A9C5: RsrcMapEntry
A9C6: Secs2Date
A9C7: Date2Secs
A9C8: SysBeep
A9C9: SysError
A9CA: PutIcon
A9CB: TEGetText
A9CC: TEInit
A9CD: TEDispose
A9CE: TextBox
A9CF: TESetText
A9D0: TECalText
A9D1: TESetSelect
A9D2: TENew
A9D3: TEUpdate
A9D4: TEClick
A9D5: TECopy
A9D6: TECut
A9D7: TEDelete
A9D8: TEActivate
A9D9: TEDeactivate
A9DA: TEIdle
A9DB: TEPaste
A9DC: TEKey
A9DD: TEScroll
A9DE: TEInsert
A9DF: TESetJust
A9E0: Munger
A9E1: HandToHand
A9E2: PtrToXHand
A9E3: PtrToHand
A9E4: HandAndHand
A9E5: InitPack
A9E6: InitAllPacks
A9E7: Pack0
A9E8: Pack1
A9E9: Pack2
A9EA: Pack3
A9EB: FP68K
A9EC: Elems68K
A9ED: Pack6
A9EE: Pack7
A9EF: PtrAndHand
A9F0: LoadSeg
A9F1: UnloadSeg
A9F2: Launch
A9F3: Chain
A9F4: ExitToShell
A9F5: GetAppParms
A9F6: GetResFileAttrs
A9F7: setResFileAttrs
A9F8: MethodDispatch
A9F9: InfoScrap
A9FA: UnlodeScrap
A9FB: LodeScrap
A9FC: ZeroScrap
A9FD: GetScrap
A9FE: PutScrap
A9FF: Debugger
AA00: OpenCPort
AA01: InitCPort
AA02: CloseCPort
AA03: NewPixMap
AA04: DisposPixMap
AA05: CopyPixMap
AA06: SetCPortPix
AA07: NewPixPat
AA08: DisposPixPat
AA09: CopyPixPat
AA0A: PenPixPat
AA0B: BackPixPat
AA0C: GetPixPat
AA0D: MakeRGBPat
AA0E: FillCRect
AA0F: FillCOval
AA10: FillCRoundRect
AA11: FillCArc
AA12: FillCRgn
AA13: FillCPoly
AA14: RGBForeColor
AA15: RGBBackColor
AA16: SetCPixel
AA17: GetCPixel
AA18: GetCTable
AA19: GetForeColor
AA1A: GetBackColor
AA1B: GetCCursor
AA1C: SetCCursor
AA1D: AllocCursor
AA1E: GetCIcon
AA1F: PlotCIcon
AA20: OpenCPicture
AA21: OpColor
AA22: HiliteColor
AA23: CharExtra
AA24: DisposCTable
AA25: DisposCIcon
AA26: DisposCCursor
AA27: GetMaxDevice
AA28: GetCTSeed
AA29: GetDeviceList
AA2A: GetMainDevice
AA2B: GetNextDevice
AA2C: TestDeviceAttribute
AA2D: SetDeviceAttribute
AA2E: InitGDevice
AA2F: NewGDevice
AA30: DisposGDevice
AA31: SetGDevice
AA32: GetGDevice
AA33: Color2Index
AA34: Index2Color
AA35: InvertColor
AA36: RealColor
AA37: GetSubTable
AA38: UpdatePixMap
AA39: MakeITable
AA3A: AddSearch
AA3B: AddComp
AA3C: SetClientID
AA3D: ProtectEntry
AA3E: ReserveEntry
AA3F: SetEntry
AA40: QDError
AA41: SetWinColor
AA42: GetAuxWin
AA43: SetCtlColor
AA44: GetAuxCtl
AA45: NewCWindow
AA46: GetNewCWindow
AA47: SetDeskCPat
AA48: GetWMgrCPort
AA49: SaveEntries
AA4A: RestoreEntries
AA4B: NewCDialog
AA4C: DelSearch
AA4D: DelComp
AA4E: SetStdCProcs
AA4F: CalcCMask
AA50: SeedCFill
AA60: DelMCEntries
AA61: GetMCInfo
AA62: SetMCInfo
AA63: DispMCInfo
AA64: GetMCEntry
AA65: SetMCEntries
AA66: MenuChoice
AA90: InitPalettes
AA91: NewPalette
AA92: GetNewPalette
AA93: DisposePalette
AA94: ActivatePalette
AA95: SetPalette
AA96: GetPalette
AA97: PMForeColor
AA98: PMBackColor
AA99: AnimateEntry
AA9A: AnimatePalette
AA9B: GetEntryColor
AA9C: SetEntryColor
AA9D: GetEntryUsage
AA9E: SetEntryUsage
AA9F: CTab2Palette
AAA0: Palette2CTab
AAA1: CopyPalette
ABF8: StdOpcodeProc
ABFF: DebugStr

A000 Traps in Alphabetical Order

AA94: ActivatePalette
A07C: ADBOp
A07B: ADBReInit
AA3B: AddComp
A04E: AddDrive
A87E: AddPt
A9AC: AddReference
A94D: AddResMenu
A9AB: AddResource
AA3A: AddSearch
A0E5: AdjEOF
A985: Alert
A010: Allocate
AA1D: AllocCursor
A0D6: AllocNode
A8C4: AngleFromSlope
AA99: AnimateEntry
AA9A: AnimatePalette
A993: AppendMenu
A071: AttachVBL
A863: BackColor
A87C: BackPat
AA0B: BackPixPat
A0C8: BasicIO
A922: BeginUpdate
A858: BitAnd
A85F: BitClr
A85A: BitNot
A85B: BitOr
A85E: BitSet
A85C: BitShift
A85D: BitTst
A859: BitXor
A0DF: BlkAlloc
A0E0: BlkDealloc
A02E: BlockMove
A0F0: BMChk
A920: BringToFront
A0CC: BTClose
A0CD: BTDelete
A0CE: BTFlush
A0CF: BTGetRecord
A0D0: BTInsert
A0D1: BTOpen
A0D2: BTSearch
A0D3: BTUpdate
A974: Button
A0BD: CacheFlush
A0C7: CacheRdIP
A0C6: CacheWrIP
AA4F: CalcCMask
A838: CalcMask
A948: CalcMenuSize
A90A: CalcVBehind
A909: CalcVis
A988: CautionAlert
A9F3: Chain
A9AA: ChangedResource
A0E7: Char2Pixel
AA23: CharExtra
A88D: CharWidth
A945: CheckItem
A0FC: CheckLoad
A911: CheckUpdate
A0EE: CkExtFS
A934: ClearMenuBar
A90B: ClipAbove
A87B: ClipRect
A053: ClkNoMem
A001: Close
AA02: CloseCPort
A9B7: CloseDeskAcc
A982: CloseDialog
A8CC: ClosePgon
A8F4: ClosePicture
A87D: ClosePort
A99A: CloseResFile
A8DB: CloseRgn
A92D: CloseWindow
A0FA: CmpFrm
A03C: CmpString
A0DC: CMSetUp
AA33: Color2Index
A864: ColorBit

A04C: CompactMem
A004: Control
A8EC: CopyBits
A817: CopyMask
AAA1: CopyPalette
AA05: CopyPixMap
AA09: CopyPixPat
A8DC: CopyRgn
A989: CouldAlert
A979: CouldDialog
A80D: Count1Resources
A81C: Count1Types
A077: CountADBs
A950: CountMItems
A99C: CountResources
A99E: CountTypes
A008: Create
A9B1: CreateResFile
AA9F: CTab2Palette
A994: CurResFile
A9C7: Date2Secs
A0D9: DeallocFile
A9FF: Debugger
ABFF: DebugStr
A03B: Delay
AA4D: DelComp
A009: Delete
A936: DeleteMenu
AA60: DelMCEntries
A952: DelMenuItem
AA4C: DelSearch
A94F: DeltaPoint
A96E: Dequeue
A992: DetachResource
A980: DialogSelect
A8E6: DiffRgn
A93A: DisableItem
AA63: DispMCInfo
AA26: DisposCCursor
AA25: DisposCIcon
A955: DisposControl
AA24: DisposCTable
A983: DisposDialog
AA93: DisposePalette
AA30: DisposGDevice
A023: DisposHandle
A932: DisposMenu
AA04: DisposPixMap
AA08: DisposPixPat
A01F: DisposPtr
A8D9: DisposRgn
A914: DisposWindow
A0B9: Disptch
A0F7: DoEject
A072: DoVBLTask
A967: DragControl
A905: DragGrayRgn
A926: DragTheRgn
A925: DragWindow
A96D: Draw1Control
A883: DrawChar
A969: DrawControls
A981: DrawDialog
A904: DrawGrowIcon
A937: DrawMenuBar
A90F: DrawNew
A8F6: DrawPicture
A884: DrawString
A885: DrawText
A03D: DrvrInstall
A03E: DrvrRemove
A082: DTInstall

A0DE: DtrmV1
A0EF: DTrmV3
A017: Eject
A9EC: Elems68K
A02B: EmptyHandle
A8AE: EmptyRect
A8E2: EmptyRgn
A939: EnableItem
A923: EndUpdate
A96F: Enqueue
A881: EqualPt
A8A6: EqualRect
A8E3: EqualRgn
A8C0: EraseArc
A8B9: EraseOval
A8C8: ErasePoly
A8A3: EraseRect
A8D4: EraseRgn
A8B2: EraseRoundRect
A98C: ErrorSound
A971: EventAvail
A9F4: ExitToShell
A0D8: ExtBTFile
A0DA: ExtendFile
A0E1: FileOpen
A8C2: FillArc
AA11: FillCArc
AA0F: FillCOval
AA13: FillCPoly
AA0E: FillCRect
AA12: FillCRgn
AA10: FillCRoundRect
A8BB: FillOval
A8CA: FillPoly
A8A5: FillRect
A8D6: FillRgn
A8B4: FillRoundRect
A96C: FindControl
A984: FindDItem
A92C: FindWindow
A841: Fix2Frac
A840: Fix2Long
A843: Fix2X
A818: FixAtan2
A84D: FixDiv
A868: FixMul
A869: FixRatio
A86C: FixRound
A94C: FlashMenuBar
A0C0: FlushCache
A032: FlushEvents
A045: FlushFile
A013: FlushVol
A901: FMSwapFont
A0E3: FndFilName
A835: FontMetrics
A862: ForeColor
A9EB: FP68K
A842: Frac2Fix
A845: Frac2X
A847: FracCos
A84B: FracDiv
A84A: FracMul
A848: FracSin
A849: FracSqrt
A8BE: FrameArc
A8B7: FrameOval
A8C6: FramePoly
A8A1: FrameRect
A8D2: FrameRgn
A8B0: FrameRoundRect
A98A: FreeAlert

A97A: FreeDialog
A01C: FreeMem
A0D7: FreeNode
A924: FrontWindow
A80E: Get1IxResource
A80F: Get1IxType
A820: Get1NamedResource
A81F: Get1Resource
A079: GetADBInfo
A9F5: GetAppParms
AA44: GetAuxCtl
AA42: GetAuxWin
AA1A: GetBackColor
A0C1: GetBlock
AA1B: GetCCursor
AA1E: GetCIcon
A87A: GetClip
AA17: GetCPixel
A95A: GetCRefCon
AA18: GetCTable
A95E: GetCTitle
A96A: GetCtlAction
A960: GetCtlValue
AA28: GetCTSeed
A9B9: GetCursor
A809: GetCVariant
A07D: GetDefaultStartup
AA29: GetDeviceList
A98D: GetDItem
AA9B: GetEntryColor
AA9D: GetEntryUsage
A011: GetEOF
A00C: GetFileInfo
A8FF: GetFName
A900: GetFNum
A88B: GetFontInfo
AA19: GetForeColor
A018: GetFPos
AA32: GetGDevice
A025: GetHandleSize
A9BB: GetIcon
A078: GetIndADB
A99D: GetIndResource
A99F: GetIndType
A946: GetItem
A84E: GetItemCmd
A990: GetIText
A93F: GetItmIcon
A943: GetItmMark
A941: GetItmStyle
A976: GetKeys
AA2A: GetMainDevice
A836: GetMaskTable
A962: GetMaxCtl
AA27: GetMaxDevice
AA64: GetMCEntry
AA61: GetMCInfo
A93B: GetMenuBar
A949: GetMHandle
A961: GetMinCtl
A972: GetMouse
A9A1: GetNamedResource
A9BE: GetNewControl
AA46: GetNewCWindow
A97C: GetNewDialog
A9C0: GetNewMBar
AA92: GetNewPalette
A9BD: GetNewWindow
AA2B: GetNextDevice
A970: GetNextEvent
A0D4: GetNode
A083: GetOSDefault

A031: GetOSEvent
AA96: GetPalette
A9B8: GetPattern
A89A: GetPen
A898: GetPenState
A9BC: GetPicture
A865: GetPixel
AA0C: GetPixPat
A874: GetPort
A021: GetPtrSize
A9A6: GetResAttrs
A9F6: GetResFileAttrs
A9A8: GetResInfo
A9A0: GetResource
A9BF: GetRMenu
A9FD: GetScrap
A9BA: GetString
AA37: GetSubTable
A146: GetTrapAddress
A080: GetVideoDefault
A014: GetVol
A007: GetVolInfo
A92F: GetWindowPic
AA48: GetWMgrCPort
A910: GetWMgrPort
A917: GetWRefCon
A919: GetWTitle
A80A: GetWVariant
A11A: GetZone
A871: GlobalToLocal
A0B5: GoDriver
A872: GrafDevice
A92B: GrowWindow
A9E4: HandAndHand
A126: HandleZone
A9E1: HandToHand
A068: HClrRBit
A260: HFSDispatch
A069: HGetState
A958: HideControl
A852: HideCursor
A827: HideDItem
A896: HidePen
A916: HideWindow
AA22: HiliteColor
A95D: HiliteControl
A938: HiliteMenu
A0E8: HiliteText
A91C: HiliteWindow
A86A: HiWord
A029: HLock
A04A: HNoPurge
A9A4: HomeResFile
A049: HPurge
A067: HSetRBit
A06A: HSetState
A02A: HUnlock
A0BA: IAZInit
A0BB: IAZPostInit
AA34: Index2Color
A9F9: InfoScrap
A9E6: InitAllPacks
A02C: InitApplZone
AA01: InitCPort
A850: InitCursor
A97B: InitDialogs
A06D: InitEvents
A8FE: InitFonts
A06C: InitFS
AA2E: InitGDevice
A86E: InitGraf
A930: InitMenus

A9E5: InitPack
AA90: InitPalettes
A86D: InitPort
A808: InitProcMenu
A016: InitQueue
A995: InitResources
A03F: InitUtil
A912: InitWindows
A019: InitZone
A935: InsertMenu
A951: InsertResMenu
A8A9: InsetRect
A8E1: InsetRgn
A826: InsMenuItem
A058: InsTime
A928: InvalRect
A927: InvalRgn
A8A4: InverRect
A8D5: InverRgn
A8B3: InverRoundRect
A8C1: InvertArc
AA35: InvertColor
A8BA: InvertOval
A8C9: InvertPoly
A97F: IsDialogEvent
A9C3: KeyTrans
A956: KillControls
A006: KillIO
A8F5: KillPicture
A8CD: KillPoly
A9F2: Launch
A0BC: LaunchInit
A0BF: Lg2Phys
A892: Line
A891: LineTo
A9A2: LoadResource
A9F0: LoadSeg
A870: LocalToGlobal
A0F2: LocCRec
A9FB: LodeScrap
A83F: Long2Fix
A867: LongMul
A86B: LoWord
AA39: MakeITable
AA0D: MakeRGBPat
A0F4: MapFBlock
A8FC: MapPoly
A8F9: MapPt
A8FA: MapRect
A8FB: MapRgn
A0C2: MarkBlock
A063: MaxApplZone
A061: MaxBlock
A11D: MaxMem
A821: MaxSizeRsrc
A837: MeasureText
AA66: MenuChoice
A93E: MenuKey
A93D: MenuSelect
A9F8: MethodDispatch
A991: ModalDialog
A036: MoreMasters
A00F: MountVol
A894: Move
A959: MoveControl
A064: MoveHHi
A877: MovePortTo
A893: MoveTo
A91B: MoveWindow
A9E0: Munger
AA4B: NewCDialog
A954: NewControl

AA45: NewCWindow
A97D: NewDialog
A166: NewEmptyHandle
AA2F: NewGDevice
A122: NewHandle
A0FB: NewMap
A931: NewMenu
AA91: NewPalette
AA03: NewPixMap
AA07: NewPixPat
A11E: NewPtr
A8D8: NewRgn
A906: NewString
A913: NewWindow
A987: NoteAlert
A856: ObscureCursor
A035: Offline
A8CE: OffsetPoly
A8A8: OffsetRect
A8E0: OfsetRgn
AA21: OpColor
A000: Open
AA20: OpenCPicture
AA00: OpenCPort
A9B6: OpenDeskAcc
A8F3: OpenPicture
A8CB: OpenPoly
A86F: OpenPort
A997: OpenResFile
A00A: OpenRF
A9C4: OpenRFPerm
A8DA: OpenRgn
A030: OSEventAvail
A9E7: Pack0
A9E8: Pack1
A82C: Pack10
A82D: Pack11
A82E: Pack12
A82F: Pack13
A830: Pack14
A831: Pack15
A9E9: Pack2
A9EA: Pack3
A9ED: Pack6
A9EE: Pack7
A816: Pack8
A82B: Pack9
A8CF: PackBits
A8BF: PaintArc
A90D: PaintBehind
A90C: PaintOne
A8B8: PaintOval
A8C7: PaintPoly
A8A2: PaintRect
A8D3: PaintRgn
A8B1: PaintRoundRect
AAA0: Palette2CTab
A98B: ParamText
A89C: PenMode
A89E: PenNormal
A89D: PenPat
AA0A: PenPixPat
A89B: PenSize
A0E2: PermssnChk
A8F2: PicComment
A94E: PinRect
A0E6: Pixel2Char
AA1F: PlotCIcon
A94B: PlotIcon
AA98: PMBackColor
AA97: PMForeColor
A80B: PopUpMenuSelect

A876: PortSize
A02F: PostEvent
A05B: PowerOff
A05A: PrimeTime
A8FD: Printing
AA3D: ProtectEntry
A8AC: Pt2Rect
A8AD: PtInRect
A8E8: PtInRgn
A9EF: PtrAndHand
A9E3: PtrToHand
A9E2: PtrToXHand
A148: PtrZone
A8C3: PtToAngle
A04D: PurgeMem
A162: PurgeSpace
A9CA: PutIcon
A9FE: PutScrap
AA40: QDError
A861: Random
A0C9: RdBlocks
A04F: RDrvrInstall
A002: Read
A0F6: ReadBM
A039: ReadDateTime
A037: ReadParam
A051: ReadXPRam
AA36: RealColor
A902: RealFont
A027: ReallocHandle
A128: RecoverHandle
A8E9: RectInRgn
A8DF: RectRgn
A0C3: RelBlock
A9A3: ReleaseResource
A0D5: RelNode
A050: RelString
A00B: Rename
A9AF: ResError
AA3E: ReserveEntry
A040: ResrvMem
AA4A: RestoreEntries
A0E4: RfNCall
AA15: RGBBackColor
AA14: RGBForeColor
A80C: RGetResource
A9AE: RmveReference
A9AD: RmveResource
A059: RmvTime
A9C5: RsrcMapEntry
A996: RsrcZoneInit
A042: RstFilLock
AA49: SaveEntries
A90E: SaveOld
A8F8: ScalePt
A8B5: ScriptUtil
A833: ScrnBitMap
A8EF: ScrollRect
A815: SCSIDispatch
A9C6: Secs2Date
A8AA: SectRect
A8E4: SectRgn
AA50: SeedCFill
A839: SeedFill
A0F8: SegStack
A91F: SelectWindow
A97E: SelIText
A921: SendBehind
A07A: SetADBInfo
A057: SetAppBase
A02D: SetApplLimit
AA1C: SetCCursor

AA3C: SetClientID
A879: SetClip
AA16: SetCPixel
AA06: SetCPortPix
A95B: SetCRefCon
A95F: SetCTitle
A96B: SetCtlAction
AA43: SetCtlColor
A963: SetCtlValue
A851: SetCursor
A03A: SetDateTime
A07E: SetDefaultStartup
AA47: SetDeskCPat
AA2D: SetDeviceAttribute
A98E: SetDItem
A8DD: SetEmptyRgn
AA3F: SetEntry
AA9C: SetEntryColor
AA9E: SetEntryUsage
A012: SetEOF
A00D: SetFileInfo
A041: SetFilLock
A043: SetFilType
A903: SetFontLock
A044: SetFPos
A814: SetFractEnable
A834: SetFScaleDisable
AA31: SetGDevice
A04B: SetGrowZone
A024: SetHandleSize
A947: SetItem
A84F: SetItemCmd
A98F: SetIText
A940: SetItmIcon
A944: SetItmMark
A942: SetItmStyle
A965: SetMaxCtl
AA65: SetMCEntries
AA62: SetMCInfo
A93C: SetMenuBar
A94A: SetMFlash
A964: SetMinCtl
A878: SetOrigin
A084: SetOSDefault
AA95: SetPalette
A875: SetPBits
A899: SetPenState
A873: SetPort
A880: SetPt
A020: SetPtrSize
A8DE: SetRecRgn
A8A7: SetRect
A9A7: SetResAttrs
A9F7: SetResFileAttrs
A9A9: SetResInfo
A99B: SetResLoad
A993: SetResPurge
AA4E: SetStdCProcs
A8EA: SetStdProcs
A907: SetString
A047: SetTrapAddress
A0CB: SetUpTags
A081: SetVideoDefault
A015: SetVol
AA41: SetWinColor
A92E: SetWindowPic
A918: SetWRefCon
A91A: SetWTitle
A01B: SetZone
A855: ShieldCursor
A957: ShowControl
A853: ShowCursor

A828:	ShowDItem	A973:	StillDown	A812:	TEPinScroll	A953:	UpdtControl
A908:	ShowHide	A986:	StopAlert	A9DD:	TEScroll	A978:	UpdtDialog
A897:	ShowPen	A88C:	StringWidth	A811:	TESelView	A054:	UprString
A915:	ShowWindow	A055:	StripAddress	A9DF:	TESetJust	A998:	UseResFile
A895:	ShutDown	A866:	StuffHex	A9D1:	TESetSelect	A92A:	ValidRect
A95C:	SizeControl	A87F:	SubPt	A9CF:	TESetText	A929:	ValidRgn
A9A5:	SizeRsrc	A0F9:	SuperLoad	A966:	TestControl	A033:	VInstall
A91D:	SizeWindow	A05D:	SwapMMUMode	AA2C:	TestDeviceAttribute	A034:	VRemove
A8BC:	SlopeFromAngle	A0B7:	SyncWait	A83E:	TEStyleNew	A977:	WaitMouseUp
A06E:	SlotManager	A9C8:	SysBeep	A0FD:	TETrimMeasure	A0B6:	WaitUntil
A06F:	SlotVInstall	A9C2:	SysEdit	A9D3:	TEUpdate	A0CA:	WrBlocks
A070:	SlotVRemove	A090:	SysEnvirons	A9CE:	TextBox	A003:	Write
A802:	SndAddModifier	A9C9:	SysError	A888:	TextFace	A038:	WriteParam
A806:	SndControl	A9B3:	SystemClick	A887:	TextFont	A9B0:	WriteResource
A801:	SndDisposeChannel	A9B2:	SystemEvent	A889:	TextMode	A052:	WriteXPRam
A803:	SndDoCommand	A9B5:	SystemMenu	A88A:	TextSize	A844:	X2Fix
A804:	SndDoImmediate	A9B4:	SystemTask	A886:	TextWidth	A846:	X2Frac
A807:	SndNewChannel	A9D8:	TEActivate	A975:	TickCount	A84C:	XCompactMem
A805:	SndPlay	A813:	TEAutoView	A83B:	TrackBox	A823:	XDisposHandle
A0B8:	SoundDead	A9D0:	TECalText	A968:	TrackControl	A832:	XFlushEvents
A88E:	SpaceExtra	A9D4:	TEClick	A91E:	TrackGoAway	A0F5:	XFSearch
A065:	StackSpace	A9D5:	TECopy	A0C4:	TrashBlocks	A825:	XGetHandleSize
A005:	Status	A9D6:	TECut	A0C5:	TrashVBlks	A81A:	XGetZone
A8BD:	StdArc	A9D9:	TEDeactivate	A0F3:	TreeSearch	A829:	XHLock
A8EB:	StdBits	A9D7:	TEDelete	A0DB:	TruncateFile	A82A:	XHUnlock
A8F1:	StdComment	A83D:	TEDispatch	A0F1:	TstMod	A81D:	XMaxMem
A8EE:	StdGetPic	A9CD:	TEDispose	A89F:	Unimplemented	A819:	XMunger
A890:	StdLine	A0FF:	TEFindLine	A8AB:	UnionRect	A822:	XNewHandle
A8F8:	StdOpcodeProc	A0FE:	TEFindWord	A8E5:	UnionRgn	A81E:	XNewPtr
A8B6:	StdOval	A83C:	TEGetOffset	A810:	Unique1ID	A8E7:	XorRgn
A8C5:	StdPoly	A9CB:	TEGetText	A9C1:	UniqueID	A824:	XSetHandleSize
A8F0:	StdPutPic	A9DA:	TEIdle	A9F1:	UnloadSeg	A81B:	XSetZone
A8A0:	StdRect	A9CC:	TEInit	A9FA:	UnlodeScrap	A9FC:	ZeroScrap
A8D1:	StdRgn	A9DE:	TEInsert	A00E:	UnmountVol		
A8AF:	StdRRect	A9DC:	TEKey	A8D0:	UnpackBits		
A882:	StdText	A9D2:	TENew	AA38:	UpdatePixMap		
A8ED:	StdTxMeas	A9DB:	TEPaste	A999:	UpdateResFile		

Labels Built into the User Area in Numerical Order

8:	BusError	13E:	PollProc	1E0:	IWM	236:	JSeek
C:	AddrError	142:	DskErr	1E4:	scratch20	23A:	JSetUpPoll
10:	Illegal	144:	SysEvtMask	1F8:	SysParam	23E:	JRecal
14:	ZeroDivide	146:	SysEvtBuf	1F8:	SPValid	242:	JControl
18:	ChkError	14A:	EventQueue	1F9:	SPATalkA	246:	JWakeUp
1C:	TrapVError	154:	EvtBufCnt	1FA:	SPATalkB	24A:	JReSeek
20:	PrivilegeViolation	156:	RndSeed	1FB:	SPConfig	24E:	JMakeSpdTbl
24:	Trace	15A:	SysVersion	200:	SPAlarm	252:	JAdrDisk
28:	Line1010	15C:	SEvtEnb	204:	SPFont	256:	JSetSpeed
100:	MonkeyLives	15D:	DSWndUpdate	206:	SPKbd	25A:	NiblTbl
102:	ScrVRes	15E:	FontFlag	207:	SPPrint	25E:	FlEventMask
104:	ScrHRes	15F:	IntFlag	208:	SPVolCtl	260:	SdVolume
106:	ScreenRow	160:	VBLQueue	209:	SPClikCaret	261:	SdEnable
108:	MemTop	16A:	Ticks	20A:	SPMisc1	262:	SoundPtr
10C:	BufPtr	16E:	MBTicks	20B:	SPMisc2	266:	SoundBase
110:	StkLowPt	172:	MBState	20C:	Time	26A:	SoundVBL
114:	HeapEnd	173:	Tocks	210:	BootDrive	27A:	SoundDCE
118:	TheZone	174:	KeyMap	212:	JShell	27E:	SoundActive
11C:	UTableBase	17C:	KeypadMap	214:	SFSaveDisk	27F:	SoundLevel
120:	MacJmp	184:	KeyLast	216:	KbdVars	280:	CurPitch
124:	DskRtnAdr	186:	KeyTime	21A:	JKybdTask	282:	SwitcherData
128:	PollRtnAddr	18A:	KeyRepTime	21E:	KbdType	286:	MemToSwitch
12C:	DskVerify	18E:	KeyThresh	21F:	AlarmState	28A:	RSDHndl
12D:	LoadTrap	190:	KeyRepThresh	220:	MemError	28E:	ROM85
12E:	MmInOK	192:	Lvl1DT	222:	DiskVars	290:	PortAUse
12F:	CPUFlag	1B2:	Lvl2DT	222:	JFigTrkSpd	291:	PortBUse
130:	ApplLimit	1D2:	UnitNtryCnt	226:	JDiskPrime	292:	ScreenVars
134:	SonyVars	1D4:	VIA	22A:	JRdAddr	29A:	JGNEFilter
138:	PWMValue	1D8:	SCCRd	22E:	JRdData	29E:	Key1Trans
13A:	PollStack	1DC:	SCCWr	232:	JWrData	2A2:	Key2Trans

2A6: SysZone
2AA: ApplZone
2AE: ROMBase
2B2: RAMBase
2B6: BasicGlob
2BA: DSAlertTab
2BE: ExtStsDT
2CE: SCCASts
2CF: SCCBSts
2D0: SerialVars
2D8: ABusVars
2DC: ABusDCE
2E0: FinderName
2F0: DoubleTime
2F4: CaretTime
2F8: ScrDmpEnb
2F9: ScrDmpType
2FA: TagData
2FC: BufTgFNum
300: BufTgFFlg
302: BufTgFBkNum
304: BufTgDate
308: DrvQHdr
312: PWMBuf2
316: MPWPtr
31A: Lo3Bytes
31E: MinStack
322: DefltStack
326: MMDefFlags
328: GZRootHnd
32C: GZRootPtr
330: GZMoveHnd
334: DSDrawProc
338: EjectNotify
33C: IAZNotify
340: CurDB
340: CkdDB
342: NxtDB
342: FSCallAsync
344: MaxDB
346: FlushOnly
347: RegRsrc
348: FLckUnlck
349: FrcSync
34A: NewMount
34B: NoEject
34C: DrMstrBlk
34E: FCBSPtr
352: DefVCBPtr
356: VCBQHdr
360: FSQHdr
360: FSBusy
362: FSQHead
366: FSQTail
36A: RgSvArea
36A: HFSVars
36A: HFSStkLen
36E: HFSStkPtr
372: WDCBsPtr
376: HFSFlags
377: CacheFlag
378: SysBMCPtr
37C: SysVolCPtr
380: SysCtlCPtr
384: DefVRefNum
386: PMSPPtr
38A: HFSTagData
392: HFSDSErr
394: CacheVars
39A: CurDirStore
39C: CacheCom
3B4: FmtDefaults

3A2: ErrCode
3A4: Params
3D6: FSTemp8
3DE: FSTemp4
3DE: FSIOErr
3E2: FSQueueHook
3E6: ExtFSHook
3EA: DskSwtchHook
3EE: ReqstVol
3F2: ToExtFS
3F6: FSFCBLen
3F8: DSAlertRect
400: DispatchTab
800: JHideCursor
804: JShowCursor
808: JShieldCursor
80C: JScrnAddr
810: JScrnSize
814: JInitCrsr
818: JSetCrsr
81C: JCrsrObscure
820: JUpdateProc
824: ScrnBase
828: MTemp
82C: RawMouse
830: Mouse
834: CrsrPin
83C: CrsrRect
844: TheCrsr
888: CrsrAddr
88C: CrsrSave
88C: JAllocCrsr
890: JSetCCrsr
894: JOpcodeProc
898: CrsrBase
89C: CrsrDevice
8A0: SrcDevice
8A4: MainDevice
8A8: DeviceList
8AC: CrsrRow
8CC: CrsrVis
8CD: CrsrBusy
8CE: CrsrNew
8CF: CrsrCouple
8D0: CrsrState
8D2: CrsrObscure
8D3: CrsrScale
8D6: MouseMask
8DA: MouseOffset
8DE: JournalFlag
8E0: JSwapFont
8E4: WidthListHand
8E4: JFontInfo
8E8: JournalRef
8EC: CrsrThresh
8EE: JCrsrTask
8F2: WWExist
8F3: QDExist
8F4: JFetch
8F8: JStash
8FC: JIODone
900: LoadVars
900: CurApRefnum
902: LaunchFlag
903: FondState
904: CurrentA5
908: CurStack
910: CurApName
930: SaveSegHandle
934: CurJTOffset
936: CurPageOption
93A: LoaderPBlock

944: PrintVars
944: PrintErr
946: ChooserBits
954: CoreEdit
960: ScrapVars
960: scrapSize
964: scrapHandle
968: scrapCount
96A: scrapState
96C: scrapName
970: ScrapTag
980: RomFont0
984: ApFontID
986: SaveFondFlags
986: GotStrike
987: FMDefaultSize
988. CurFMInput
988: CurFMFamily
98A: CurFMSize
98C: CurFMFace
98D: CurFMNeedBits
98E: CurFMDevice
990: CurFMNumer
994: CurFMDenom
998: FOutRec
998: FOutError
99A: FOutFontHandle
99E: FOutBold
99F: FOutItalic
9A0: FOutULOffset
9A1: FOutULShadow
9A2: FOutULThick
9A3: FOutShadow
9A4: FOutExtra
9A5: FOutAscent
9A6: FOutDescent
9A7: FOutWidMax
9A8: FOutLeading
9A9: FOutUnused
9AA: FOutNumber
9AE: FOutDenom
9B2: FMDotsPerInch
9B6: FMStyleTab
9CE: ToolScratch
9D6: WindowList
9DA: SaveUpdate
9DC: PaintWhite
9DE: WMgrPort
9E2: DeskPort
9E6: OldStructure
9EA: OldContent
9EE: GrayRgn
9F2: SaveVisRgn
9F6: DragHook
9FA: scratch8
A02: OneOne
A06: MinusOne
A0A: AtMenuBottom
A0A: TopMenuItem
A0E: IconBitmap
A1C: MenuList
A20: MBarEnable
A22: CurDeKind
A24: MenuFlash
A26: TheMenu
A28: SavedHandle
A2C: MBarHook
A30: MenuHook
A34: DragPattern
A3C: DeskPattern
A44: DragFlag
A46: CurDragAction

A4A: FPState
A50: TopMapHndl
A54: SysMapHndl
A58: SysMap
A5A: CurMap
A5C: ResReadOnly
A5E: ResLoad
A60: ResErr
A62: TaskLock
A63: FScaleDisable
A64: CurActivate
A68: CurDeactive
A6C: DeskHook
A70: TEDoText
A74: TEMeasal
A78: ApplScratch
A84: GhostWindow
A88: CloseOrnHook
A8C: ResumeProc
A90: SaveProc
A94: SaveSP
A98: ANumber
A9A: ACount
A9C: DABeeper
AA0: DAStrings
AB0: TEScrpLength
AB4: TEScrpHandle
AB8: AppPacks
AD8: SysResName
AEC: AppParmHandle
AF0: DSErrCode
AF2: ResErrProc
AF6: TEWdBreak
AFA: DlgFont
B00: TrapAgain
B04: KeyMVars
B06: ROMMapHndl
B0A: PWMBuf1
B0E: BootMask
B10: WidthPtr
B14: ATalkHk1
B18: ATalkHk2
B22: HWCfgFlags
B26: TopMenuItem
B28: AtMenuBottom
B2A: WidthTabHandle
B2E: SCSIDrvrs
B30: TimeVars
B34: BtDskRfn
B36: BootTmp8
B3F: TlArbitrate
B40: JDiskSel
B44: JSendCmd
B48: JDCDReset
B4C: LastSPExtra
B50: BNMQHd
B54: MenuDisable
B58: MBDFHndl
B5C: MBSaveLoc
B9E: ROMMapInsert
B9F: TmpResLoad
BA0: IntlSpec
BA0: IntlSpec
BA5: WordRedraw
BA6: SysFontFam
BA8: SysFontSize
BAA: MBarHeight
BAC: TESysJust
BAE: HiHeapMark
BB2: SegHiEnable
BB3: FDevDisable
BC0: NewUnused

BC2:	LastFOND	C2D:	VidType	CF8:	ADBBase	D42:	FMExist
BC6:	FONDID	C2E:	VidMode	CFC:	WarmStart	D50:	MenuCInfo
BC8:	App2Packs	C2F:	SCSIPoll	D00:	TimeDBRA	D60:	ChunkyDepth
BE8:	MAErrProc	C30:	SEVarBase	D02:	TimeSCCDB	D62:	CrsrPtr
BEC:	MASuperTap	CB0:	MMUFlags	D04:	SlotQDT	D66:	PortList
BF4:	FractEnable	CB1:	MMUType	D08:	SlotPrTbl	D6A:	MickeyBytes
BF5:	UsedFWidths	CB2:	MMUMode	D0C:	SlotVBLQ	D6E:	QDErr
BF6:	FScaleHFact	CB4:	MMUTbl	D10:	ScrnVBLPtr	D70:	VIA2DT
BFA:	FScaleVFact	CB8:	MMUTblSize	D14:	SlotTicks	D90:	sInitFlags
C00:	SCSIBase	CBC:	sInfoPtr	D1C:	AGBHandle	D92:	DTQueue
C04:	SCSIDMA	CC0:	ASCBase	D20:	TableSeed	D94:	DTskQHdr
C08:	SCSIHsk	CC4:	SMGlobals	D24:	sRsrcTblPtr	D98:	DTskQTail
C0C:	SCSIGlobals	CC8:	theGDevice	D28:	JVBLTask	D9C:	JDTInstall
C10:	RGBBlack	CCC:	CQDGlobals	D2C:	WMgrCPort		
C16:	RGBWhite	CD0:	AuxWinHead	D30:	VertRRate		
C20:	RowBits	CD4:	AuxCtlHead	D32:	SynListHandle		
C22:	ColLines	CD8:	DeskCPat	D36:	LastFore		
C24:	ScreenBytes	CDC:	LastBinPat	D3E:	LastMode		
C2C:	NMIFlag	CE4:	DeskPatEnable	D40:	LastDepth		

Labels Built Into the User Area in Alphabetical Order

2DC:	ABusDCE	898:	CrsrBase	222:	DiskVars	99F:	FOutItalic
2D8:	ABusVars	8CD:	CrsrBusy	400:	DispatchTab	9A8:	FOutLeading
A9A:	ACount	8CF:	CrsrCouple	AFA:	DlgFont	9AA:	FOutNumeer
CF8:	ADBBase	89C:	CrsrDevice	2F0:	DoubleTime	998:	FOutRec
C:	AddrError	8CE:	CrsrNew	A44:	DragFlag	9A3:	FOutShadow
D1C:	AGBHandle	8D2:	CrsrObscure	9F6:	DragHook	9A0:	FOutULOffset
21F:	AlarmState	834:	CrsrPin	A34:	DragPattern	9A1:	FOutULShadow
A98:	ANumber	D62:	CrsrPtr	34C:	DrMstrBlk	9A2:	FOutULThick
984:	ApFontID	83C:	CrsrRect	308:	DrvQHdr	9A9:	FOutUnused
BC8:	App2Packs	8AC:	CrsrRow	3F8:	DSAlertRect	9A7:	FOutWidMax
130:	ApplLimit	88C:	CrsrSave	2BA:	DSAlertTab	A4A:	FPState
A78:	ApplScratch	8D3:	CrsrScale	334:	DSDrawProc	BF4:	FractEnable
2AA:	ApplZone	8D0:	CrsrState	AF0:	DSErrCode	349:	FrcSync
AB8:	AppPacks	8EC:	CrsrThresh	142:	DskErr	360:	FSBusy
ABC:	AppParmHandle	8CC:	CrsrVis	124:	DskRtnAdr	A63:	FScaleDisable
CC0:	ASCBase	A64:	CurActivate	3EA:	DskSwtchHook	BF6:	FScaleHFact
B14:	ATalkHk1	910:	CurApName	12C:	DskVerify	BFA:	FScaleVFact
B18:	ATalkHk2	900:	CurApRefnum	15D:	DSWndUpdate	342:	FSCallAsync
B28:	AtMenuBottom	340:	CurDB	D92:	DTQueue	3F6:	FSFCBLen
A0A:	AtMenuBottom	A68:	CurDeactive	D94:	DTskQHdr	3DE:	FSIOErr
CD4:	AuxCtlHead	A22:	CurDeKind	D98:	DTskQTail	360:	FSQHdr
CD0:	AuxWinHead	398:	CurDirStore	338:	EjectNotify	362:	FSQHead
2B6:	BasicGlob	A46:	CurDragAction	3A2:	ErCode	366:	FSQTail
B50:	BNMQHd	994:	CurFMDenom	14A:	EventQueue	3E2:	FSQueueHook
210:	BootDrive	98E:	CurFMDevice	154:	EvtBufCnt	3DE:	FSTemp4
B0E:	BootMask	98C:	CurFMFace	3E6:	ExtFSHook	3D6:	FSTemp8
B36:	BootTmp8	988:	CurFMFamily	2BE:	ExtStsDT	A84:	GhostWindow
B34:	BtDskRfn	988:	CurFMInput	34E:	FCBSPtr	986:	GotStrike
10C:	BufPtr	98D:	CurFMNeedBits	BB3:	FDevDisable	9EE:	GrayRgn
304:	BufTgDate	990:	CurFMNumer	2E0:	FinderNume	330:	GZMoveHnd
302:	BufTgFBkNum	98A:	CurFMSize	348:	FLckUnlck	328:	GZRootHnd
300:	BufTgFFlg	934:	CurJTOffset	25E:	FlEventMask	32C:	GZRootPtr
2FC:	BufTgFNum	A5A:	CurMap	346:	FlushOnly	114:	HeapEnd
8:	BusError	936:	CurPageOption	987:	FMDefaultSize	392:	HFSDSErr
39C:	CacheCom	280:	CurPitch	9B2:	FMDotsPerInch	376:	HFSFlags
377:	CacheFlag	904:	CurrentA5	D42:	FMExist	36A:	HFSStkLen
394:	CacheVars	908:	CurStack	9B6:	FMStyleTab	36E:	HFSStkPtr
2F4:	CaretTime	A9C:	DABeeper	39E:	FmtDefaults	38A:	HFSTagData
18:	ChkError	AA0:	DAStrings	BC6:	FONDID	36A:	HFSVars
946:	ChooserBits	322:	DefltStack	903:	FondState	BAE:	HiHeapMark
D60:	ChunkyDepth	352:	DefVCBPtr	15E:	FontFlag	B22:	HWCfgFlags
340:	CkdDB	384:	DefVRefNum	9A5:	FOutAscent	33C:	IAZNotify
A88:	CloseOrnHook	CD8:	DeskCPat	99E:	FOutBold	A0E:	IconBitmap
C22:	ColLines	A6C:	DeskHook	9AE:	FOutDenom	10:	Illegal
954:	CoreEdit	CE4:	DeskPatEnable	9A6:	FOutDescent	15F:	IntFlag
12F:	CPUFlag	A3C:	DeskPattern	998:	FOutError	BA0:	IntlSpec
CCC:	CQDGlobals	9E2:	DeskPort	9A4:	FOutExtra	BA0:	IntlSpec
888:	CrsrAddr	8A8:	DeviceList	99A:	FOutFontHandle	1E0:	IWM

252: JAdrDisk
88C: JAllocCrsr
242: JControl
81C: JCrsrObscure
8EE: JCrsrTask
B48: JDCDReset
226: JDiskPrime
B40: JDiskSel
D9C: JDTInstall
8F4: JFetch
222: JFigTrkSpd
8E4: JFontInfo
29A: JGNEFilter
800: JHideCursor
814: JInitCrsr
8FC: JIODone
21A: JKybdTask
24E: JMakeSpdTbl
894: JOpcodeProc
8DE: JournalFlag
8E8: JournalRef
22A: JRdAddr
22E: JRdData
23E: JRecal
24A: JReSeek
80C: JScrnAddr
810: JScrnSize
236: JSeek
B44: JSendCmd
890: JSetCCrsr
818: JSetCrsr
256: JSetSpeed
23A: JSetUpPoll
212: JShell
808: JShieldCursor
804: JShowCursor
8F8: JStash
8E0: JSwapFont
820: JUpdateProc
D28: JVBLTask
246: JWakeUp
232: JWrData
21E: KbdType
216: KbdVars
29E: Key1Trans
2A2: Key2Trans
184: KeyLast
174: KeyMap
B04: KeyMVars
17C: KeypadMap
190: KeyRepThresh
18A: KeyRepTime
18E: KeyThresh
186: KeyTime
CDC: LastBinPat
D40: LastDepth
BC2: LastFOND
D36: LastFore
D3E: LastMode
B4C: LastSPExtra
902: LaunchFlag
28: Line1010
31A: Lo3Bytes
93A: LoaderPBlock
12D: LoadTrap
900: LoadVars
192: Lvl1DT
1B2: Lvl2DT
130: MacJmp
B80: MAErrProc
8A4: MainDevice
B8C: MASuperTap
344: MaxDB
A20: MBarEnable
BAA: MBarHeight
A2C: MBarHook
B58: MBDFHndl
B5C: MBSaveLoc
172: MBState
16E: MBTicks
220: MemError
108: MemTop
286: MemToSwitch
D50: MenuCInfo
B54: MenuDisable
A24: MenuFlash
A30: MenuHook
A1C: MenuList
D6A: MickeyBytes
31E: MinStack
A06: MinusOne
326: MMDefFlags
12E: MmInOK
CB0: MMUFlags
CB2: MMUMode
CB4: MMUTbl
CB8: MMUTblSize
CB1: MMUType
100: MonkeyLives
830: Mouse
8D6: MouseMask
8DA: MouseOffset
316: MPWPtr
828: MTemp
34A: NewMount
BC0: NewUnused
25A: NiblTbl
C2C: NMIFlag
34B: NoEject
342: NxtDB
9EA: OldContent
9E6: OldStructure
A02: OneOne
9DC: PaintWhite
3A4: Params
386: PMSPPtr
13E: PollProc
128: PollRtnAddr
13A: PollStack
290: PortAUse
291: PortBUse
D66: PortList
944: PrintErr
944: PrintVars
20: PrivilegeViolation
B0A: PWMBuf1
312: PWMBuf2
138: PWMValue
D6E: QDErr
8F3: QDExist
2B2: RAMBase
82C: RawMouse
347: RegRsrc
3EE: ReqstVol
A60: ResErr
AF2: ResErrProc
A5E: ResLoad
A5C: ResReadOnly
A8C: ResumeProc
C10: RGBBlack
C16: RGBWhite
36A: RgSvArea
156: RndSeed
28E: ROM85
2AE: ROMBase
980: RomFont0
B06: ROMMapHndl
B9E: ROMMapInsert
C20: RowBits
28A: RSDHndl
A28: SavedHandle
986: SaveFondFlags
A90: SaveProc
930: SaveSegHandle
A94: SaveSP
9DA: SaveUpdate
9F2: SaveVisRgn
2CE: SCCASts
2CF: SCCBSts
1D8: SCCRd
1DC: SCCWr
968: scrapCount
964: scrapHandle
96C: scrapName
960: scrapSize
96A: scrapState
970: ScrapTag
960: ScrapVars
1E4: scratch20
9FA: scratch8
2F8: ScrDmpEnb
2F9: ScrDmpType
C24: ScreenBytes
106: ScreenRow
292: ScreenVars
104: ScrHRes
824: ScrnBase
D10: ScrnVBLPtr
102: ScrVRes
C00: SCSIBase
C04: SCSIDMA
B2E: SCSIDrvrs
C0C: SCSIGlobals
C08: SCSIHsk
C2F: SCSIPoll
261: SdEnable
260: SdVolume
BB2: SegHiEnable
2D0: SerialVars
C30: SEVarBase
15C: SEvtEnb
214: SFSaveDisk
CBC: sInfoPtr
D90: sInitFlags
D08: SlotPrTbl
D04: SlotQDT
D14: SlotTicks
D0C: SlotVBLQ
CC4: SMGlobals
134: SonyVars
27E: SoundActive
266: SoundBase
27A: SoundDCE
27F: SoundLevel
262: SoundPtr
26A: SoundVBL
200: SPAlarm
1F9: SPATalkA
1FA: SPATalkB
209: SPClikCaret
1FB: SPConfig
204: SPFont
206: SPKbd
20A: SPMisc1
20B: SPMisc2
207: SPPrint
1F8: SPValid
208: SPVolCtl
8A0: SrcDevice
D24: sRsrcTblPtr
110: StkLowPt
282: SwitcherData
D32: SynListHandle
378: SysBMCPtr
380: SysCtlCPtr
146: SysEvtBuf
144: SysEvtMask
BA6: SysFontFam
BA8: SysFontSize
A58: SysMap
A54: SysMapHndl
1F8: SysParam
AD8: SysResName
15A: SysVersion
37C: SysVolCPtr
2A6: SysZone
B3F: T1Arbitrate
D20: TableSeed
2FA: TagData
A62: TaskLock
A70: TEDoText
A74: TERecal
AB4: TEScrpHandle
AB0: TEScrpLength
BAC: TESysJust
AF6: TEWdBreak
844: TheCrsr
CC8: theGDevice
A26: TheMenu
118: TheZone
16A: Ticks
20C: Time
D00: TimeDBRA
D02: TimeSCCDB
B30: TimeVars
B9F: TmpResLoad
173: Tocks
3F2: ToExtFS
9CE: ToolScratch
A50: TopMapHndl
B26: TopMenuItem
A0A: TopMenuItem
24: Trace
B00: TrapAgain
1C: TrapVError
1D2: UnitNtryCnt
BF5: UsedFWidths
11C: UTableBase
160: VBLQueue
356: VCBQHdr
D30: VertRRate
1D4: VIA
D70: VIA2DT
C2E: VidMode
C2D: VidType
CFC: WarmStart
372: WDCBsPtr
8E4: WidthListHand
B10: WidthPtr
B2A: WidthTabHandle
9D6: WindowList
D2C: WMgrCPort
9DE: WMgrPort
BA5: WordRedraw
8F2: WWExist
14: ZeroDivide

Appendix B—TMON Warning and Error Messages

The following are the messages that will show up in the Monitor's alert window. Each one is accompanied by a short explanation of when to expect that message.

WARNING! The monitor has been damaged. Be very careful!

The internal checksum routine that examines the Monitor detected a change in the Monitor's code.

Illegal instruction at $*XXXXXX*.
Divide by zero at $*XXXXXX*.
Divide by zero before $*XXXXXX*.
CHK exception at $*XXXXXX*.
CHK exception before $*XXXXXX*.
TRAPcc exception at $*XXXXXX*.
TRAPV exception before $*XXXXXX*.
Privilege violation at $*XXXXXX*.
Trace interrupt at $*XXXXXX*.
Address error. PC=$*XXXXXX*.
Trap #$*Y* at $*ZZZZZZ*.
Trap #$*Y* before $*ZZZZZZ*.

One of the above messages will be shown when a 680x0 instruction exception occurs. It should be noted that a trace interrupt will occur whenever you use single step or trace. See the **Exception Handling** section of the *Technical Reference* for more information.

Unassigned interrupt #$*XXX* (format $*Y*) at $*ZZZZZZ*.
Unassigned interrupt at $*XXXXXX*.
Level *X* interrupt at $*YYYYYY*.

This message will be shown when a hardware interrupt has been detected by TMON. See the **Exception Handling** section of the *Technical Reference* for more information.

The A000 trap or subroutine has returned.

This message will be shown after control returns to TMON from a GoSub or a Step through an A000 Trap.

Breakpoint at $*XXXXXX*.

This message will be shown when a user defined TMON break point is encountered.

System error #$*XXXX* at $*YYYYYY*.

An error was reported to the system via the _SysError trap.

Bus error. PC=$*XXXXXX*.
Access address: $*XXXXXX* (user data), Instruction: yes, mode: read.
Access address: $*XXXXXX* (user program), Instruction: yes, mode: write.
Access address: $*XXXXXX* (supervisor data), Instruction: no, mode: write.
Access address: $*XXXXXX* (supervisor program), Instruction: no, mode: read.
Access address: $*XXXXXX* (exception), Instruction: yes, mode: read.
Access address: $*XXXXXX* (illegal), Instruction: no, mode: write.

A bus error has occurred. The additional data defines the type of bus cycle present on the bus at the time of the error. The 68020 bus error exception is more complicated and this information will not be shown. See the **Exception Handling** section of the *Technical Reference* for more information.

Welcome to Monitor version 2.8
Written by Waldemar Horwat.

This message will be displayed upon initial entry to TMON.

No more windows can be created.

There is a limit of 19 total windows that TMON can open at any one time. You have tried to open another window after reaching this limit.

Mouse antifreeze completed. Be careful with the serial ports.

This message is displayed whenever you use the mouse unfreeze command. See the **Mouse Unfreeze** section of the *Technical Reference* for further information.

I don't want to execute the next instruction.

Using Step or GoSub on the next instruction will make it impossible for TMON to regain control. To continue execution, you can use Exit or Trace. The only exception to this is when the next instruction is a _SysError, in which case you cannot continue execution from that address. See the **Exit, GoSub, Step, and Trace** section of the *Technical Reference* for more information.

Appendix C—TMON Hints and Tips

Many people have been using TMON for some time now. A great deal of wisdom has been accumulated regarding how to use TMON to find and recover from bugs. Some ways in which TMON is useful are obvious; others are not so obvious. I want to use this appendix to provide some ideas as to what you can do with TMON that may not be immediately obvious (although many of these hints fall into the "why didn't *I* think of that?" category). We owe a special thanks to Apple Technical Support, who have written several of these tips up in various Tech Notes.

MPW Tools Gone Amok

If you are an MPW user, chances are that you've gotten into a situation where a tool was running and you wanted to stop it dead in its tracks. Perhaps for one reason or another hitting ⌘-. doesn't work, or the tool has crashed into TMON and you need to recover. If you have TMON installed, you can stop the tool very quickly by holding down the Option key and pressing the interrupt switch. Press ⌘R to open the "Regs" window. Press the Tab key to put the cursor by the PC field. Now type "STOPTOOL" including the quotes, and press the Return key. Now press ⌘E to exit. You've changed the PC to point to an MPW routine that stops tools dead in their tracks, and all should be well.

Catching a Failure to Check Common Errors

One problem with some Macintosh programs and desk accessories is that they neglect to check the results of memory manager or resource manager operations. This can have disastrous consequences. One way that TMON can help to catch bugs like this is by using TMON's checksumming capability to see if either of two important low RAM globals has changed. You can do a checksum from "MemErr" to "MemErr"+1 or from "ResErr" to "ResErr"+1 on all traps, and anytime that the bytes within the range change, TMON will be entered on the next trap. It's a good idea to use this in conjunction with Trap Record so that you can see a history of what trap(s) may have generated the error that TMON caught.

Looking at Other Heap Zones

Some applications and programming environments create at least one heap zone besides the Macintosh's normal two, the application zone and the system zone. In order to enable TMON to see one of these other zones, assuming that you know its starting address, just open a dump window at "ApplZone" (don't forget the quotes!) and make a note of the address there (a good way to do this is to open a "Num" window and type @"ApplZone"). Position the cursor at the beginning of the address, and enter the address of the zone that you wish to see. Press the Return key. Now open the heap window. It will say that it's showing the application zone, but it's really showing the custom zone. Don't forget to change "ApplZone" back to the application zone's address when you are done!

Alternate _ExitToShell

The user area's function to ExitToShell is smart; it tries to close all open files and resource files before doing the _ExitToShell. However, there are times when the process of trying to close all files doesn't work. The File Manager may be screwed up somehow. If you want to try to _ExitToShell without all the extra overhead, just open the "Regs" window and change the PC to !_ExitToShell. Now use the "Exit" function. Since the PC points to the address of the _ExitToShell routine, you should go back to the Finder (or whatever the shell application is). However, there may be open files and so forth lying around, so it is a good idea to immediately shut down your system and start over. Note that you can cheat and launch anything by editing the string at FinderName.

Restricting Trap Intercepting Functions to the Application Zone

This sounds a lot tougher than it actually is. There are times when you want to perform some trap-related function such as Trap intercept, but you only want it to take effect on traps in your program, i.e. in the ap-

plication zone. You can restrict trap intercepting functions to certain PC ranges. There's an easy way to restrict them to the application heap: using the indirection operator (described in the Technical Reference Manual) and TMON's built-in labels. You can combine the two on a trap function command line like this:

```
000 FFF @"ApplZone" @"ApplLimit"
```

These options for the trap intercepting functions will cause them to take effect for all traps, but only for PC values between the contents of ApplZone and the contents of ApplLimit, i.e. within the application zone.

Breakpoints in Unloaded Segments

If you're writing a multi-segment application, something that you may have come across is the necessity to set a breakpoint at some point in the program that isn't within a loaded code segment. You may have spent some time wondering what to do. The answer is to use the user area's ability to load a resource to load in the CODE resource that you need. Even though the intersegment jump table hasn't been updated when you do that, the code is where it will be when the jump table *is* updated. You may therefore set breakpoints in it.

Is TMON Installed?

Every now and then you may have to write a program that needs to know whether TMON is installed or not. For example, during debugging you may wish to place _Debugger traps at important points in the program. _Debugger will crash the system with a bomb box if no debugger is installed, so you need to execute _Debugger only if TMON is around. You can determine if TMON is installed by looking at the longword stored at $FC and, if it is in RAM and is even, looking at the word stored there. If the word is equal to "WH" then TMON is installed. ("WH" are TMON's signature bytes, and are Waldemar Horwat's initials.) The word following the "WH" is TMON's internal version number.

Running Out of Room in Label Tables

If you are developing a large application and using .MAP files to provide TMON with labels for your code, you may find that you can't allocate a label table large enough to hold all of the labels. If this is the case, use a disk block editing tool such as FEdit to edit your volume's boot blocks. You can increase the size of the system heap that way. Change it to allow you to allocate however much space you need. Be aware that you also must change the version number of the boot blocks so that it is at least $21 on 128K ROMs or newer, otherwise the changes will be ignored.

Walking Through the VBL Queue

TMON allows you to configure it so that VBL tasks remain running while TMON is active. Sometimes there are undesirable interactions between TMON and VBL tasks that are active. It's nice to be able to track down all currently active VBL tasks. The way to do this is to walk through the VBL queue.

First open two windows, a dump window and a disassembly window. Make sure that the dump window is active and position the cursor in the address field (you can do this by clicking anywhere in the window and pressing the Tab key). Instead of typing an address or anchoring to a register, type @<"VBLQueue"+2> and press the Return key. This will cause the dump window to display starting at the first VBL record in the queue.

The first longword is a pointer to the next VBL record in the queue. We'll use it in a moment. Another important field is six bytes (counting from zero) past the beginning of the record. It's the address of the VBL task. If you activate the disassembly window and position the cursor in the address field, you can simply type @<*VBLRecord*+6> replacing *VBLRecord* with whatever address is showing in the address field of the dump window. In other words, if the dump window is dumping from $17D6, then type @<17D6+6> in the disassembly window. This will disassemble the VBL task. To continue through the queue, just enter the first longword shown in the VBL record as the address to dump from. If the longword at $17D6 is $00002596, then enter 2596 (or @17D6) as the dump address. You can then enter @<2596+6> in the disassembly window to show the next VBL task. This may be continued until the first longword of the VBL record is zero, indicating the end of the queue.

Note that if you know the entry number of a VBL record in the queue, you can dump it immediately by prefixing the dump expression with the correct number of indirection operators. In other words, if you want to dump the fourth VBL record in the queue, type @@@@<"VBLQueue"+2> as the dump address. For a complete explanation of what these expressions mean, see the "Numbers" section of the Technical Reference Manual.

TMON and Context-Switching Environments

As a rule, TMON works well with environments like Switcher and Finder 6.0 as long as TMON is installed before the context-switching system software. TMON Startup was designed to load TMON in these circumstances. If you wish to use TMON with Switcher or Finder 6.0, use TMON Startup.

ResumeProc Functions

Every once in a great while an application will implement a ResumeProc. A ResumeProc is a routine which handles the "Resume" button on the bomb box that shows up when something is *really* wrong. Most applications don't do this.

If you are using TMON with an application that has installed a ResumeProc and a system error occurs, you'll wind up in TMON rather than at a bomb box. TMON has no obvious Resume button, so what you *can* do is this: open the "Regs" window, position the cursor over the PC value, and type @"ResumeProc" quotes and all. Before you do this, you should make sure that the value at "ResumeProc" is non-zero. Once you've set the PC to @"ResumeProc" using the Exit function will pass control back to the application at the point that it would be if you had clicked on the "Resume" button in the bomb box.

The Mystical, Magical V and N Registers

Using the "Num" window always sets the N register to whatever the evaluated result is. This is handy for calculating values which can then be used in other expressions. An even handier register is V, because so many TMON functions set it. Many user area functions do, and it can also be set by pressing Return when the cursor is on a heap window line. For example, open an "Asmbly" window and anchor it to the V register. Now open the heap window, and find, for example, a "CDEF" resource. Place the cursor on that line in the heap window and press Return. The "Asmbly" window will now show a disassembly of that CDEF! Another example: if you have just used the Find function in the user area to search for _GetResource in "CODE0003," you can disassemble starting at the location where it was found by anchoring an "Asmbly" window to V.

What Version of the User Area Do I Have?

A common question with an extendable debugger with several customized user areas is: what version do I have? The easiest way to determine that (and, coincidentally, the size of the user area) is to open a "Num" window and type @USER with no quotes. N will be set to a longword consisting of the user area size (in the high word) and the user area version (in the low word). The 16-bit decimal value is the user area version.

Getting Through _LoadSeg Quickly

_LoadSeg is one of the traps that TMON refuses to step or GoSub through, because it never returns (it passes control to the segment being loaded). Your only choice, if you need to continue, is to Trace through it. If the segment in question has many entry points, this can take some time.

Apple provided a clever undocumented shortcut. If the byte at LoadTrap ($12D) is non-zero, a _Debugger trap will be executed before _LoadSeg passes control to the newly-loaded segment. Thanks to this, you can open a dump window (if you use 12D as the address the dump will actually be from 12C; be aware of this) and change the byte at 12D to something other than 00. Now go ahead and Exit TMON. TMON will be re-entered with the PC pointing to the instruction after the _Debugger trap in _LoadSeg. Stepping twice will put the PC at the first instruction of the entry point of the new segment. Don't forget to set 12D to 00 again afterward.

Tools for Getting Into TMON

A handy thing to have is a tool that gets you into TMON. Since the _Debugger trap will enter TMON, all that you need is an FKEY resource or a desk accessory that executes a _Debugger trap.

Viewing ROM Resources

Another nice thing to be able to do is to see what resources you have in your Macintosh's ROM. To do this, first use Trap Intercept to intercept a resource manager trap (_GetResource is a good choice). Use Trace to trace into the trap dispatcher in ROM. Now set the byte at "ROMMapInsert" to FF and the next byte to whatever the value of "ResLoad" is. Now use GoSub. If _GetResource is not in ROM, use GoSub until you get to ROM. Use GoSub to execute the first two subroutine calls. Now open the file window. GoSub again until a file reference number of 1 appears. File 1 is actually the ROM resource map, which you can view in the file window to your heart's content.

Saving Your File

Here is a reason to keep TMON loaded in memory at all times, even when you are not debugging a program. We have all probably experienced the agony of a system bomb just as we were about to save the document we had been working on for the last two hours. TMON will sometimes allow you to recover the lost data. The idea is to find the top of the event loop in the application you were running and restart it there. This will sometimes return you to the program in just a stable enough condition to save your file. First make sure that A5 is correct. Open the register window and type `@"CurrentA5"` in the A5 position. Now open the heap window and the user area window. Get the address and length of the first code resource. Do a search on `_GetNextEvent` for 2 bytes starting at the beginning of the code resource and going to the end. Repeat this for each code resource until you find `_GetNextEvent`. Open an assembly window to this address. If you start scrolling down the code, you will generally find a jump or branch (usually unconditional) to a point just above the `_GetNextEvent`. This is usually the top of the event loop. Set the PC in the register window to the address you just found. Hit the exit button and pray. If you find yourself back in your program, do a Save As... to a floppy disk and then re-boot. Remember that the state of the system is pretty uncertain, so save and get out as fast as possible. Good luck.

Index

ICOM SIMULATIONS, INC.

648 S. Wheeling Road
Wheeling, IL 60090
312/520-4440